

# Code Generation for Data Processing

Alexis Engelke

Winter 2025/26



# Contents

<b>1. Introduction and Interpretation</b>	<b>1</b>
1.1. Organization . . . . .	1
1.2. Overview . . . . .	4
1.3. High-Level Structure of Compilers . . . . .	5
1.4. Interpretation . . . . .	7
1.5. Context of Compilation . . . . .	12
<b>2. Compiler Front-end</b>	<b>15</b>
2.1. Lexing . . . . .	15
2.2. Parsing . . . . .	17
2.3. Semantic Analysis . . . . .	23
2.4. Miscellaneous . . . . .	24
<b>3. Intermediate Representations</b>	<b>27</b>
3.1. Motivation . . . . .	27
3.2. Control Flow Graph . . . . .	29
3.3. Other Graph IRs . . . . .	33
3.4. Linear IRs . . . . .	33
3.5. Single Static Assignment . . . . .	36
3.6. IR Design . . . . .	40
<b>4. LLVM-IR</b>	<b>45</b>
4.1. Overview . . . . .	45
4.2. LLVM-IR . . . . .	46
4.3. API . . . . .	51
<b>5. Analyses and Transformations</b>	<b>55</b>
5.1. Motivation . . . . .	55
5.2. Dead Code Elimination . . . . .	55
5.3. Sparse Conditional Constant Propagation . . . . .	57
5.4. Simple Common Subexpression Elimination . . . . .	59
5.5. Dominator Tree . . . . .	61
5.6. Common Subexpression Elimination . . . . .	63
5.7. Inlining . . . . .	64
5.8. Loop Analysis . . . . .	65
5.9. Loop Optimizations . . . . .	67
<b>6. Optimizations in LLVM</b>	<b>71</b>
6.1. Overview . . . . .	71

6.2. Canonicalization . . . . .	72
6.3. Flags . . . . .	77
6.4. Inter-Procedural Optimizations . . . . .	80
6.5. Running and Writing LLVM Passes . . . . .	81
<b>7. Vectorization</b>	<b>83</b>
<b>8. Instruction Selection</b>	<b>95</b>
8.1. Overview . . . . .	95
8.2. Macro Expansion . . . . .	96
8.3. Tree Covering . . . . .	98
8.4. DAG Covering . . . . .	102
8.5. Graph Covering . . . . .	104
8.6. ISel in Practice . . . . .	104
8.7. LLVM Instruction Selection . . . . .	105
<b>A. Exercise Solutions</b>	<b>111</b>

# 1. Introduction and Interpretation

## 1.1. Organization

### [Slide 2] Module “Code Generation for Data Processing”

#### Learning Goals

- Getting from an intermediate code representation to machine code
- Designing and implementing IRs and machine code generators
- Apply for: JIT compilation, query compilation, ISA emulation

#### Prerequisites

- Computer Architecture, Assembly ERA, GRA/ASP
- Databases, Relational Algebra GDB
- Beneficial: Compiler Construction, Modern DBs

### [Slide 3] Topic Overview

#### Introduction

- Introduction and Interpretation
- Compiler Front-end

#### Intermediate Representations

- IR Concepts and Design
- LLVM-IR
- Analyses and Optimizations

#### Compiler Back-end

- Instruction Selection
- Register Allocation
- Linker, Loader, Debuginfo

#### Applications

- JIT-compilation + Sandboxing
- Query Compilation
- Binary Translation

### [Slide 4] Lecture Organization

- Lecturer: Dr. Alexis Engelke [engelke@in.tum.de](mailto:engelke@in.tum.de)
- Time slot: Thu 10-14, 02.11.018
- Material: <https://db.in.tum.de/teaching/ws2526/codegen/>

### **Exam**

- Written exam, 90 minutes, **no retake**, 2026-02-27 14:00
- (Might change to oral on very low registration count)

### **[Slide 5] Exercises**

- Regular homework, often with programming exercise
  - Submission via POST request (see assignments)
    - Grading with  $\{*, +, \sim, -\}$ , feedback on best effort
- \* Grade \*: excellent submission, goes beyond expectations
  - \* Grade +: good submission, meets expectations
  - \* Grade  $\sim$ : acceptable submission, but is below expectations
  - \* Grade -: unacceptable submission, too far below expectations
- Exercises integrated into lecture
  - Hands-on programming or analysis of systems (needs laptop)
  - Occasionally: present and discuss homework solutions

### **Grade Bonus**

- Requirement:  $N - 2$  “sufficiently working” homework submissions and one presentations of homework in class (depends on submission count)
- Bonus: grades in  $[1.3; 4.0]$  improved by 0.3/0.4

### **[Slide 6] Why study compilers?**

- Critical component of every system, functionality and performance
  - Compiler mostly *alone* responsible for using hardware well
- Brings together many aspects of CS:
  - Theory, algorithms, systems, architecture, software engineering, (ML)
- New developments/requirements pose new challenges
  - New architectures, environments, language concepts, ...
- High complexity!

**[Slide 7] Compiler Lectures @ TUM**

<b>Compiler Construction</b> IN2227, SS, THEO Front-end, parsing, semantic analyses, types	<b>Program Optimization</b> IN2053, WS, THEO Analyses, transformations, abstract interpretation	<b>Virtual Machines</b> IN2040, SS, THEO Mapping programming paradigms to IR/bytecode
<b>Programming Languages</b> CIT3230000, WS Implementation of advanced language features	<b>Code Generation</b> CIT3230001, WS Back-end, machine code generation, JIT comp.	

**[Slide 8] Why study code generation?**

- Frameworks (LLVM, ...) exist and are comparably good, but often not good enough (performance, features)
  - Many systems with code gen. have their own back-end
  - E.g.: V8, WebKit FTL, .NET RyuJIT, GHC, Zig, QEMU, Umbra, ...
- Machine code is not the only target: bytecode
  - Often used for code execution
  - E.g.: V8, Java, .NET MSIL, BEAM (Erlang), Python, MonetDB, eBPF, ...
  - Allows for flexible design
  - But: efficient execution needs machine code generation

**[Slide 9] Proebsting's Law**

“Compiler advances double computing power every 18 years.”

– Todd Proebsting, 1998<sup>a</sup>

<sup>a</sup><http://proebsting.cs.arizona.edu/law.html>

- Still optimistic; depends on number of abstractions

The performance increases compilers can make on existing code are typically low. However, optimizing compilers gain more abilities in simplifying needlessly complex code, enabling the use of more abstractions and therefore higher level code. These abstractions are removed/optimized during compilation, enabling languages to promote these as *zero-cost abstractions*. They do, however, have a cost: compile times.

Also note that some of these “zero-cost” abstractions actually *do* have some run-time cost. For example, the mere possibility of C++ exceptions can cause less efficient

machine code and might prevent optimizations due to the more complex control flow possibilities.

### 1.2. Overview

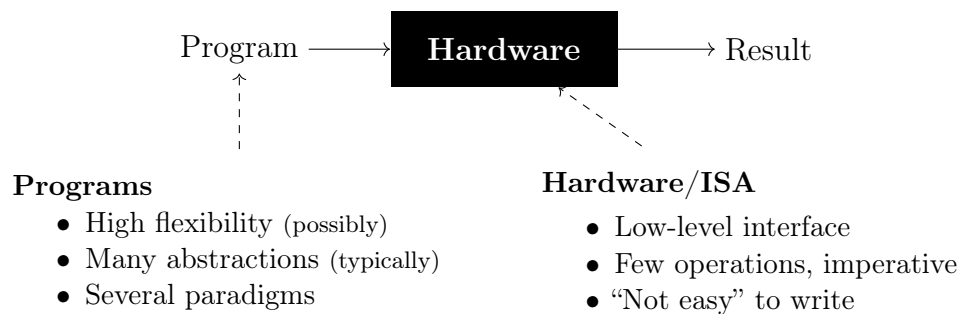
#### [Slide 10] Motivational Example: Brainfuck

- Turing-complete esoteric programming language, 8 operations
  - Input/output: `.` `,`
  - Moving pointer over infinite array: `<` `>`
  - Increment/decrement: `+` `-`
  - Jump to matching bracket if (not) zero: `[` `]`

++++++[->+++++<]>.

- Execution with pen/paper? ☹

#### [Slide 11] Program Execution



#### [Slide 12] Motivational Example: Brainfuck – Interpretation

- Write an interpreter!

```
unsigned char state[10000];
unsigned ptr = 0, pc = 0;
while (prog[pc])
    switch (prog[pc++]) {
        case '.': putchar(state[ptr]); break;
        case ',': state[ptr] = getchar(); break;
        case '>': ptr++; break;
        case '<': ptr--; break;
        case '+': state[ptr]++; break;
        case '-': state[ptr]--; break;
        case '[': state[ptr] || (pc = matchParen(pc, prog)); break;
        case ']': state[ptr] && (pc = matchParen(pc, prog)); break;
    }
```



**[Slide 13] Program Execution****Compiler**

- Translate program to other lang.
- Might optimize/improve program

- C, C++, Rust → machine code
- Python, Java → bytecode

Multiple compilation steps can precede the “final interpretation”

**Interpreter**

- Directly execute program
- Computes program result

- Shell scripts, Python bytecode, machine code (conceptually)

## 1.3. High-Level Structure of Compilers

**[Slide 14] Compilers**

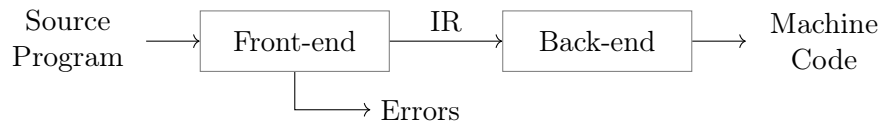
- Targets: machine code, bytecode, or other source language
- Typical goals: better language usability and performance
  - Make lang. usable at all, faster, use less resources, etc.
- Constraints: specs, resources (comp.-time, etc.), requirements (perf., etc.)
- Examples:
  - “Classic” compilers source → machine code
  - JIT compilation of JavaScript, WebAssembly, Java bytecode, ...
  - Database query compilation
  - ISA emulation/binary translation

**[Slide 15] Compiler Structure: Monolithic**

- Inflexible architecture, hard to retarget

Some languages like C are designed to be compilable in a single pass without building any intermediate representation of the code between source and assembly. Single-pass compilers exist, but often have very limited possibilities to transform the code. They might not even know basic code properties, e.g., the size of the stack frame, during compilation of a function.

**[Slide 16] Compiler Structure: Two-phase architecture**



Front-end

- Parses source code
- Detect syntax/semantical errors
- Emit *intermediate representation* encode semantics/knowledge
- Typically:  $\mathcal{O}(n)$  or  $\mathcal{O}(n \log n)$

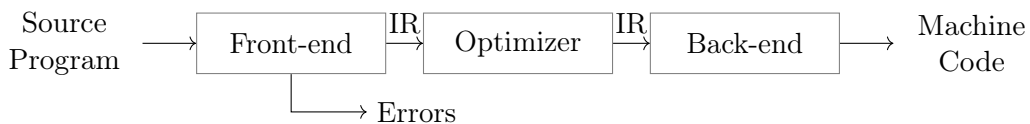
Back-end

- Translate IR to target architecture
- Can assume valid IR ( $\leadsto$  no errors)
- Possibly one back-end per arch.
- Contains  $\mathcal{NP}$ -complete problems

After parsing, all information is encoded in the IR, including references to source code constructs for debugging support. The input source code is (at least conceptually) no longer needed.

In practice, there are very rare cases where the back-end can also raise errors. This can happen, for example, when some very architecture-specific constraints might be hard to verify during parsing (e.g., inline assembly constraints in combination with available registers).

**[Slide 17] Compiler Structure: Three-phase architecture**



- Optimizer: analyze/transform/rewrite program inside IR
- Conceptual architecture: real compilers typically much more complex
  - Several IRs in front-end and back-end, optimizations on different IRs
  - Multiple front-ends for different languages
  - Multiple back-ends for different architectures

Example Clang/LLVM (will be covered in more detail later): Clang parses the input into an abstract syntax tree (IR 1), uses this for semantic analyses; then Clang transforms the code into LLVM-IR (IR 2), which is primarily used for optimization; then the LLVM back-end transforms the code further into LLVM's Machine IR (IR 3), executes some low-level optimizations and register allocation there; the assembly

printer of the back-end then lowers the code further to LLVM's machine code representation (IR 4), before finally emitting machine code. Some optimizations inside this pipeline, e.g. vectorization, might even build further representation of the code.

Why are compilers using so many different code representations? Different transformations work best at different abstraction levels. Diagnosing unused variables, for example, requires information about the source code. Optimization of arithmetic computations is easier in a data-flow-focused representation, where no explicit variables exist. Low-level modifications, like folding operations into complex addressing modes of the ISA, need a code representation where ISA instructions are already present.

### [Slide 18] Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc. *Re*
  - Example: `a+b*c`  $\rightarrow$  `ID(a) PLUS ID(b) TIMES ID(c)`
2. Parser: build (abstract) syntax tree, check for syntax errors *CFG*
  - Syntax Tree: describe grammatical structure of complete program Example: `expr("a", op("+"), expr("b", op("*"), expr("c")))`
  - Abstract Syntax Tree: only relevant information, more concise Example: `plus("a", times("b", "c"))`
3. Semantic Analysis: check types, variable existence, etc.
4. IR Generator: produce IR for next stage
  - This might be the AST itself

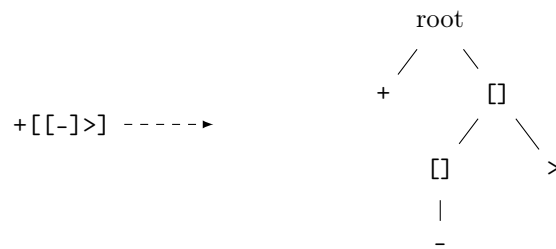
### [Slide 19] Compiler Back-end

1. Instruction Selection: map IR operations to target instructions
  - Use target features: special insts., addressing modes, ...
  - Still using virtual/unlimited registers
2. Instruction Scheduling: optimize order for target arch.
  - Start memory/high-latency earlier, etc.
  - Requires knowledge about micro-architecture
3. Register Allocation: map values to fixed register set/stack
  - Use available registers effectively, minimize stack usage

## 1.4. Interpretation

### [Slide 20] Motivational Example: Brainfuck – Front-end

- Need to skip comments
- Bracket searching is expensive/redundant
- Idea: “parse” program!
- Tokenizer: yield next operation, skipping comments
- Parser: find matching brackets, construct AST



### [Slide 21] Motivational Example: Brainfuck – AST Interpretation

- AST can be interpreted recursively

```
struct node { char kind; unsigned cldCnt; struct node* cld; };
struct state { unsigned char* arr; size_t ptr; };
void donode(struct node* n, struct state* s) {
    switch (n->kind) {
        case '+': s->arr[s->ptr]++; break;
        // ...
        case '[': while (s->arr[s->ptr]) children(n, s); break;
        case 0: children(n, s); break; // root
    }
}
void children(struct node* n, struct state* s) {
    for (unsigned i = 0; i < n->cldCnt; i++) donode(n->cld + i, s);
}
```

### [Slide 22] Motivational Example: Brainfuck – Optimization

- Inefficient sequences of `+/-</>` can be combined
  - Trivially done when generating IR
- Fold patterns into more high-level operations

---

#### In-Class Exercise:

Look at some Brainfuck programs. Which patterns are beneficial to fold?

*Solution on page 111.*

---

### [Slide 23] Motivational Example: Brainfuck – Optimization

- Fold offset into operation
  - `right(2) add(1) = addoff(2, 1) right(2)`
  - Also possible with loops
- Analysis: does loop move pointer?
  - Loops that keep position intact allow more optimizations
  - Maybe distinguish “regular loops” from arbitrary loops?
- Get rid of all “effect-less” pointer movements
- Combine arithmetic operations, disambiguate addresses, etc.

**[Slide 24] Motivational Example: Brainfuck – Bytecode**

- Tree is nice, but rather inefficient  $\leadsto$  flat and compact bytecode
- Avoid pointer dereferences/indirections; keep code size small
- Maybe dispatch two instructions at once?
  - `switch (ops[pc] | ops[pc+1] << 8)`
- Superinstructions: combine common sequences to one instruction

Dispatching multiple instructions at once can be problematic due to the explosion of cases that need to be implemented (often results in large jump tables and lots of code with resulting cache misses and branch mispredictions). Often, it is advisable to not always switch over multiple neighbored instructions, but instead combine common sequences into superinstructions.

**[Slide 25] Threaded Interpretation<sup>1</sup>**

- Simple `switch-case` dispatch has lots of branch misses
- Threaded interpretation: at end of a handler, jump to next op

```
struct op { char op; char data; };
struct state { unsigned char* arr; size_t ptr; };
void threadedInterp(struct op* ops, struct state* s) {
    static const void* table[] = { &CASE_ADD, &CASE_RIGHT, };
#define DISPATCH do { goto *table[(++pc)->op]; } while (0)

    struct op* pc = ops;
    DISPATCH;

CASE_ADD: s->arr[s->ptr] += pc->data; DISPATCH;
CASE_RIGHT: s->arr += pc->data; DISPATCH;
}
```

With threaded interpretation there is not a single indirect jump instruction inside the dispatcher, but one indirect jump instruction per operation. Each of these indirect jumps then occupies a different branch prediction slot in the CPU. If an operation of type X is typically followed by an operation of type Y, with threaded interpretation the CPU has a much better chance of correctly predicting the dispatch branch to the next operation, because the indirect jump at the end of operation X typically jumps to operation Y. Without threaded interpretation, there would be only a single indirect branch, which is much harder to predict.

Threaded interpretation is especially useful on older and less powerful CPUs. Recent CPUs (e.g., Intel since Skylake, AMD since Zen 3, Apple Silicon) store the history of branches and use this for better prediction. On such processors, threaded interpretation might not improve performance (or gains might be lower).

<sup>1</sup>MA Ertl and D Gregg. “The structure and performance of efficient interpreters”. In: *JILP* 5 (2003), pp. 1–25. URL: <http://www.jilp.org/vol5/v5paper12.pdf>.

### [Slide 26] Threaded Interpretation with Tail Calls

- Threaded interpretation can also be implemented with tails calls

```
struct op { char op; char data; };
struct state { unsigned char* arr; size_t ptr; };
void tcInterp(struct op* pc, struct state* s);
static void fn_add(struct op* pc, struct state* s) {
    s->arr[s->ptr] += pc->data; tcInterp(pc + 1, s); }
static void fn_right(struct op* pc, struct state* s) {
    s->arr += pc->data; tcInterp(pc + 1, s); }
void tcInterp(struct op* pc, struct state* s) {
    typedef void (*Fn)(struct op* pc, struct state* s);
    static const Fn fns[] = { fn_add, fn_right };
    fns[pc->op](pc, s);
}
```

Computed goto is a GNU extension and similar features are not available in many other languages. However, it is possible to achieve a similar effect through indirect tail calls<sup>a</sup>.

In this example<sup>b</sup>, the resulting assembly is almost identical for x86-64, but for AArch64, some more engineering effort would be required.

The code as written has another problem: At -O0, no tail call optimization will be performed. Some compilers like Clang provide a way to force tail calls in C/C++ code:

```
void tcInterp(struct op* pc, struct state* s) {
    // ... (and adapt individual functions likewise)
    [[clang::musttail]] return fns[pc->op](pc, s);
}
```

In the context of functional programming, this style is also referred to as *continuation-passing style*.

<sup>a</sup>A tail call is a call at the end of a function without adding a new entry to the stack frame.

<sup>b</sup><https://godbolt.org/z/E5drG61aK>

### [Slide 27] Direct Threading<sup>2</sup>

- Use function pointer as operation to avoid indirection

```
struct op; struct state;
typedef void (*Fn)(struct op* pc, struct state* s);
struct op { Fn op; char data; };
struct state { unsigned char* arr; size_t ptr; };

void fn_add(struct op* pc, struct state* s) {
    s->arr[s->ptr] += pc->data; pc[1].op(pc + 1, s); }
void fn_right(struct op* pc, struct state* s) {
    s->arr += pc->data; pc[1].op(pc + 1, s); }
void dtInterp(struct op* ops, struct state* s) {
    ops[0].op(ops, s);
}
```

---

<sup>2</sup>JR Bell. “Threaded Code”. In: *CACM* 16.6 (1973), pp. 370–372. URL: <https://dl.acm.org/doi/pdf/10.1145/362248.362270>.

While there might be a benefit of reducing the instruction count, the effect is typically insignificant on out-of-order CPUs, as the extra load instruction is typically a cache hit and the latency is hidden by other operations.

### [Slide 28] Threaded Interpretation — Comparison

---

#### In-Class Exercise:

What are benefits/drawbacks of the three threading approaches<sup>a</sup>?

- Indirect threading with computed `goto`
- Indirect threading with tail calls
- Direct threading with tail calls

*Solution on page 111.*

---

<sup>a</sup><https://godbolt.org/z/4rcEv4sqj>

---

- Some differences on code size, readability, and maintainability
- Performance: depends on hardware, context – always measure!

### [Slide 29] Fast Interpretation

- Key technique to “avoid” compilation to machine code
- Preprocess program into efficiently executable bytecode
  - Easily identifiable opcode, homogeneous structure
  - Can be linear (fast to execute), but trees also work
  - Match bytecode ops with needed operations  $\rightsquigarrow$  fewer instructions
  - Larger operations preferable, but not too many (prediction)
- Perhaps optimize – if it’s worth the benefit
  - Fold constants, combine instructions, ...
  - Consider superinstructions for common sequences
- For very cold code: avoid transformations at all

### [Slide 30] Interpretation vs. Compilation

Fundamental benefits of compilation:

- Elimination of interpreter dispatch
  - Can be significant for bytecodes with many small operations
- Use of CPU registers for values across bytecode instructions
  - Interpreter: most values must be stored in memory
  - Register allocation can bring large improvements

This is a major driver for performance even on modern CPUs, which have two related optimizations: Store-to-load forwarding, where the value from a store will be forwarded to a later load from the same address; and memory renaming,

where the physical register of a stored value is kept and a subsequent load with a similar address operand speculatively reuses that register. However, CPU resources for these optimizations are limited and using registers directly is much more efficient.

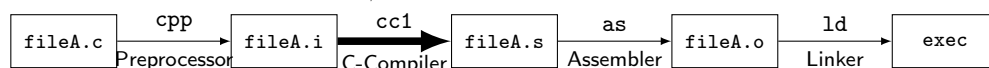
- No fundamental benefit: optimizations
  - Many optimizations can also be applied on bytecode

Fundamental benefits of interpretation: simple, portable

### 1.5. Context of Compilation

#### [Slide 31] Compiler: Surrounding – Compile-time

- Typical environment for a C/C++ compiler:



- Calling Convention: interface with other objects/libraries
- Build systems, dependencies, debuggers, etc.
- Compilation target machine (hardware, VM, etc.)

#### [Slide 32] Compiler: Surrounding – Run-time

- OS interface (I/O, ...)
- Memory management (allocation, GC, ...)
- Parallelization, threads, ...
- VM for execution of virtual assembly (JVM, ...)
- Run-time type checking
- Error handling: exception unwinding, assertions, ...
- Reflection, RTTI

#### [Slide 33] Motivational Example: Brainfuck – Runtime Environment

- Needs I/O for . and ,
- Error handling: unmatched brackets
- Memory management: infinitely-sized array

---

#### In-Class Exercise:

How to efficiently emulate an infinitely sized array?

*Solution on page 111.*

---

#### [Slide 34] Compilation point: AoT vs. JIT

Ahead-of-Time (AoT)

- All code has to be compiled



- No dynamic optimizations
- Compilation-time secondary concern

Just-in-Time (JIT)

- Compilation-time is critical
- Code can be compiled on-demand
  - Incremental optimization, too
- Handle cold code fast
- Dynamic specializations possible
- Allows for `eval()`

Various hybrid combinations possible

**[Slide 35] Introduction and Interpretation – Summary**

- Compilation vs. interpretation and combinations
- Compilers are key to usable/performant languages
- Target language typically machine code or bytecode
- Three-phase architecture widely used
- Interpretation techniques: bytecode, threaded interpretation, ...
- JIT compilation imposes different constraints

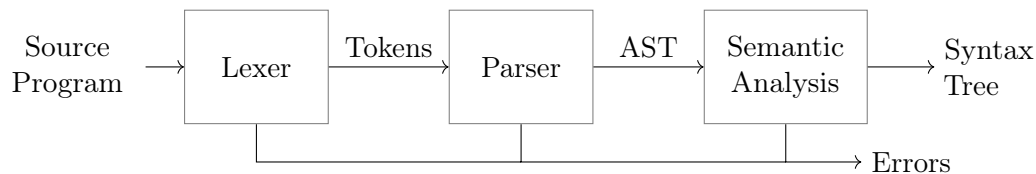
**[Slide 36] Introduction and Interpretation – Questions**

- What is typically compiled and what is interpreted? Why?
  - PostScript, C, JavaScript, HTML, SQL
- What are typical types of output languages of compilers?
- How does a compiler IR differ from the source input?
- What is the impact of the language paradigm on optimizations?
- What are important factors for an efficient interpreter?
- What are inherent benefits of compilation over interpretation?
- What are key differences between AoT and JIT compilation?



## 2. Compiler Front-end

### [Slide 38] Compiler Front-end



- Typical architecture: separate lexer, parser, and context analysis
  - Allows for more efficient lexical analysis
  - Smaller components, easier to understand, etc.
- Some languages: preprocessor and macro expansion

### 2.1. Lexing

#### [Slide 39] Lexer

- Convert stream of chars to stream of words (*tokens*)
- Detect/classify identifiers, numbers, operators, ...
- Strip whitespace, comments, etc.

`a+b*c` → `ID(a) PLUS ID(b) TIMES ID(c)`

- Typically representable as regular expressions

#### [Slide 40] Typical Token Kinds

- Punctuators `( ) [ ] { } ; = + += | ||`
- Identifiers `abc123 main`
- Keywords `void int __asm__`
- Numeric constants `123 0xab1 5.7e3 0x1.8p1 09.1f`
- Char constants `'a' u'œ'`
- String literals `"abc\x12\n"`
- Internal `EOF COMMENT UNKNOWN INDENT DEDENT`
  - Comments might be useful for annotations, e.g. `// fallthrough`

Indentation-based languages like Python need separate tokens for indent/dedent, the indentation level is tracked in the lexer. Parsing numbers may need special care to correctly handle all possible cases of integer and floating-point numbers.

### [Slide 41] Lexer Implementation

```
struct Token { enum Kind { IDENT, EOF, PLUS, PLUSEQ, /*...*/ };
    std::string_view v; Kind kind; };
Token next(std::string_view v) {
    if (v.empty()) return Token{v, Token::EOF};
    if (v.starts_with("+=")) return Token{"+"sv, Token::PLUSEQ};
    if (v.starts_with("+")) return Token{"+"sv, Token::PLUS};
    switch (v[0]) {
    case ' ', '\n', '\t': return next(v.substr(1)); // skip whitespace
    case 'a' ... 'z', 'A' ... 'Z', '_': {
        Token t = // ... parse identifier, e.g. using regex
        if (auto kind = isKeyword(t.v)) return Token{*kind, t.v};
        return t;
    }
    case '0' ... '9': // ... parse number
    default: return Token{v.substr(0, 1), Token::ERROR};
    }
}
```

This is just a minimal and non-optimized implementation to illustrate the concept. Performance-focused implementations do not use explicit regular expressions but write the state machine into code.

The struct `Token` has room for improvement. First, a `string_view` is unnecessarily large with 16 bytes, most tokens are smaller than  $2^{16}$  bytes. Some tracking of the source locations is advisable for attaching diagnostics to their origin inside the code, for example by storing a file ID and the byte offset into the file. By tracking the byte offsets of line breaks, the line number can be reconstructed in  $\mathcal{O}(\log n)$  from the byte offset.

Another optimization strategy is string interning, where identifiers are converted into unique integers (or pointers) during parsing. During later phases, comparing interned strings is much more efficient, as it is just an integer/pointer comparison. Another benefit is that the entire input file does not need to be kept in memory during parsing.

### [Slide 42] Lexing C??=

```
main() <%
    // yay, this is C99??/
    puts("hi_world!");
    puts("what's_up??!");
%>
```

Output: what's up|

- Trigraphs for systems with more limited encodings/char sets
- Digraphs to provide a more readable alternative...

Besides digraphs, trigraphs, and the preprocessor, C has another weird property: identifier names can be split by `\`, which concatenates two lines. It is necessary to construct the “real” identifier first. To simplify memory management in such cases, a bump pointer allocator (allocate large chunks of memory from the OS, then simply bump the end pointer for every allocation) can be useful to store such constructed names.

#### [Slide 43] Lexer Implementation

- Essentially a DFA (for most languages)
  - Set of regexes  $\rightarrow$  NFA  $\rightarrow$  DFA
- Respect whitespace/separators for operators, e.g. `+` and `+=`
- Automatic tools (e.g., flex) exist; most compilers do their own
- Keywords typically parsed as identifiers first
  - Check identifier if it is a keyword; can use perfect hashing
- Other practical problems
  - UTF-8 homoglyphs; trigraphs; pre-processing directives

A tool to generate perfect hash tables from a set of keywords is gperf. Example, compile with `gperf -L C++ -C -E -t <input>`:

```
struct keyword {char* name; int val; }
%%
int, 1
char, 2
void, 3
if, 4
else, 5
while, 6
return, 7
```

## 2.2. Parsing

#### [Slide 44] Parsing

- Convert stream of tokens into (abstract) syntax tree
  - Most programming languages are context-sensitive
    - Variable declarations, argument count, type match, etc.  $\rightsquigarrow$  separated into semantic analysis
- Syntactically valid: `void foo = doesntExist / "abc";`
- Grammar usually specified as CFG

#### [Slide 45] Context-Free Grammar (CFG)

- Terminals: basic symbols/tokens
- Non-terminals: syntactic variables

- Start symbol: non-terminal defining language
- Productions: non-terminal  $\rightarrow$  series of (non-)terminals

```
stmt   $\rightarrow$  whileStmt | breakStmt | exprStmt
whileStmt  $\rightarrow$  while ( expr ) stmt
breakStmt  $\rightarrow$  break ;
exprStmt  $\rightarrow$  expr ;
expr    $\rightarrow$  expr + expr | expr * expr | expr = expr | ( expr ) | number
```

### [Slide 46] Hand-written Parsing – First Try

- One function per non-terminal
- Check expected structure
- Return AST node
- Need look-ahead!

```
NodePtr parseBreakStmt() {
    consume(Token::BREAK);
    consume(Token::SEMICOLON);
    return newNode(Node::BreakStmt);
}
NodePtr parseWhileStmt() {
    consume(Token::WHILE);
    consume(Token::LPAREN);
    NodePtr expr = parseExpr();
    consume(Token::RPAREN);
    NodePtr body = parseStmt();
    return newNode(Node::WhileStmt,
        {expr, body});
}
NodePtr parseStmt() {
    // whoops!
}
```

### [Slide 47] Hand-written Parsing – Second Try

- Need look-ahead to distinguish production rules
- Consequences for grammar:
  - No left-recursion
  - First  $n$  terminals must allow distinguishing rules
  - $LL(n)$  grammar;  $n$  typically 1 $\Rightarrow$  Not all CFGs (easily) parseable (but most programming langs. are)
- Now... expressions

```
NodePtr parseBreakStmt() { /*...*/ }
NodePtr parseWhileStmt() { /*...*/ }

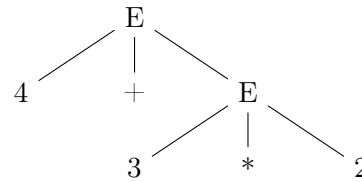
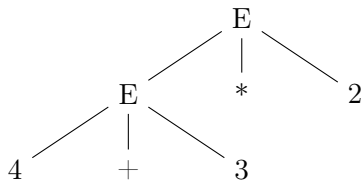
NodePtr parseStmt() {
    Token t = peekToken();
    if (t.kind == Token::BREAK)
        return parseBreakStmt();
}
```

```

if (t.kind == Token::WHILE)
    return parseWhileStmt();
// ...
NodePtr expr = parseExpr();
consume(Token::SEMICOLON);
return newNode(Node::ExprStmt,
    {expr});
}

```

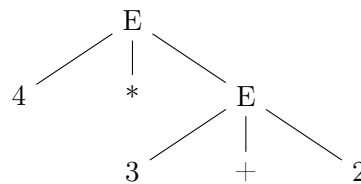
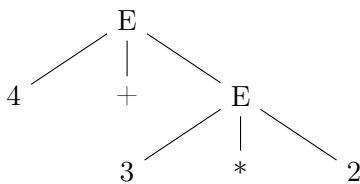
**[Slide 48] Ambiguity**

$$expr \rightarrow expr + expr \mid expr * expr \mid expr = expr \mid ( expr ) \mid \text{number}$$
Input:  $4 + 3 * 2$ 

The grammar, as specified, is ambiguous, there are two possible ways to parse the input.

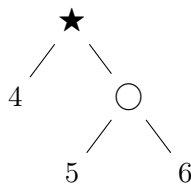
**[Slide 49] Ambiguity – Rewrite Grammar?**

$$primary \rightarrow ( expr ) \mid \text{number}$$

$$expr \rightarrow primary + expr \mid primary * expr \mid primary = expr \mid primary$$
Input:  $4 + 3 * 2$ Input:  $4 * 3 + 2$ 

The grammar is no longer ambiguous, but the result might not be expected, conventionally, multiplication has a stronger binding than addition.

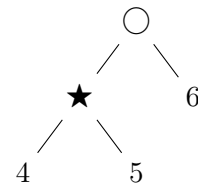
**[Slide 50] Ambiguity – Precedence**Input:  $4 \star 5 \bigcirc 6$



If  $\text{prec}(\bigcirc) > \text{prec}(\star)$  or equal prec.  
and  $\star$  is right-associoc.

Examples:

- $4 + 5 \cdot 6$  ( $\text{prec}(\cdot) > \text{prec}(+)$ )
- $a = b = c$  ( $=$  is right-associoc.)  
     $b = c$  should be executed first



If  $\text{prec}(\bigcirc) < \text{prec}(\star)$  or equal prec.  
and  $\star$  is left-associoc.

Examples:

- $4 + 5 < 6$  ( $\text{prec}(<) < \text{prec}(+)$ )
- $a + b - c$  ( $+$  is left-associoc.)  
     $a + b$  should be executed first

### [Slide 51] Hand-written Parsing – Expression Parsing

- Start with basic expr.:
  - Number, variable, etc.
  - Parenthesized expr.
    - Parse full expression
    - Next token must be )
- Unary expr: followed by expr. with higher prec.
  - $- < \text{unary} - < [] / ->$

```
NodePtr parseExpr(unsigned minPrec=0);
NodePtr parsePrimaryExpr() {
    switch (Token t = next(); t.kind) {
        case Token::IDENT:
            return makeNode(Node::IDENT, t.v);
        case Token::NUMBER: // ...
        case Token::MINUS:
            // Only exprs with high precedence
            return makeNode(Node::UMINUS,
                {parseExpr(UNARY_PREC)});
        case Token::LPAREN: // ...
            // ...
    }
}
```

### [Slide 52] Hand-written Parsing – Expression Parsing

- Only allow ops. with higher prec. on the right child
  - Right-associoc.: allow same
- Lower prec.: return + insert higher up in the tree

```
OpDesc OPS[] = { // {prec, rassoc}
    [Token::MUL] = {12, false},
    [Token::ADD] = {11, false},
```



```
[Token::EQ] = {2, true},
[Token::QUEST] = {3, true}, // ?:
}
NodePtr parseExpr(unsigned minPrec=1) {
    auto lhs = parsePrimaryExpr();
    while (auto op = OPS[next().kind];
           op.prec >= minPrec) {
        // ... handle (, [, ?: ...
        auto newPrec = op.rassoc ?
            op.prec : op.prec + 1;
        auto rhs = parseExpr(newPrec);
        lhs = makeNode(op.nodeKind,
            {lhs, rhs});
    }
    return lhs;
}
```

---

**In-Class Exercise:** $a = 3 * 2 + 1;$  $a = b + c + d = 1;$  $a ? 1 : b ? 2 : 3;$ *Solution on page 112.*

---

**[Slide 53] Top-down vs. Bottom-up Parsing**

## Top-down Parsing

- Start with top rule
- Every step: choose expansion
- LL(1) parser
  - Left-to-right, Leftmost Derivation
- “Easily” writable by hand
- Error handling rather simple
- Covers many prog. languages

## Bottom-up Parsing

- Start with text
- Reduce to non-terminal
- LR(1) parser
  - Left-to-right, Rightmost Derivation
  - Strict super-set of LL(1)
- Often: uses parser generator
- Error handling more complex
- Covers nearly all prog. languages

**[Slide 54] Parser Generators**

- Writing parsers by hand can be large effort
- Parser generators can simplify parser writing a lot
  - Yacc/Bison, PLY, ANTLR, ...

- Automatic generation of parser/parsing tables from CFG
  - Finds ambiguities in the grammar
  - Lexer often written by hand
- Used heavily in practice, unless error handling is important

### [Slide 55] Bison Example – part 1

```
%define api.pure full
%define api.value.type {ASTNode*}
%param { Lexer* lexer }
%code{
static int yylex(ASTNode** lvalp, Lexer* lexer);
}
%token NUMBER
%token WHILE "while"
%token BREAK "break"

// precedence and associativity
%right '='
%left '+'
%left '*'
```

### [Slide 56] Bison Example – part 2

```
%%
stmt : WHILE '(' expr ')' stmt { $$ = mkNode(WHILE, $1, $2); }
      | BREAK ';' { $$ = mkNode(BREAK, NULL, NULL); }
      | expr ';' { $$ = $1; }
      ;
expr : expr '+' expr { $$ = mkNode('+', $1, $2); }
      | expr '*' expr { $$ = mkNode('*', $1, $2); }
      | expr '=' expr { $$ = mkNode('=', $1, $2); }
      | '(' expr ')' { $$ = $1; }
      | NUMBER
      ;
%%
static int yylex(ASTNode** lvalp, Lexer* lexer) {
    /* return next token, or YYEOF/... */ }
```

Compile with `bison -dg input.ypp`, it will emit a C++ header, the implementation file, and also a graph showing the state machine of the parser.

### [Slide 57] Parsing in Practice

- Some use parser generators, e.g. Python some use hand-written parsers, e.g. GCC, Clang, Swift, Go
- Optimization of grammar for performance
  - Rewrite rules to reduce states, etc.
- Useful error-handling: complex!
  - Try skipping to next separator, e.g. `;` or `,`

- Programming languages are not always context-free
  - C: `foo* bar;`
  - May need to break separation between lexer and parser

In fact, many compilers<sup>a</sup> use hand-written parsers, because they allow for better error messages a more graceful handling of syntax errors, leading to more reported errors during a single (failing) compilation.

<sup>a</sup><https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html>

#### [Slide 58] Parsing C++

- C++ is not context-free (inherited from C): `T * a;`
- C++ is ambiguous: `Type (a), b;`
  - Can be a declaration or a comma expression
- C++ templates are Turing-complete<sup>1</sup>
- C++ *parsing* is hence *undecidable*<sup>2</sup>
  - Template instantiation combined with `C T * a` ambiguity

## 2.3. Semantic Analysis

#### [Slide 59] Semantic Analysis

- Syntactical correctness  $\nRightarrow$  correct program `void foo = doesntExist / ++"abc";`
- Needs context-sensitive analysis:
  - Variable existence, storage, accessibility, ...
  - Function existence, arguments, ...
  - Operator type compatibility
  - Attribute allowance
- Additional type complexity: inference, polymorphism, ...

#### [Slide 60] Semantic Analysis: Scope Checking with AST Walking

- Idea: walk through AST (in DFS-order) and validate on the way
- Keep track of scope with declared variables
  - Might need to keep track of defined types separately

---

#### In-Class Exercise:

How to implement the scope data structure?

---

<sup>1</sup>TL Veldhuizen. *C++ templates are Turing complete*. 2003. URL: <http://port70.net/~nsz/c/c%2B%2B/turing.pdf>.

<sup>2</sup>J Haberman. *Parsing C++ is literally undecidable*. 2013. URL: <https://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html>.

- For identifiers: check existence and get type
- For expressions: check types and derive result type
- For assignment: check lvalue-ness of left side
- *Might* be possible during AST creation
- Needs care with built-ins and other special constructs

### [Slide 61] Semantic Analysis and Post-Parsing Transformations

- Check for error-prone code patterns
  - Completeness of `switch`, out-of-range constants, unused variables, ...
- Check method calls, parameter types
- Duplicate code for templates
- Make implicit value conversions explicit
- Handle attributes: visibility, warnings, etc.
- Mangle names, split functions (OpenMP), ABI-specific setup, ...
- Last step: generate IR code

## 2.4. Miscellaneous

### [Slide 62] Parsing Performance

Is parsing/front-end performance important?

- Not necessarily: normal compilers
  - Some languages (e.g., Rust) need unbounded time *for parsing*
- Somewhat: JIT compilers
  - Start-up time is generally noticable
- Somewhat more: Developer tools
  - Imagine: waiting for seconds just for updated syntax highlighting
  - Often uses tricks like incremental updates to parse tree

### [Slide 63] Data Types

- Important part of programming languages
- Might have large variety and compatibility
  - Numbers, Strings, Arrays, Compound Types (struct/union), Enum, Templates, Functions, Pointers, ...
  - Class hierarchy, Interfaces, Abstract Classes, ...
  - Integer/float compatibility, promotion, ...
- Might have implicit conversions

**[Slide 64] Data Types: Implementing Classes**

- Simple `class/struct`: trivial, just bunch of fields
  - Methods take (pointer to) `this` as implicit parameter
- Single inheritance: also trivial – extend struct at end
- Virtual methods: store vtable in object representation
  - vtable = table of function pointers for virtual methods
  - Each sub-class has their own vtable
- Multiple inheritance is much more involved
- Dynamic casts: needs run-time type information (RTTI)

**[Slide 65] Recommended Lectures**

AD IN2227 “Compiler Constructions” covers parsing/analysis in depth

AD CIT3230000 “Programming Languages” covers dispatching/mixins/...

**[Slide 66] Compiler Front-end – Summary**

- Lexer splits input into tokens
  - Essentially Regex-Matching + Keywords; rather simple
- Parser constructs (abstract) syntax tree from tokens
  - Top-down vs. bottom-up parsing
  - Typical: top-down for control flow; bottom-up for expressions
  - Respect precedence and associativity for operators
- Semantic analysis ensures meaningful program
- Some data structures are complex to implement
- Some programming languages are more difficult to parse

**[Slide 67] Compiler Front-end – Questions**

- What are typical components of a compiler front-end?
- What output does the lexer produce?
- How does a parser disambiguate rules?
- What is the typical way to handle operator precedence?
- Why are not all programming languages describable using CFGs?
- How to implement classes with virtual functions?



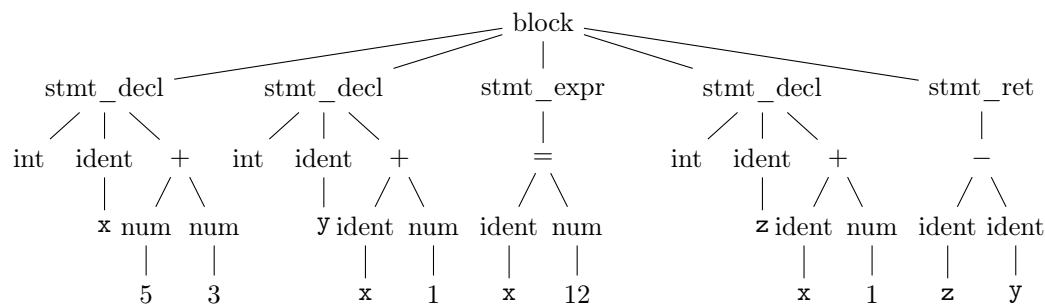
## 3. Intermediate Representations

### 3.1. Motivation

#### [Slide 69] Intermediate Representations: Motivation

- So far: program parsed into AST
- + Great for language-related checks
- + Easy to correlate with original source code (e.g., errors)
- Hard for analyses/optimizations due to high complexity
  - variable names, control flow constructs, etc.
  - Data and control flow implicit
- Highly language-specific

#### [Slide 70] Intermediate Representations: Motivation



Question: how to optimize? Is  $x+1$  redundant?  $\rightsquigarrow$  hard to tell ☹

In this representation, it is very easy to see that the two  $+1$  operations have different operands on the left side and are therefore not trivially redundant.

#### [Slide 71] Intermediate Representations: Motivation

```
x1    ←  5  +  3
y1    ←  x1  +  1
x2    ←  12
z1    ←  x2  +  1
tmp1  ←  z1  -  y1
return tmp1
```

Question: how to optimize? Is `x+1` redundant?  $\rightsquigarrow$  No! ☺

#### [Slide 72] Intermediate Representations

- Definitive program representation inside compiler
  - During compilation, only the (current) IR is considered

In practice, there are, of course, exceptions to the general rule; sometimes an IR contains references to a previous/higher-level IR. An example is LLVM's low-level Machine IR, which only represents single functions and therefore references to global variables use the higher-level LLVM IR.

- Goal: simplify analyses/transformations
  - *Technically*, single-step compilation is possible for, e.g., C ... but optimizations are hard without proper IRs
- Compilers *design* IRs to support frequent operations
  - IR design can vary strongly between compilers
- Typically based on **graphs** or **linear instructions** (or both)

#### [Slide 73] Compiler Design: Effect of Languages – Imperative

- Step-by-step execution of program    modification of state
- Close to hardware execution model
- Direct influence of result
- Tracking of state is complex
- Dynamic typing: more complexity
- Limits optimization possibilities

```
void addvec(int* a, const int* b) {  
    for (unsigned i = 0; i < 4; i++)  
        a[i] += b[i]; // vectorizable?  
}  
func:  
    mov [rdi], rsi  
    mov [rdi+8], rdx  
    mov [rdi], 0 // redundant?  
    ret
```

Tracking state, especially when memory is involved, is one of the main challenges during optimization. In the first example, the loop is not easily vectorizable, because `a` and `b` could point to the same underlying array (e.g., with `addvec(buf + 1, buf)`).



**[Slide 74] Compiler Design: Effect of Languages – Declarative**

- Describes execution target
- Compiler has to derive good mapping to imperative hardware
- Allows for more optimizations
- Mapping to hardware non-trivial
  - Might need more stages
  - Preserve semantic info for opt!
- Programmer has less “control”

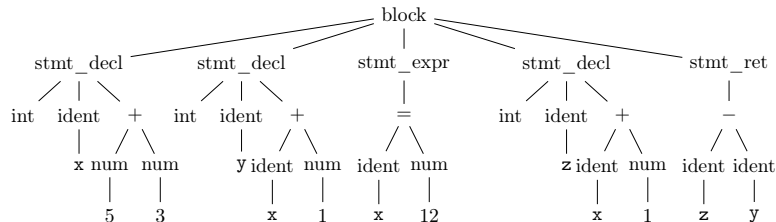
```

select s.name
from studenten s
where exists (select 1
              from hoeren h
              where h.matrno=s.matrno)
let rec fac = function
| 0 | 1 -> 1
| n -> n * fac (n - 1)

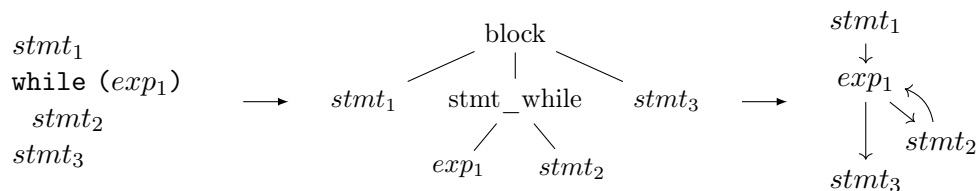
```

**[Slide 75] Graph IRs: Abstract Syntax Tree (AST)**

- Code representation close to the source
- Representation of types, constants, etc. might differ
- Storage might be problematic for large inputs

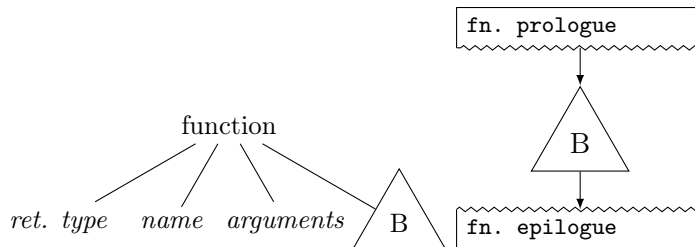
**3.2. Control Flow Graph****[Slide 76] Graph IRs: Control Flow Graph (CFG)**

- Motivation: model control flow between different code sections
- Graph nodes represent **basic blocks**
  - Basic block: sequence of branch-free code (modulo exceptions)
  - Typically represented using a linear IR

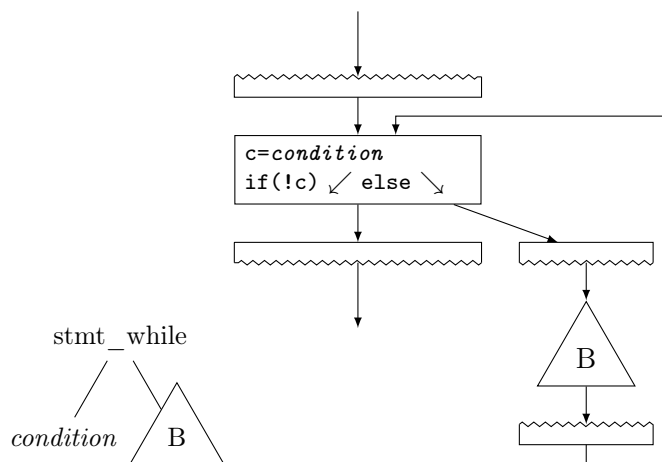


#### [Slide 77] Build CFG from AST – Function

- Idea: Keep track of current insert block while walking through AST



#### [Slide 78] Build CFG from AST – While Loop



Written in pseudo-code:

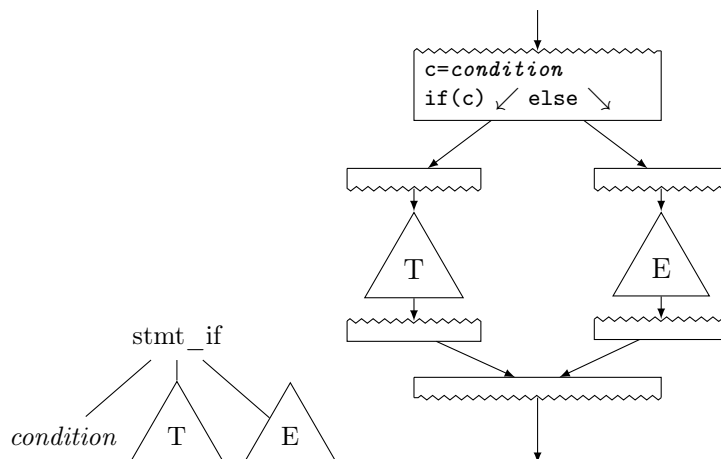
```
IRValue generateCFG(ASTNode* node, BasicBlock*& insPos) {
    switch (node->kind()) {
        case ASTNode::Function:
            insPos = generatePrologue(node);
            generateCFG(node->child(0), insPos);
            generateEpilogue(insPos);
            return nullptr;
        case ASTNode::Block:
            for (ASTNode* child : node->children())
                generateCFG(child, insPos);
            return nullptr;
        case ASTNode::While: {
            BasicBlock* cond = newBlock();
            BasicBlock* body = newBlock();
            BasicBlock* end = newBlock();
            branchTo(insPos, cond);
            insPos = cond;
            IRValue brcond = generateCFG(node->child(0), insPos);
            // NB: generateCFG can modify insPos
            branchToCond(insPos, brcond, body, end);
            insPos = body;
        }
    }
}
```

```

generateCFG(node->child(1), insPos);
branchTo(insPos, cond);
insPos = end;
return nullptr;
}
// ...
}
}

```

### [Slide 79] Build CFG from AST – If Condition



### [Slide 80] Build CFG from AST: Switch

#### Linear search

```

t ← exp
if t == 3: goto B3
if t == 4: goto B4
if t == 7: goto B7
if t == 9: goto B9
goto BD

```

+ Trivial  
– Slow, lot of code

#### Binary search

```

t ← exp
if t == 7: goto B7
elif t > 7:
    if t == 9: goto B9
else:
    if t == 3: goto B3
    if t == 4: goto B4
goto BD

```

+ Good: sparse values  
– Even more code

#### Jump table

```

t ← exp
if 0 ≤ t < 10:
    goto table[t]
goto BD

```

table = {  
    B<sub>D</sub>, B<sub>D</sub>, B<sub>D</sub>, B<sub>3</sub>,  
    B<sub>4</sub>, B<sub>D</sub>, ... }

+ Fastest  
– Table can be large,  
  needs ind. jump

Typically, it is beneficial to keep switches in a high-level form during optimizations, as they are more expressive than, e.g., a corresponding search tree. The corresponding node in the CFG has then one outgoing edge per case plus the edge for the default target.

### [Slide 81] Build CFG from AST: Break, Continue, Goto, Computed Goto

- break/continue: trivial

- Keep track of target block, insert branch
- **goto**: also trivial
  - Split block at target label, if needed
  - But: may lead to irreducible control flow graph (see later)
- Computed **goto**: trivial is bad
  - Split block at target label, if needed
  - Every computed **goto** is a branch to every address-taken label
  - CFG can grow extremely dense for dispatch code!

Very dense CFGs are problematic for the runtime of many graph algorithms — many algorithms need at the very least to consider every edge, so the runtime of such algorithms grows by at least  $\Omega(|N|^2)$ , often more. Typically, therefore, only a single computed **goto** is emitted into the IR as a separate basic block; all computed **gotos** in the code are re-routed to go through this new block. Only very late in the code generation pipeline, this block is duplicated again into all its predecessors to produce the expected behavior. (Note that, due to compiler bugs, this might not always happen.)

#### [Slide 82] CFG: Formal Definition

- **Flow graph**:  $G = (N, E, s)$  with a digraph  $(N, E)$  and entry  $s \in N$ 
  - Each node is a basic block,  $s$  is the entry block
  - $(n_1, n_2) \in E$  iff  $n_2$  might be executed immediately after  $n_1$
  - All  $n \in N$  shall be reachable from  $s$  (unreachable nodes can be discarded)
  - Nodes without successors are end points

#### [Slide 83] CFG from C – Example

---

##### In-Class Exercise:

Derive the CFG for the these functions. Assume a **switch** instruction exists.

```
int fn1() {
    if (a()) {
        while (b()) {
            c();
            if (d())
                continue;
            e();
        }
    } else {
        f();
    }
    return g();
}
```

```
int fn2() {
    a();
    do switch (c()) {
        case 1:
            while (d()) {
                e();
            }
        case 2:
            f();
    }
    default:
        g();
    } while (h());
    return b();
}
```

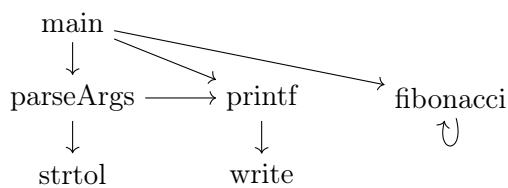
*Solution on page 113.*

---

### 3.3. Other Graph IRs

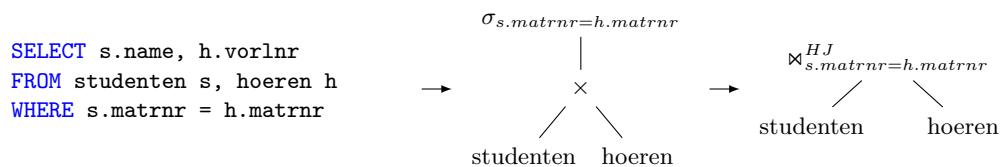
#### [Slide 84] Graph IRs: Call Graph

- Graph showing (possible) call relations between functions
- Useful for interprocedural optimizations
  - Function ordering
  - Stack depth estimation
  - ...



#### [Slide 85] Graph IRs: Relational Algebra

- Higher-level representation of query plans
  - Explicit data flow
- Allow for optimization and selection actual implementations
  - Elimination of common sub-trees
  - Joins: ordering, implementation, etc.



### 3.4. Linear IRs

#### [Slide 86] Linear IRs: Stack Machines

- Operands stored on a stack
- Operations pop arguments from top and push result
- Typically accompanied with variable storage
- Generating IR from AST: trivial
- Often used for bytecode, e.g. Java, Python

- + Compact code, easy to generate and implement
- Performance, hard to analyze

```

push 5
push 3
add
pop x
  
```

```
push x
push 1
add
pop y
push 12
pop x
push x
push 1
add
pop z
```

#### [Slide 87] Linear IRs: Register Machines

- Operands stored in registers
- Operations read and write registers
- Typically: infinite number of registers
- Typically: three-address form
  - $dst = src1 \text{ op } src2$
- Generating IR from AST: trivial
- E.g., GIMPLE, eBPF, Assembly

```
x      ← 5  + 3
y      ← x  + 1
x      ← 12
z      ← x  + 1
tmp1 ← z  - y
return tmp1
```

#### [Slide 88] Example: High GIMPLE

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}

int fac (int n)
gimple_bind < // <-- still has lexical scopes
  int D.1950;
  int res;

  gimple_assign <integer_cst, res, 1, NULL, NULL>
  gimple_goto <<D.1947>>
  gimple_label <<D.1948>>
  gimple_assign <mult_expr, _1, n, n, NULL>
  gimple_assign <mult_expr, res, res, _1, NULL>
  gimple_assign <plus_expr, n, n, -1, NULL>
  gimple_label <<D.1947>>
  gimple_cond <ne_expr, n, 0, <D.1948>, <D.1946>>
```

```
gimple_label <<D.1946>>
gimple_assign <var_decl, D.1950, res, NULL, NULL>
gimple_return <D.1950>
>
$ gcc -fdump-tree-gimple-raw -c foo.c
```

### [Slide 89] Example: Low GIMPLE

```
int foo(int n) {
    int res = 1;
    while (n) {
        res *= n * n;
        n -= 1;
    }
    return res;
}
int fac (int n)
{
    int res;
    int D.1950;

    gimple_assign <integer_cst, res, 1, NULL, NULL>
    gimple_goto <<D.1947>>
    gimple_label <<D.1948>>
    gimple_assign <mult_expr, _1, n, n, NULL>
    gimple_assign <mult_expr, res, res, _1, NULL>
    gimple_assign <plus_expr, n, n, -1, NULL>
    gimple_label <<D.1947>>
    gimple_cond <ne_expr, n, 0, <D.1948>, <D.1946>>
    gimple_label <<D.1946>>
    gimple_assign <var_decl, D.1950, res, NULL, NULL>
    gimple_goto <<D.1951>>
    gimple_label <<D.1951>>
    gimple_return <D.1950>
}
$ gcc -fdump-tree-lower-raw -c foo.c
```

### [Slide 90] Example: Low GIMPLE with CFG

```
int foo(int n) {
    int res = 1;
    while (n) {
        res *= n * n;
        n -= 1;
    }
    return res;
}
int fac (int n) {
    int res;
    int D.1950;
    <bb 2> :
    gimple_assign <integer_cst, res, 1, NULL, NULL>
    goto <bb 4>; [INV]
    <bb 3> :
    gimple_assign <mult_expr, _1, n, n, NULL>
    gimple_assign <mult_expr, res, res, _1, NULL>
    gimple_assign <plus_expr, n, n, -1, NULL>
    <bb 4> :
```

```
gimple_cond <ne_expr, n, 0, NULL, NULL>
  goto <bb 3>; [INV]
else
  goto <bb 5>; [INV]
<bb 5> :
  gimple_assign <var_decl, D.1950, res, NULL, NULL>
  <bb 6> :
gimple_label <<L3>>
  gimple_return <D.1950>
}
$ gcc -fdump-tree-cfg-raw -c foo.c
```

#### [Slide 91] Linear IRs: Register Machines

- Problem: no clear def-use information
  - Is  $x + 1$  the same?
  - Hard to track actual values!
- How to optimize?

⇒ Disallow mutations of variables

```
x      ←  5  +  3
y      ←  x  +  1
x      ←  12
z      ←  x  +  1
tmp1 ←  z  -  y
return tmp1
```

## 3.5. Single Static Assignment

#### [Slide 92] Single Static Assignment: Introduction

- Idea: disallow mutations of variables, value set in declaration
- Instead: create new variable for updated value
- SSA form: every computed value has a unique definition
  - Equivalent formulation: each name describes result of one operation

$x$	$\leftarrow$	5	+	3		$v_1$	$\leftarrow$	5	+	3
$y$	$\leftarrow$	$x$	+	1		$v_2$	$\leftarrow$	$v_1$	+	1
$x$	$\leftarrow$	12			$\longrightarrow$	$v_3$	$\leftarrow$	12		
$z$	$\leftarrow$	$x$	+	1		$v_4$	$\leftarrow$	$v_3$	+	1
$tmp_1$	$\leftarrow$	$z$	-	$y$		$v_5$	$\leftarrow$	$v_4$	-	$v_2$
return		$tmp_1$				return		$v_5$		

#### [Slide 93] Single Static Assignment: Control Flow

- How to handle diverging values in control flow?



- Solution:  $\Phi$ -nodes to merge values depending on predecessor
  - Value depends on edge used to enter the block
  - All  $\Phi$ -nodes of a block execute concurrently (ordering irrelevant)

```
entry: x ← ...
      if (x > 2) goto cont
then:  x ← x * 2
cont:  return x
```

→

```
entry: v1 ← ...
      if (v1 > 2) goto cont
then:  v2 ← v1 * 2
cont:  v3 ←  $\Phi(\text{entry} : v_1, \text{then} : v_2)$ 
      return v3
```

#### [Slide 94] Example: GIMPLE in SSA form

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}

int fac (int n) { int res, D.1950, _1, _6;
  <bb 2> :
  gimple_assign <integer_cst, res_4, 1, NULL, NULL>
  goto <bb 4>; [INV]
  <bb 3> :
  gimple_assign <mult_expr, _1, n_2, n_2, NULL>
  gimple_assign <mult_expr, res_8, res_3, _1, NULL>
  gimple_assign <plus_expr, n_9, n_2, -1, NULL>
  <bb 4> :
  # gimple_phi <n_2, n_5(D)(2), n_9(3)>
  # gimple_phi <res_3, res_4(2), res_8(3)>
  gimple_cond <ne_expr, n_2, 0, NULL, NULL>
  goto <bb 3>; [INV]
  else
  goto <bb 5>; [INV]
  <bb 5> :
  gimple_assign <ssa_name, _6, res_3, NULL, NULL>
  <bb 6> :
  gimple_label <<L3>>
  gimple_return <_6>
}

$ gcc -fdump-tree-ssa-raw -c foo.c
```

### [Slide 95] SSA Construction – Local Value Numbering

- Simple case: inside block – keep mapping of variable to value

Code	SSA IR	Variable Mapping
	$v_1 \leftarrow \text{add } 5, 3$	$x \rightarrow v_3$
$x \leftarrow 5 + 3$	$v_2 \leftarrow \text{add } v_1, 1$	$y \rightarrow v_2$
$y \leftarrow x + 1$	$v_3 \leftarrow \text{const } 12$	$z \rightarrow v_4$
$x \leftarrow 12$	$v_4 \leftarrow \text{add } v_3, 1$	$tmp_1 \rightarrow v_5$
$z \leftarrow x + 1$	$v_5 \leftarrow \text{sub } v_4, v_2$	
$tmp_1 \leftarrow z - y$	$\text{ret } v_5$	
$\text{return } tmp_1$		

### [Slide 96] SSA Construction – Across Blocks

- SSA construction with control flow is non-trivial
- Key problem: find value for variable in predecessor
- Naive approach:  $\Phi$ -nodes for all variables everywhere
  - Create empty  $\Phi$ -nodes for variables, populate variable mapping
  - Fill blocks (as on last slide)
  - Fill  $\Phi$ -nodes with last value of variable in predecessor
- Why is this a bad idea?  $\Rightarrow$  *don't do this!*
  - *Extremely inefficient, code size explosion, many dead  $\Phi$*

### [Slide 97] SSA Construction – Across Blocks (“simple”<sup>1</sup>)

- Key problem: find value in predecessor
- Idea: *seal* block once all direct predecessors are known
  - For acyclic constructs: trivial
  - For loops: seal header once loop block is generated
- Current block not sealed: add  $\Phi$ -node, fill on sealing
- Single predecessor: recursively query that
- Multiple preds.: add  $\Phi$ -node, fill now

Confer the (very readable) paper for a more formal specification of the algorithm.  
The removal of trivial and redundant  $\Phi$ -nodes is not strictly required.

### [Slide 98] SSA Construction – Example

```
int foo(int n) {
    int res = 1;
    while (n) {
        res *= n * n;
    }
}
```

---

<sup>1</sup>M Braun et al. “Simple and efficient construction of static single assignment form”. In: *CC*. 2013, pp. 102–122. URL: [https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9\\_6.pdf](https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9_6.pdf).

```

    n -= 1;
}
return res;
}

func foo(v1)
entry: sealed; varmap: n→v1, res→v2
    v2 ← 1
header: sealed; varmap: n→φ1, res→φ2
    φ1 ← φ(entry: v1, body: v6)
    φ2 ← φ(entry: v2, body: v5)
    v3 ← equal φ1, 0
    br v3, cont, body
body: sealed; varmap: n→v6, res→v5
    v4 ← mul φ1, φ1
    v5 ← mul φ2, v4
    v6 ← sub φ1, 1
    br header
cont: sealed; varmap: res→φ2
    ret φ2

```

Note: the *sealed* state and the variable map are only used during SSA construction. They can be discarded afterwards and are *not* part of the IR. The information is also not sufficient for debug information, for that purpose as it doesn't sufficiently cover all program points; tracking variable states for debug information typically requires extra instructions or annotations in the IR.

### [Slide 99] SSA Construction – Example

#### In-Class Exercise:

Construct an IR in SSA form for the following C functions.

```

int phis(int a, int b){
    a = a * b;
    if (a > b * b) {
        int c = 1;
        while (a > 0)
            a = a - c;
    } else {
        a = b * b;
    }
    return a;
}

int swap(int a, int b, int c) {
    while (c > 0) {
        int tmp = a;
        a = b;
        b = tmp;
        c = c - 1;
    }
    return a;
}

```

*Solution on page 113.*

### [Slide 100] SSA Construction – Pruned/Minimal Form

- Resulting SSA is *pruned* – all  $\phi$  are used
- But not *minimal* –  $\phi$  nodes might have single, unique value
- When filling  $\phi$ , check that multiple real values exist
  - Otherwise: replace  $\phi$  with the single value

- On replacement, update all  $\phi$  using this value, they might be trivial now, too
- Sufficient? Not for irreducible CFG
  - Needs more complex algorithms<sup>2</sup> or different construction method<sup>3</sup>

AD IN2053 “Program Optimization” covers this more formally

#### [Slide 101] SSA: Implementation

- Value is often just a reference to the instruction
- $\phi$  nodes placed at beginning of block
  - They execute “concurrently” and on the edges, after all
- Variable number of operands required for  $\phi$  nodes
- Storage format for instructions and basic blocks
  - Consecutive in memory: hard to modify/traverse
  - Array of pointers:  $\mathcal{O}(n)$  for a single insertion...
  - Linked List: easy to insert, but pointer overhead

Is SSA a graph IR?

Only if instructions have no side effects, consider `load`, `store`, `call`, ...  
These *can* be solved using explicit dependencies as SSA values, e.g. for memory

## 3.6. IR Design

#### [Slide 103] IR Design: High-level Considerations

- Define purpose!
- Structure: SSA vs. something else; control flow
  - Control flow: basic blocks/CFG vs. structured control flow
  - Remember: SSA can be considered as a DAG, too
  - SSA is easy to analyse, but non-trivial to construct/leave
- Broader integration: keep multiple stages in single IR?
  - Example: create IR with high-level operations, then incrementally lower
  - Model machine instructions in same IR?
  - Can avoid costly transformations, but adds complexity

---

<sup>2</sup>M Braun et al. “Simple and efficient construction of static single assignment form”. In: *CC*. 2013, pp. 102–122. URL: [https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9\\_6.pdf](https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9_6.pdf).

<sup>3</sup>R Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *TOPLAS* 13.4 (1991), pp. 451–490. URL: <https://dl.acm.org/doi/pdf/10.1145/115372.115320>.

**[Slide 104] IR Design: Operations**

- Data types
  - Simple type structure vs. complex/aggregate types?
  - Keep relation to high-level types vs. low-level only?
  - Virtual data types, e.g. for flags/memory?
- Instruction format
  - Single vs. multiple results?
  - Strongly typed vs. more generic result/operand types?
  - Operand number – fixed vs. dynamic?

**[Slide 105] IR Design: Operations**

- Allow instruction side effects?
  - E.g.: memory, floating-point arithmetic, implicit control flow
- Operation complexity and abstraction
  - E.g.: `CheckBounds`, `GetStackPtr`, `HashInt128`
  - E.g.: `load` vs. `MOVQconstidx4`
- Extensibility for new operations (e.g., new targets, high-level ops)

**[Slide 106] IR Design: Desired Operations on IR**

- Replacing all uses of an instruction with a different value
- Inserting an instruction or phi node/block argument
- Removing an instruction or phi node/block argument
- Changing the operand of an instruction
- Finding predecessors and successors of a basic block
- Finding all users of an instruction result
- For optimization-focused IRs, all these operations should be fast

**[Slide 107] IR Design: Implementation**

- Maintain user lists?
  - Simplifies optimizations, but adds considerable overhead
  - Replacement can use `copy` and lazy canonicalization
  - User *count* might be sufficient alternative
- Storage layout: operation size and locations
  - For performance: reduce heap allocations, small data structures
- Special handling for arguments vs. all-instructions?
- Metadata for source location, register allocation, etc.
- SSA:  $\phi$  nodes vs. block arguments?

#### [Slide 108] IR Example: Go SSA

- Strongly typed
  - Structured types decomposed
- Explicit memory side-effects
- Also High-level operations
  - `IsInBounds`, `VarDef`
- Only one type of value/instruction
  - `Const64`, `Arg`, `Phi`
- No user list, but user count
- Also used for arch-specific repr.

```
env GOSSAFUNC=fac go build test.go
b1:
  v1 (?) = InitMem <mem>
  v2 (?) = SP <uintptr>
  v5 (?) = LocalAddr <*int> {~r1} v2 v1
  v6 (7) = Arg <int> {n} (n[int])
  v8 (?) = Const64 <int> [1] (res[int])
  v9 (?) = Const64 <int> [2] (i[int])
Plain -> b2 (+9)
b2: <- b1 b4
  v10 (9) = Phi <int> v9 v17 (i[int])
  v23 (12) = Phi <int> v8 v15 (res[int])
  v12 (+9) = Less64 <bool> v10 v6
If v12 -> b4 b5 (likely) (9)
b4: <- b2
  v15 (+10) = Mul64 <int> v23 v10 (res[int])
  v17 (+9) = Add64 <int> v10 v8 (i[int])
Plain -> b2 (9)
b5: <- b2
  v20 (12) = VarDef <mem> {~r1} v1
  v21 (+12) = Store <mem> {int} v5 v23 v20
Ret v21 (+12)
```

#### [Slide 109] Intermediate Representations – Summary

- An IR is an internal representation of a program
- Main goal: simplify analyses and transformations
- IRs typically based on graphs or linear instructions
- Graph IRs: AST, Control Flow Graph, Relational Algebra
- Linear IRs: stack machines, register machines, SSA
- Single Static Assignment makes data flow explicit
- SSA is extremely popular, although non-trivial to construct
- IR design depends on purpose and integration constraints

#### [Slide 110] Intermediate Representations – Questions

- Who designs an IR? What are design criteria?
- Why is an AST not suited for program optimization?
- How to convert an AST to another IR?

- What are the benefits/drawbacks of stack/register machines?
- What benefits does SSA offer over a normal register machine?
- How do  $\phi$ -instructions differ from normal instructions?





## 4. LLVM-IR

### 4.1. Overview

[Slide 112] LLVM<sup>1</sup>

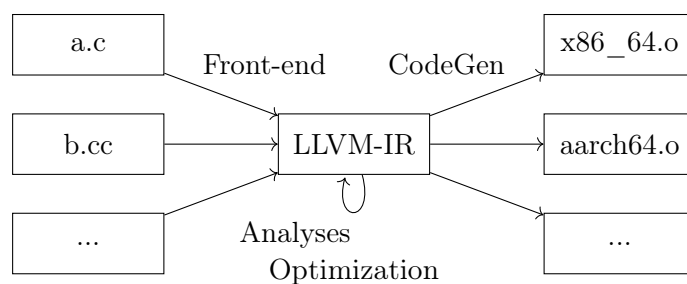
#### LLVM “Core” Library

- Optimizer and compiler back-end
- “Set of compiler components”
  - IRs: LLVM-IR, SelDag, MIR
  - Analyses and Optimizations
  - Code generation back-ends
- Started from Chris Lattner’s master’s thesis
- Used for C, C++, Swift, D, Julia, Rust, Haskell, ...

#### LLVM Project

- Umbrella for several projects related to compilers/toolchain
  - LLVM Core
  - Clang: C/C++ front-end for LLVM
  - libc++, compiler-rt: runtime support
  - LLDB: debugger
  - LLD: linker
  - MLIR: experimental IR framework

[Slide 113] LLVM: Overview



- Independent front-end derives LLVM-IR, LLVM does opt. and code gen.
- LTO: dump LLVM-IR into object file, optimize at link-time

---

<sup>1</sup>C Lattner and V Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *CGO*. 2004, pp. 75–86. URL: <http://www.llvm.org/pubs/2004-01-30-CGO-LLVM.pdf>.

The single IR allows multiple front-ends to reuse the same back-end infrastructure. Thus, generating LLVM-IR provides an easy way to target a wide range of architectures.

For link-time optimization, the LLVM-IR is stored in the object files instead of the machine code. At link-time, a linker plugin detects these files, merges the LLVM-IR from all object files, and then runs the actual compilation as part of the linking step. We will look at LTO again later when discussing object file generation and linking.

## 4.2. LLVM-IR

### [Slide 114] LLVM-IR: Overview

- SSA-based IR, representations textual, bitcode, in-memory
- Hierarchical structure
  - Module
  - Functions, global variables
  - Basic blocks
  - Instructions
- Strongly/strictly typed

```
define dso_local i32 @foo(i32 %0) {  
    %2 = icmp eq i32 %0, 0  
    br i1 %2, label %10, label %3  
  
3: ; preds = %1, %3  
    %4 = phi i32 [ %7, %3 ], [ 1, %1 ]  
    %5 = phi i32 [ %8, %3 ], [ %0, %1 ]  
    %6 = mul nsw i32 %5, %5  
    %7 = mul nsw i32 %6, %4  
    %8 = add nsw i32 %5, -1  
    %9 = icmp eq i32 %8, 0  
    br i1 %9, label %10, label %3  
  
10: ; preds = %3, %1  
    %11 = phi i32 [ 1, %1 ], [ %7, %3 ]  
    ret i32 %11  
}
```

### [Slide 115] LLVM-IR: Data types

- First class types:
  - `i<N>` – arbitrary bit width integer, e.g. `i1`, `i25`, `i1942652`
  - `ptr/ptr addrspace(1)` – pointer with optional address space
  - `float/double/half/bfloat/fp128/...`
  - `<N x ty>` – vector type, e.g. `<4 x i32>`
- Aggregate types:
  - `[N x ty]` – constant-size array type, e.g. `[32 x float]`
  - `{ ty, ... }` – struct (can be packed/opaque), e.g. `{i32, float}`

- Other types:
  - `ty (ty, ...)` – function type, e.g. `{i32, i32} (ptr, ...)`
  - `void`
  - `label/token/metadata`

Although structure types can be used in various places in the IR, e.g., a single instruction to load a large structure from memory, this is strongly discouraged: LLVM is not optimized for this and both code quality and compile times get considerably worse. Only use struct types for globals and to implement multiple return values.

#### [Slide 116] LLVM-IR: Modules

- Top-level entity, one compilation unit – akin to C/C++
- Contains global values, specified with linkage type
- Global variable declarations/definitions

```
@externInt = external global i32, align 4
@globVar = global i32 4, align 4
@staticPtr = internal global ptr null, align 8
```
- Function declarations/definitions

```
declare i32 @readPtr(ptr)
define i32 @return1() {
    ret i32 1
}
```
- Global named metadata (discarded during compilation)

#### [Slide 117] LLVM-IR: Functions

- Functions definitions contain all code, not nestable
- Single return type (or `void`), multiple parameters, list of basic blocks
  - No basic blocks  $\Rightarrow$  function declaration
- Specifiers for `callconv`, section name, other attributes
  - E.g.: `noinline/alwaysinline, noreturn, readonly`
- Parameter and return can also have attributes
  - E.g.: `noalias, nonnull, sret(<ty>)`

#### [Slide 118] LLVM-IR: Basic Block

- Sequence of instructions
  - $\phi$  nodes come first
  - Regular instructions come next
  - Must end with a terminator
- First block in function is entry block Entry block cannot be branch target

##### [Slide 119] LLVM-IR: Instructions – Control Flow and Terminators

- Terminators end a block/modify control flow
- `ret <ty> <val>/ret void`
- `br label <dest>/br i1 <cond>, label <then>, label <else>`
- `switch/indirectbr`
- `unreachable`
- Few others for exception handling
- Not a terminator: `call`

Although `call` does modify control flow in some sense, the assumption is that every function call returns ordinarily. When special control flow for exceptions is needed, the `invoke` instruction is used, which specifies one basic block as successor for the ordinary case and one basic block for the exceptional case.

##### [Slide 120] LLVM-IR: Instructions – Arithmetic-Logical

- `add/sub/mul/udiv/sdiv/urem/srem`
  - Arithmetic uses two's complement
  - Division corner cases are *undefined behavior*
- `fneg/fadd/fsub/fmul/fdiv/frem`
- `shl/lshr/ashr/and/or/xor`
  - Out-of-range shifts have an undefined result
- `icmp <pred>/fcmp <pred>/select <cond>, <then>, <else>`
- `trunc/zext/sext/fptrunc/fpext/fptoui/fptosi/uitofp/sitofp`
- `bitcast`
  - Cast between equi-sized datatypes by reinterpreting bits

Technically, out-of-range shifts return `poison`, see below.

##### [Slide 121] LLVM-IR: Instructions – Memory and Pointer

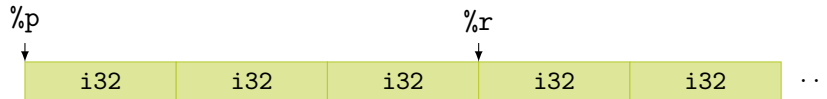
- `alloca <ty>` – allocate addressable stack slot
- `load <ty>, ptr <ptr>/store <ty> <val>, ptr <ptr>`
  - May be `volatile` (e.g., MMIO) and/or `atomic`
- `cmpxchg/atomicrmw` – similar to hardware operations
- `ptrtoint/inttoptr`
- `getelementptr` – address computation on `ptr`/structs/arrays

`alloca` allocates stack memory whenever it is executed. There is a difference between statically-sized `allocas` that are in the entry block versus in dynamically-sized `allocas` or `allocas` in other blocks:

- Static allocas become part of the fixed-size stack frame.
- Dynamic allocas require a dynamically-sized stack frame, which is less efficient (use of a frame pointer is required) and can cause unbounded stack growth.

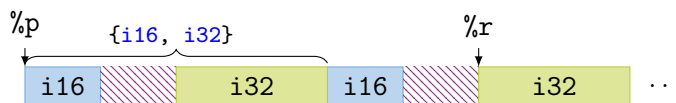
### [Slide 122] LLVM-IR: getelementptr Examples

- `%r = getelementptr i32, ptr %p, i64 3`



Equivalent in C: `&((int*) p)[3]`

- `%r = getelementptr {i16, i32}, ptr %p, i64 1, i32 1`



Equivalent in C: `&((struct {short _0; int _1;}*) p)[1]._1`

- Also works with nested structs and arrays (and vectors)
- Pointer and array can be dynamic, struct indices must be constant

### [Slide 123] LLVM-IR: undef and poison

- **undef** – unspecified value, compiler may choose any value
  - `%b = add i32 %a, i32 undef → i32 undef`
  - `%c = and i32 %a, i32 undef → i32 %a`
  - `%d = xor i32 %b, i32 %b → i32 undef`
  - `br i1 undef, label %p, label %q → undefined behavior`
- **poison** – result of erroneous operations
  - Delay *undefined behavior* on illegal operation until actually relevant
  - Allows to speculatively “execute” instructions in IR
  - `%d = shl i32 %b, i32 34 → i32 poison`

### [Slide 124] LLVM-IR: Intrinsics

- Not all operations provided as instructions
- Intrinsic functions: special functions with defined semantics
  - Replaced during compilation, e.g., with instruction or lib call
- Benefit: no changes needed for parser/bitcode/... on addition
- Examples:
  - `declare iN @llvm.ctpop.iN(iN <src>)`
  - `declare {iN, i1} @llvm.sadd.with.overflow.iN(iN %a, iN %b)`
  - `memcpy, memset, sqrt, returnaddress, ...`

### [Slide 125] LLVM-IR: Tools

- clang can emit LLVM-IR bitcode `clang -O -emit-llvm -c test.c -o test.bc`

- `llvm-dis` disassembles bitcode to textual LLVM-IR `clang -O -emit-llvm -c test.c -o - | llvm-dis`
- `llc` compiles LLVM-IR (textual or bitcode) to assembly `clang -O -emit-llvm -c test.c -o - | llc clang -O -emit-llvm -c test.c -o - | llvm-dis | llc`

Example Listings omitted – they would span several slides

Note: `-emit-llvm` emits the LLVM IR after optimizations. To get the IR before optimizations, use `-Xclang -disable-llvm-passes`. The LLVM IR generated when compiling with `-O0` is different.

### [Slide 126] LLVM-IR: Example

```
define <4 x float> @foo2(<4 x float> %0, <4 x float> %1) {  
    %3 = alloca <4 x float>, align 16  
    %4 = alloca <4 x float>, align 16  
    store <4 x float> %0, ptr %3, align 16  
    store <4 x float> %1, ptr %4, align 16  
    %5 = load <4 x float>, ptr %3, align 16  
    %6 = load <4 x float>, ptr %4, align 16  
    %7 = fadd <4 x float> %5, %6  
    ret <4 x float> %7  
}
```

### [Slide 127] LLVM-IR: Example

```
define i32 @foo3(i32 %0, i32 %1) {  
    %3 = tail call { i32, i1 } @llvm.smul.with.overflow.i32(i32 %0, i32 %1)  
    %4 = extractvalue { i32, i1 } %3, 1  
    %5 = extractvalue { i32, i1 } %3, 0  
    %6 = select i1 %4, i32 -2147483648, i32 %5  
    ret i32 %6  
}
```

### [Slide 128] LLVM-IR: Example

---

#### In-Class Exercise:

```
define i32 @sw(i32 %0) {  
    switch i32 %0, label %4 [  
        i32 4, label %5  
        i32 5, label %2  
        i32 8, label %3  
        i32 100, label %5  
    ]  
2: ; preds = %1  
    br label %5  
3: ; preds = %1  
    br label %5  
4: ; preds = %1  
    br label %5  
5: ; preds = %1, %1, %4, %3, %2
```

```

    %6 = phi i32 [ %0, %4 ], [ 9, %3 ], [ 32, %2 ], [ 12, %1 ], [ 12, %1 ]
    ret i32 %6
}

```

*Solution on page 114.*

## [Slide 129] LLVM-IR: Example

### In-Class Exercise:

```

@a = private unnamed_addr constant [7 x i32] [i32 12, i32 32, i32 12,
                                              i32 12, i32 9, i32 12, i32 12], align 4

define dso_local i32 @f(i32 %0) {
    %2 = add i32 %0, -4
    %3 = icmp ult i32 %2, 7
    br i1 %3, label %4, label %13

4: ; preds = %1
    %5 = trunc i32 %2 to i8
    %6 = lshr i8 %5, %5
    %7 = and i8 %6, 1
    %8 = icmp eq i8 %7, 0
    br i1 %8, label %13, label %9

9: ; preds = %4
    %10 = sext i32 %2 to i64
    %11 = getelementptr inbounds [7 x i32], ptr @a, i64 0, i64 %10
    %12 = load i32, ptr %11, align 4
    br label %13

13: ; preds = %1, %4, %9
    %14 = phi i32 [ %12, %9 ], [ %0, %4 ], [ %0, %1 ]
    ret i32 %14
}

```

*Solution on page 114.*

## 4.3. API

### [Slide 130] LLVM-IR API

- LLVM offers two APIs: C++ and C
  - C++ is the full API, exposing nearly all internals
  - C API is more limited, but more stable
- Nearly all major versions have breaking changes
- Some support for multi-threading:
  - All modules/types/... associated with an `LLVMContext`
  - Different contexts may be used in different threads

### [Slide 131] LLVM-IR C++ API: Basic Example

```

#include <llvm/IR/IRBuilder.h>
int main(void) {
    llvm::LLVMContext ctx;
    auto modUP = std::make_unique<llvm::Module>("mod", ctx);

    llvm::Type* i64 = llvm::Type::getInt64Ty(ctx);
}

```

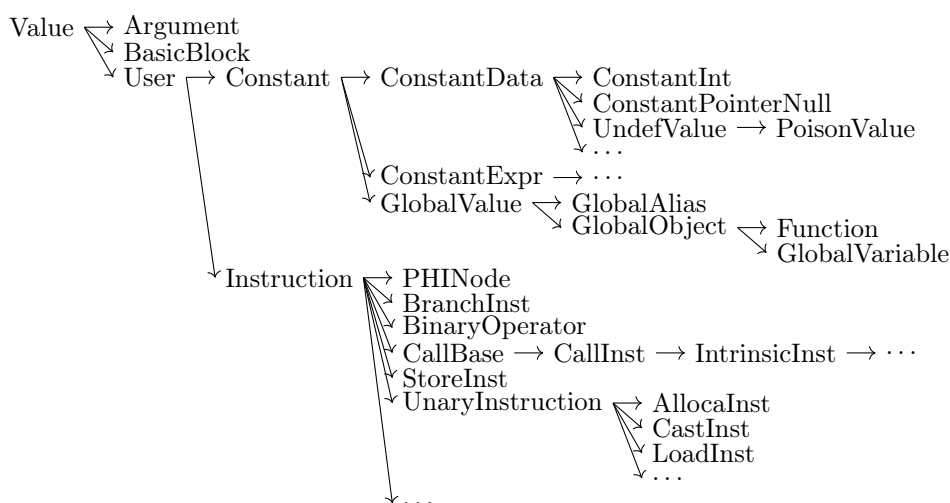
```

llvm::FunctionType* fnTy = llvm::FunctionType::get(i64, {i64}, false);
llvm::Function* fn = llvm::Function::Create(fnTy,
    llvm::GlobalValue::ExternalLinkage, "addOne", modUP.get());
llvm::BasicBlock* entryBB = llvm::BasicBlock::Create(ctx, "entry", fn);

llvm::IRBuilder<> irb(entryBB);
llvm::Value* add = irb.CreateAdd(fn->getArg(0), irb.getInt64(1));
irb.CreateRet(add);
modUP->print(llvm::outs(), nullptr);
return 0;
}

```

### [Slide 132] LLVM-IR API: Almost Everything is a Value... (excerpt)



See LLVM Doxygen<sup>a</sup> for a full graph.

<sup>a</sup>[https://llvm.org/doxygen/classllvm\\_1\\_1Value.html](https://llvm.org/doxygen/classllvm_1_1Value.html)

### [Slide 133] LLVM-IR API: Programming Environment

- LLVM implements custom RTTI
  - `isa<>`, `cast<>`, `dyn_cast<>`
- LLVM implements a multitude of specialized data structures
  - E.g.: `SmallVector<T, N>` to keep  $N$  elements stack-allocated
  - Custom vectors, sets, maps; see manual<sup>2</sup>
- Preferably uses `ArrayRef`, `StringRef`, `Twine` for references
- LLVM implements custom streams instead of std streams
  - `outs()`, `errs()`, `dbgs()`

<sup>2</sup><https://www.llvm.org/docs/ProgrammersManual.html>



Many of these data types are used for efficiency. Standard C++ RTTI is inefficient while LLVM's implementation is very flexible, fast, and has a low memory usage in data structures. The sub-class type of the value is stored in an 8-bit integer.

`SmallVector` is preferred over `std::vector` not just because of the inline storage, but also because (for non-`char` types) it only uses 32-bit integers for length/capacity (lower memory usage, often sufficient) and grows more efficiently for trivially movable data structures (`realloc`).

`Twine` is a lazily evaluated string. For example, when specifying `Twine("foo") + 5`, on-stack data structures are constructed to represent this sequence, but the resulting string is constructed only when and if it is actually used. This also allows constructing strings directly into target buffers.

Standard C++ streams are not just inefficient, implementations also tend to inject global constructors in all files. Therefore, LLVM has its own stream implementation. With `raw_svector_ostream` and `raw_string_ostream`, a `raw_ostream` can be used to write into a `SmallVector` or `std::string`.

#### [Slide 134] LLVM and IR Design

- LLVM provides a decent general-purpose IR for compilers
- But: not ideal for all purposes
  - High-level optimizations difficult, e.g. due to lost semantics
  - Several low-level operations only exposed as intrinsics
  - IR rather complex, high code complexity
  - High compilation times, not very efficient data structures
- Thus: heavy trend towards custom IRs

#### [Slide 135] LLVM-IR – Summary

- LLVM is a modular compiler framework
- Extremely popular and high-quality compiler back-end
- Primarily provides optimizations and a code generator
- Main interface is the SSA-based LLVM-IR
  - Easy to generate, friendly for writing front-ends/optimizations

#### [Slide 136] LLVM-IR – Questions

- What is the structure of an LLVM-IR module/function?
- Which LLVM-IR data types exist? How do they relate to the target architecture?
- How do semantically invalid operations in LLVM-IR behave?
- What is special about intrinsic functions?
- How to derive LLVM-IR from C code using Clang?



## 5. Analyses and Transformations

### 5.1. Motivation

#### [Slide 138] Program Transformation: Motivation

- “User code” is often not very efficient
- Also: no need to, compiler can (often?) optimize better
  - More knowledge: e.g., data layout, constants after inlining, etc.
- Allows for more pragmatic/simple code
- Generating “better” IR code on first attempt is expensive
  - What parts are actually used? How to find out?
- Transformation to “better” code must be done *somewhere*
- Optimization is a misnomer: we don’t know whether it improves code!
  - Many transformations are driven by heuristics
- Many types of optimizations are well-known<sup>1</sup>

### 5.2. Dead Code Elimination

#### [Slide 139] Dead Block Elimination

- CFG not necessarily connected
- E.g., consequence of optimization
  - Conditional branch → unconditional branch
- Removing dead blocks is trivial
  1. DFS traversal of CFG from entry, mark visited blocks
  2. Remove unmarked blocks

#### [Slide 140] Optimization Example 1

```
define i32 @fac(i32 %0) {  
  br label %for.header  
for.header: ; preds = %for.body, %1  
  %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]  
  %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]  
  %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]  
  %cond = icmp sle i32 %i, %0
```

---

<sup>1</sup>FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. URL: <https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf>.

```

    br i1 %cond, label %for.body, label %exit
for.body: ; preds = %for.header
    %a.new = mul i32 %a, %i
    %b.new = add i32 %b, %i
    %i.new = add i32 %i, 1
    br label %for.header
exit: ; preds = %for.header
    %absum = add i32 %a, %b
    ret i32 %a
}

```

**[Slide 141] Simple Dead Code Elimination (DCE)**

- Look for trivially dead instructions
    - No users or side-effects
    - Calls *might* be removed
1. Add all instructions to work queue
  2. While work queue not empty:
    - a) Check for deadness (zero users, no side-effects)
    - b) If dead, remove and add all operands to work queue

**Warning:** Don't implement it this naively, this is inefficient

**[Slide 142] Applying Simple DCE**

```

define i32 @fac(i32 %0) {
eff.: cf    br label %for.header
for.header: ; preds = %for.body, %1
users: 3    %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
users: 2    %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
users: 4    %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
users: 1    %cond = icmp sle i32 %i, %0
eff.: cf    br i1 %cond, label %for.body, label %exit
for.body: ; preds = %for.header
users: 1    %a.new = mul i32 %a, %i
users: 1    %b.new = add i32 %b, %i
users: 1    %i.new = add i32 %i, 1
eff.: cf    br label %for.header
exit: ; preds = %for.header
users: 0    %absum = add i32 %a, %b
eff.: cf    ret i32 %a
}

```

In this example, the instruction `%abssum` can be removed. This reduces the number of users of `%a` and `%b` by 1. As no other instructions have a user count of 0 after this change, the algorithm terminates.

**[Slide 143] Dead Code Elimination**

- Problem: unused value cycles

- Idea: find “value sinks” and mark all needed values as live unmarked values can be removed
    - Sink: instruction with side effects (e.g., store, control flow)
    - Alternative formulation: assume instruction is dead unless proven otherwise
1. Only mark instrs. with side effects as live
  2. Populate work list with newly added live instrs.
  3. While work list not empty:
    - a) Mark dead operand instructions as live and add to work list
  4. Remove instructions not marked as live

**[Slide 144] Applying Liveness-based DCE**

```
define i32 @fac(i32 %0) {  
  live br1 label %for.header  
  for.header: ; preds = %for.body, %1  
  live %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]  
  
  live %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]  
  live %cond = icmp sle i32 %i, %0  
  live br2 i1 %cond, label %for.body, label %exit  
  for.body: ; preds = %for.header  
  live %a.new = mul i32 %a, %i  
  
  live %i.new = add i32 %i, 1  
  live br3 label %for.header  
  exit: ; preds = %for.header  
  
  live ret i32 %a  
}
```

Work list (stack)

This algorithm finds the dead value cycle of %b from the previous example. (Refer to the slide deck for the animated version.)

## 5.3. Sparse Conditional Constant Propagation

**[Slide 145] Optimization Example 2**

```
define i32 @trivial() {  
  entry:  
    br label %for.cond  
  for.cond:  
    %x = phi i32 [ 0, %entry ], [ %inc, %for.inc ]  
    %cmp = icmp ugt i32 %x, 0  
    br i1 %cmp, label %for.inc, label %ret  
  for.inc:  
    %inc = add i32 %x, 1  
    br label %for.cond  
  ret:  
}
```

```
ret i32 %x
}
```

### [Slide 146] Sparse Conditional Constant Propagation<sup>2</sup>

- Constant folding and dead code elimination separately insufficient
- Idea: be optimistic about reachability and constantness
  - Assume block is dead/value is constant unless proven otherwise
  - Ignore edges that are not proven as executable
- For every value: store lattice element
  - Lattice:  $\top$  (any value)  $>$  *constant* (single constant value)  $>$   $\perp$  (dynamic)
  - $\top \wedge x = x, \perp \wedge x = \perp, x \wedge y = (x \text{ if } x = y \text{ else } \perp)$
- For every edge: store executability
  - $\phi$ -nodes: edges that are not (yet) marked executable get  $\top$

### [Slide 147] SCCP Algorithm

State:

- EdgeWorklist
- InstrWorklist
- EdgeExecutable
  - Initialize **false**
- InstrLattice
  - Initialize  $\top$
- EdgeWorklist  $\leftarrow$  edge to entry block
- Get item from any work list; stop when empty
- Edge – if not yet visited, mark visited and:
  - Visited block first time: *visit* all instrs (incl.  $\phi$ )
  - Otherwise: *visit* just  $\phi$ -nodes
- Instr – if block is reachable: *visit*
- At end: replace instrs with const lattice values

### [Slide 148] SCCP Algorithm II

*visit*

- $\phi$ -node: update lattice, meet of visited edge values
- Other instructions: try constant fold
- If lattice value changed:
  - Add all users to instruction worklist
  - Branch with constant/no condition: add selected edge to worklist
  - Branch with  $\perp$ : add all edges to worklist

---

<sup>2</sup>MN Wegman and FK Zadeck. “Constant propagation with conditional branches”. In: *TOPLAS* 13.2 (1991), pp. 181–210. URL: <https://dl.acm.org/doi/pdf/10.1145/103135.103136>.

**[Slide 149]****In-Class Exercise:**

```

define i32 @fn() {
entry:
  br label %loop
loop:
  %j2 = phi i32 [ 1, %entry ], [ %j4, %ifmerge ]
  %k2 = phi i32 [ 0, %entry ], [ %k4, %ifmerge ]
  %kcond = icmp slt i32 %k2, 100
  br i1 %kcond, label %loopbody, label %ret
loopbody:
  %jcond = icmp slt i32 %j2, 20
  br i1 %jcond, label %then, label %else
then:
  %k3 = add i32 %k2, 1
  br label %ifmerge
else:
  %k5 = add i32 %k2, 1
  br label %ifmerge
ifmerge:
  %j4 = phi i32 [ 1, %then ], [ %k2, %else ]
  %k4 = phi i32 [ %k3, %then ], [ %k5, %else ]
  br label %loop
ret:
  ret i32 %j2
}

```

Apply SCCP algorithm and fold constants. Then, remove dead instructions and blocks.

**In-Class Exercise:**

Code attribution: Adapted from `llvm/test/Transforms/SCCP/sccptest.ll` from the LLVM Project, licensed under Apache-2.0 WITH LLVM-Exception.

*Solution on page 114.*

**[Slide 150] SCCP: Misc**

- Strictly more powerful than separate constant folding + DCE
- Runtime:  $\mathcal{O}(\#CFG\text{-edges} + \#instr\text{-operands})$ 
  - Every CFG edge is processed at most once
  - Every def-use relation is processed at most twice lattice can lower at most twice ( $\top \rightarrow const, const \rightarrow \perp$ )
- Can also be used as inter-procedural optimization reaching into module-internal functions

**5.4. Simple Common Subexpression Elimination****[Slide 151] Optimization Example 3**

```

define i32 @foo(i32 %0, ptr %1, ptr %2) {
  %4 = zext i32 %0 to i64

```

```

%5 = getelementptr i32, ptr %1, i64 %4
%6 = load i32, ptr %5, align 4
%7 = zext i32 %0 to i64
%8 = getelementptr i32, ptr %2, i64 %7
%9 = load i32, ptr %8, align 4
%10 = add nsw i32 %6, %9
ret i32 %10
}

```

### [Slide 152] Common Subexpression Elimination (CSE) – Attempt 1

- Idea: find/eliminate redundant computation of same value
- Keep track of previously seen values in hash map
- Iterate over all instructions
  - If found in map, remove and replace references
  - Otherwise add to map
- Easy, right?

This algorithm only works for chains of blocks, but not for merging control flow.  
It is therefore rarely useful, see later.

### [Slide 153] CSE Attempt 1 – Example 1

```

define i32 @foo(i32 %0, ptr %1, ptr %2) {
→ ht      %4 = zext i32 %0 to i64
→ ht      %5 = getelementptr i32, ptr %1, i64 %4
→ ht      %6 = load i32, ptr %5, align 4
dup %4    %7 = zext i32 %0 to i64
→ ht      %8 = getelementptr i32, ptr %2, i64 %7%4
→ ht      %9 = load i32, ptr %8, align 4
→ ht      %10 = add nsw i32 %6, %9
→ ht      ret i32 %10
}

```

- Obsolete instr. can be killed immediately, or in a later DCE

### [Slide 154] CSE Attempt 1 – Example 2

```

define i32 @square(i32 %a, i32 %b) {
entry:
→ ht      %cmp = icmp slt i32 %a, %b
→ ht      br i1 %cmp, label %if.then, label %if.end
if.then: ; preds = %entry
→ ht      %add1 = add i32 %a, %b
→ ht      br label %if.end
if.end: ; preds = %if.then, %entry
→ ht      %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
dup %add1 %add2 = add i32 %a, %b
→ ht      %res = add i32 %condvar, %add2%add1
→ ht      ret i32 %res
}

```



---

Instruction does not dominate all uses!      error: input module is broken!

---

## 5.5. Dominator Tree

### [Slide 155] Domination

- Remember: CFG  $G = (N, E, s)$  with digraph  $(N, E)$  and entry  $s \in N$
- Dominate:  $d \text{ dom } n$  iff every path from  $s$  to  $n$  contains  $d$ 
  - Dominators of  $n$ :  $DOM(n) = \{d \mid d \text{ dom } n\}$
- Strictly dominate:  $d \text{ sdom } n \Leftrightarrow d \text{ dom } n \wedge d \neq n$
- Immediate dominator:  $\text{idom}(n) = d : d \text{ sdom } n \wedge \nexists d'. d \text{ sdom } d' \wedge d' \text{ sdom } n$

$\Rightarrow$  All strict dominators are always executed before the block

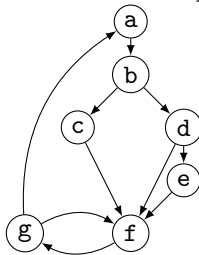
$\Rightarrow$  All values from dominators available/usable

$\Rightarrow$  All values not from dominators **not** usable

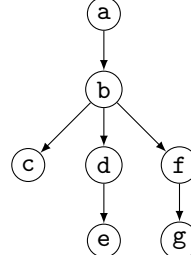
### [Slide 156] Dominator Tree

- Tree of immediate dominators
- Allows to iterate over blocks in pre-order/post-order
- Answer  $a \text{ sdom } b$  quickly

Control Flow Graph



Dominator Tree

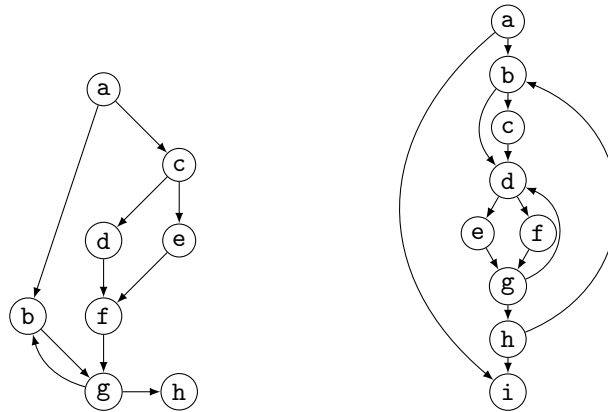


### [Slide 157] Dominator Tree – Example

---

#### In-Class Exercise:

Construct the dominator tree for the following CFGs (entry at **a**):



Solution on page 115.

**[Slide 158] Dominator Tree: Construction**

- Naive: inefficient (but reasonably simple)<sup>3</sup>
  - For each block: find a path from the root – superset of dominators
  - Remove last block on path and check for alternative path
  - If no alternative path exists, last block is idom
- Lengauer–Tarjan: more efficient methods<sup>4</sup>
  - Simple method in  $\mathcal{O}(m \log n)$ ; sophisticated method in  $\mathcal{O}(m \cdot \alpha(m, n))$  ( $\alpha(m, n)$  is the inverse Ackermann function, grows *extremely* slowly)
  - Used in some compilers<sup>5</sup>
- Semi-NCA:  $\mathcal{O}(n^2)$ , but lower constant factors<sup>6</sup>

Most notable, LLVM doesn't use the Lengauer–Tarjan algorithm. Instead, they use the Semi-NCA algorithm, which has  $\mathcal{O}(n^2)$  runtime, but lower constant factors and is therefore substantially faster for certain (typical) inputs<sup>a</sup>.

<sup>a</sup>J Kuderski. “Dominator Trees and incremental updates that transcend times”. In: *LLVM Dev Meeting*. Oct. 2017. URL: [https://llvm.org/devmtg/2017-10/slides/Kuderski-Dominator\\_Trees.pdf](https://llvm.org/devmtg/2017-10/slides/Kuderski-Dominator_Trees.pdf).

**[Slide 159] Dominator Tree: Implementation**

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator: ...lookup *idom*
- Iterate over all dominators/-dominated by: ...trivial

<sup>3</sup>ES Lowry and CW Medlock. “Object code optimization”. In: *CACM* 12.1 (1969), pp. 13–22. URL: <https://dl.acm.org/doi/pdf/10.1145/362835.362838>.

<sup>4</sup>T Lengauer and RE Tarjan. “A fast algorithm for finding dominators in a flowgraph”. In: *TOPLAS* 1.1 (1979), pp. 121–141. URL: <https://dl.acm.org/doi/pdf/10.1145/357062.357071>

<sup>5</sup>Example: <https://github.com/WebKit/WebKit/blob/aabfacb/Source/WTF/wtf/Dominators.h>

<sup>6</sup>L Georgiadis. “Linear-Time Algorithms for Dominators and Related Problems”. PhD thesis. Princeton University, Nov. 2005. URL: <https://www.cs.princeton.edu/techreports/2005/737.pdf>

- Check whether  $a$  sdom  $b$ <sup>7</sup>
  - $a.preNum < b.preNum \wedge a.postNum > b.postNum$
  - After updates, numbers might be invalid: recompute or walk tree
- Problem: dominance of unreachable blocks ill-defined  $\leadsto$  special handling

## 5.6. Common Subexpression Elimination

### [Slide 160] CSE Attempt 2

- Option 1:
  - For identical instructions, store all
  - Add dominance check before replacing
  - Visit nodes in reverse post-order (i.e., topological order)
- Option 2 (Global Value Numbering):<sup>8</sup>
  - Do a DFS over dominator tree
  - Use scoped hashmap to track available values

Does this work? Yes.

### [Slide 161] CSE: Hashing an Instruction (and Beyond)

- Needs hash function *and* “relaxed” equality
- Idea: combine opcode and operands/constants into hash value
  - E.g. assign number to values (hence “value numbering”)
  - Alternatively: use pointer or index for instruction result operands
- Canonicalize commutative operations
  - Order operands deterministically, e.g., by number/address
- Identities:  $a+(b+c)$  vs.  $(a+b)+c$

### [Slide 162] More Complex Global Value Numbering

- Hash-based approach only catches trivially removable duplicates
- Alternative: partition values into *congruence classes*
  - Congruent values are guaranteed to always have the same value
- Optimistic approach: values are congruent unless proven otherwise
- Pessimistic approach: values are not congruent unless proven
- Combinable with: reassociation, DCE, constant folding
- Rather complex, but can be highly beneficial<sup>9</sup>

---

<sup>7</sup>PF Dietz. “Maintaining order in a linked list”. In: *STOC*. 1982, pp. 122–127. URL: <https://dl.acm.org/doi/pdf/10.1145/800070.802184>.

<sup>8</sup>P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997. URL: <https://www.cs.rice.edu/~keith/Promo/CRPC-TR94517.pdf.gz>.

<sup>9</sup>K Gargi. “A sparse algorithm for predicated global value numbering”. In: *PLDI*. 2002, pp. 45–56.

## 5.7. Inlining

### [Slide 163] Inlining

- Inlining: copy function body in place of the call
  - Historically also referred to as “procedure integration”
- Benefits:
  - More optimization opportunities: constant propagation, dead code elimination, better register allocation, etc.
  - Avoids call overhead: stack frame setup, register saving, etc.
- Inlining is crucial for performance in many programming languages
  - “Zero-cost” abstractions typically rely on small functions
  - Critical in e.g. C++/Rust; not too important for C

### [Slide 164] Inlining: Decision

- First: determine whether function is inlinable

Not all functions are inlinable. For example, in LLVM, functions with computed `goto` cannot be inlined, because basic blocks whose address was taken cannot be duplicated. Another example functions compiled from different languages with different exception handling mechanisms (the exception handling routine needs language-specific metadata, which in general cannot be merged; see later).
- Easy way: delegate to programmer (`__attribute__((always_inline))`)

This attribute is, in fact, considered as somewhat problematic, because it is not just a very strong inlining hint, but guarantees inlining even with optimizations disabled. Therefore, even at `-O0`, compilers must execute an optimization pass to specifically handle functions annotated with this attribute.
- Otherwise: very difficult problem
- Problem 1: ordering
  - Inlining big `a()` into `b()` might make `b()` too big to inline but inlining `b()` into `c()` might provide more opportunities
- Problem 2: cost model and thresholds
  - Estimate cost of inlining, e.g. additional code size, more branches
  - Estimate optimization gains and hotness
  - Some attempts to use machine learning here

Typical heuristics are based on somewhat arbitrary thresholds, which are typically higher for higher optimization levels. There’re also approaches to use machine learning

to guide inlining decisions<sup>a</sup>.

<sup>a</sup>M Trofin et al. *MLGO: a Machine Learning Guided Compiler Optimizations Framework*. 2021. arXiv: 2101.04808 [cs.PL]. URL: <https://arxiv.org/abs/2101.04808>.

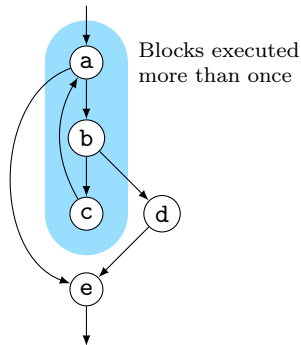
### [Slide 165] Inlining Transformation

- Copy original function in place of the call
  - Split basic block containing function call
  - Needs map of old to new value
  - Constant fold/simplify instructions along the way
  - Replace constant conditional branches  $\rightsquigarrow$  fewer copying
- Replace returns with branches and  $\phi$ -node to/at continuation point
- Move static `alloca` to beginning
- Dynamic `alloca`: save/restore stack pointer
  - Prevent unbounded stack growth in loops
  - LLVM provides `stacksave/stackrestore` intrinsics
- Exceptions may need special treatment

## 5.8. Loop Analysis

### [Slide 166] What is a Loop?

```
void func() {
  while (a()) {
    if (b()) {
      d();
      break;
    }
    c();
  }
  e();
}
```

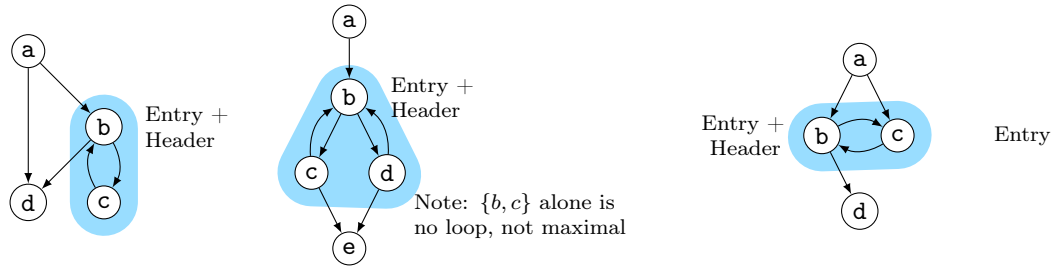


- Loops in source code  $\neq$  loops in CFG
- `d` is *not* part of loop: executed at most once
- $\rightsquigarrow$  Need algorithm to find loops in CFG

### [Slide 167] Loops

- Loop: maximal SCC  $L$  with at least one internal edge<sup>10</sup> (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of  $L$
  - Header  $h$ : first entry found (might be ambiguous)
- Loop nested in  $L$ : loop in subgraph  $L \setminus \{h\}$

<sup>10</sup>P Havlak. “Nesting of reducible and irreducible loops”. In: *TOPLAS* 19.4 (1997), pp. 557–567. URL: <https://dl.acm.org/doi/pdf/10.1145/262004.262005>.



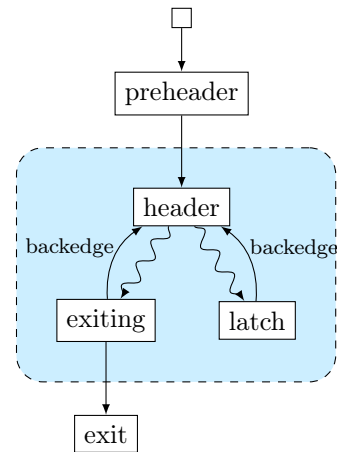
### [Slide 168] Natural Loops

- Natural Loop: loop with single entry
  - $\Rightarrow$  Header is unique
  - $\Rightarrow$  Header dominates all block
  - $\Rightarrow$  Loop is reducible
- Backedge: edge from block to header
- Predecessor: block with edge into loop
- Preheader: unique predecessor

#### Formal Definition

Loop  $L$  is reducible iff  $\exists h \in L . \forall n \in L . h \text{ dom } n$

CFG is reducible iff all loops are reducible



### [Slide 169] Finding Natural Loops

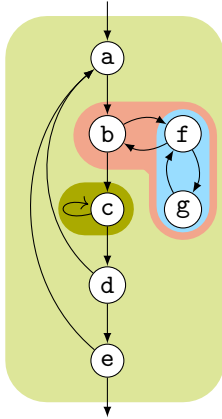
- Modified version<sup>11</sup> of Tarjan's algorithm<sup>12</sup>
- Iterate over dominator tree in post order
- Each block: find predecessors dominated by the block
  - None  $\leadsto$  no loop header, continue
  - Any  $\leadsto$  loop header, these edges *must* be backedges
- Walk through predecessors until reaching header again
  - All blocks on the way must be part of the loop body
  - Might encounter nested loops, update loop parent

<sup>11</sup>G Ramalingam. "Identifying loops in almost linear time". In: *TOPLAS* 21.2 (1999), pp. 175–188. URL: <https://dl.acm.org/doi/pdf/10.1145/316686.316687>.

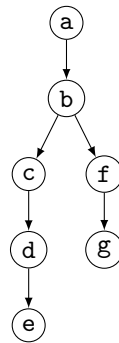
<sup>12</sup>R Tarjan. "Testing flow graph reducibility". In: *STOC*. 1973, pp. 96–107. URL: <https://dl.acm.org/doi/pdf/10.1145/800125.804040>.

**[Slide 170] Finding Natural Loops: Example**

Control Flow Graph



Dominator Tree

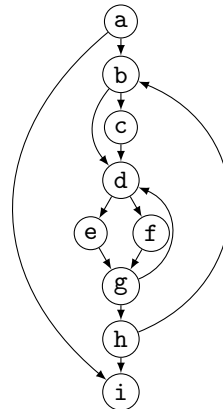
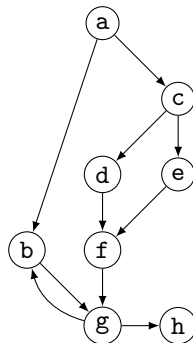


Loop Info

Loop **A**: {c}  
 header: c; parent: D  
 Loop **B**: {f,g}  
 header: f; parent: C  
 Loop **C**: {b,f,g}  
 header: b; parent: D  
 Loop **D**: {a,b,c,d,e,f,g}  
 header: a; parent: NULL

**[Slide 171] Loop Analysis – Example****In-Class Exercise:**

Apply the previous algorithm to find loops in the following CFGs (entry at a):



*Solution on page 115.*

**5.9. Loop Optimizations****[Slide 172] Loop Invariant Code Motion (LICM)**

- Analyze loops, iterate over loop tree in post-order
    - I.e., visit inner loops first
- ↑ Hoist:<sup>13</sup> iterate over blocks of loop in reverse post-order

<sup>13</sup><https://github.com/bytecodealliance/wasmtime/blob/bd6fe11/craneflight/codegen/src/licm.rs>

- For each movable inst., check for loop-defined operands
- If not, move to preheader (create one, if not existent)
- Otherwise, add inst. to set of values defined inside loop

↓ Sink: Iterate over blocks of loop in post-order

- For each movable inst., check for users inside loop
- If none, move to unique exit (if existent)

### [Slide 173] Loop Unswitch

- Hoist loop-invariant conditionals outside loop

After loop-invariant code is moved outside of the loop, it is trivial to determine whether a condition is loop-invariant.

<pre>while (...)   if (loopInvariantCond)     A   else     B   C</pre>	→	<pre>if (loopInvariantCond)   while (...)     A     C   else     while (...)       B       C</pre>
--	---	--

- Might duplicate parts of the loop  $\rightsquigarrow$  needs heuristics
- Might lead to exponential code size increase

For  $n$  independently unswitchable conditions, fully unswitching the loop leads to  $2^n$  loops.

### [Slide 174] Loop Rotation

- Rotate loop condition to bottom
  - Essentially, convert `while{}` to `if{do{}}while{}`

This reduces the number of branches per loop trip from 2 to 1. Additionally, if the loop is known to have a trip count of at least 1, the initial condition can be eliminated.

<pre>define void @src() { ; ... loopheader:   %iv = phi ...   %cond = ...   br i1 %cond, %body, %exit body: ; ...   br %loopheader exit: ; ... }</pre>	→	<pre>define void @tgt() { ; ...   %cond1 = ...   br i1 %cond1, %body, %exit body:   %iv = phi ... ; ...   %cond2 = ...   br i1 %cond2, %body, %exit exit: ; ... }</pre>
--	---	---



**[Slide 175] Loop Strength Reduction**

- Loop induction variables (e.g., loop counters) are often multiplied
  - E.g., for array indexing
- Strength reduction: replace “strong” multiply with “weak” addition
- Needs: analysis of loop induction variables and their recurrence
- Rewrite through replaced/extra loop induction variable
  - E.g., replace scaled index operation with pointer addition
- Information about ISA addressing modes beneficial

**[Slide 176] Analyses and Transformations – Summary**

- Program Transformation critical for performance improvement
- Code not necessarily better
- Analyses are important to drive transformations
  - Dominator tree, loop detection, value liveness
- Important optimizations
  - Dead code elimination, sparse conditional constant propagation, common sub-expression elimination, loop-invariant code motion, loop unswitching, loop rotation, loop strength reduction

**[Slide 177] Analyses and Transformations – Questions**

- Why is “optimization” a misleading name for a transformation?
- How to find unused code sections in a function’s CFG?
- Why is a liveness-based DCE better than a simple, user-based DCE?
- Why is SCCP more powerful than separate DCE/constant prop.?
- What is a dominator tree useful for?
- What is the difference between an irreducible and a natural loop?
- How to find natural loops in a CFG?
- How does the algorithm handle irreducible loops?
- Why is sinking a loop-invariant inst. harder than hoisting?



## 6. Optimizations in LLVM

### 6.1. Overview

#### [Slide 179] Optimizations in LLVM

- Analysis: collect information about IR
  - Examples: loop info, alias analysis, branch probabilities
  - Results are cached, re-run when outdated
- Pass: perform transformation on module/CGSCC/function/loop
  - Examples: instruction combine, simplify CFG, global value numbering, scalar replacement of aggregates (SROA; promotes `alloca` to SSA)
  - Input and output IR must be valid
  - Can use results of analyses; invalidates (some) analyses on changes
- Pass Manager: execute series of passes of same granularity
  - Otherwise, use adaptor: `createFunctionToLoopPassAdaptor`

#### [Slide 180] LLVM Optimization Pipeline (Simplified)<sup>1</sup>

Module Pipeline (`buildPerModuleDefaultPipeline`): two parts

Module Simplification Pipeline (`buildModuleSimplificationPipeline`)

- Early function simplification (SimplifyCFG, SROA, EarlyCSE)
- CGSCC (post-order) (repeated after devirtualization):
  - Inliner
  - Function Simplification Pipeline (`buildFunctionSimplificationPipeline`)
    - \* SROA, SimplifyCFG, InstCombine, Reassociate, GVN, DCE, LICM
    - \* Most passes are here

Inside a call graph strongly-connected component (CGSCC), functions are simplified in post-order (i.e., leaf functions first). The inliner therefore makes decisions based on the simplified callee.

Other acronyms: SROA is Scalable Replacement Of Aggregates; GVN is Global Value Numbering (elimination of common expressions); DCE is Dead Code Elimination; LICM is Loop Invariant Code Motion (hoisting instructions that compute the same value inside the loop out of the loop).

Module Optimization Pipeline (`buildModuleOptimizationPipeline`)

---

<sup>1</sup>See `llvm/lib/Passes/PassBuilderPipelines.cpp` for full details.

- Primarily hard-to-reverse loop optimizations that are not simplifications
- E.g. vectorization, loop unrolling, loop distribution

## 6.2. Canonicalization

### [Slide 181] SSA Construction: Mem2Reg and SROA

- Front-ends typically emit `alloca`s for variables
  - Easier; also simplifies debugging and debug information

Variables stored in `alloca`s have a single location, which greatly simplifies debug information, as the variable does not need to be tracked through registers. Additionally, as the variable is loaded on every use, modifications of the variable in the debugger are possible and in most cases immediately propagated.
- Mem2reg: promote `alloca` to SSA values/`phis`
  - Condition: only `load/store`, no address taken
  - Essentially just SSA construction

In code, this pass is called `PromotePass`.
- SROA: scalar replacement of aggregate
  - Separate structure fields into separate variables
  - Also promote them to SSA, subsumes `mem2reg`

SROA is run multiple times in the pipeline, as further simplifications might enable the promotion of more variables (e.g., when escapes of the variable are eliminated).

### [Slide 182] Simplification

- Many operations can be expressed in various ways
    - E.g.: `sub i32 %x, 1, add i32 %x, -1, add i32 -1, %x`
  - Problem: always need to consider all equivalent cases
- ⇒ Canonicalize IR: one single preferred form
- Single instructions: e.g. operand order
  - Instruction sequences: e.g. series of additions, useless PHIs
  - CFG: e.g. branches, loop layout
- E.g.: constants go to the right, constant `sub` is `add`, loops with fixed trip count turned into `for (i = 0; i != 25; ++i), ...`

Canonicalization is one of LLVM's core approaches to program optimizations. In addition to simplifying the implementation of pattern matches (passes can assume the IR to be canonical form, although they must work correctly even on non-canonical IR), the canonicalizations also simplify and thereby optimize the program.

In contrast to other optimizations, canonicalizations are not based on heuristics and should always work in the same direction (i.e., one pass should not undo the transformation of a previous pass). Inlining is a notable exception of a heuristics-based pass run during the simplification pipeline.

Note that some canonicalization done on the IR are undone in the back-end. Examples include tail duplication (duplicating the function returns) and moving certain instructions back into loops (e.g., to reduce register pressure).

Unfortunately, there is no documentation on the canonical form of IR; it is defined implicitly by what the passes (currently) produce.

### [Slide 183] InstCombine

- Main instruction canonicalization and simplification pass
- Includes many arithmetic-logical patterns, folding of PHIs, ...
- Implementation:
  - Add instructions to work list
    - \* Trivially dead/constant instructions eliminated immediately
    - \* Blocks traversed in reverse post-order
  - Patterns applied to instructions; on match, replace all uses
  - Changed instruction re-enqueues all users to work list

The block ordering in and adding instructions in the order as they appear inside the basic block causes definitions to be visited before their uses (apart from  $\phi$ -nodes). Therefore, adding users to the work list frequently has no effect, as it already contains all users.

- Historically: fixed-point iteration, now just one iter. per run

It turned out that most combines are performed in the first run. Very often, the second iteration would not change the IR but would still need to inspect it, causing easily avoidable compile time costs.

As the pass is executed multiple times throughout the default pass pipeline, combines missed in the first iteration are then simply performed in a later run of the pass.

- Covers many patterns, many patterns missing, >50kLOC C++

### [Slide 184] IR Pattern Matching

- IR patterns are typically DAG matches on def-use graph
- Typically implemented via helper functions<sup>2</sup>

```
// Excerpt from InstCombinerImpl::visitAdd
// (add (xor A, B) (and A, B)) --> (or A, B)
```

---

<sup>2</sup>llvm/IR/PatternMatch.h

```
// (add (and A, B) (xor A, B)) --> (or A, B)
if (match(&I, m_c_BinOp(m_Xor(m_Value(A), m_Value(B)),
                        m_c_And(m_Deferred(A), m_Deferred(B)))))
    return BinaryOperator::CreateOr(A, B);
```

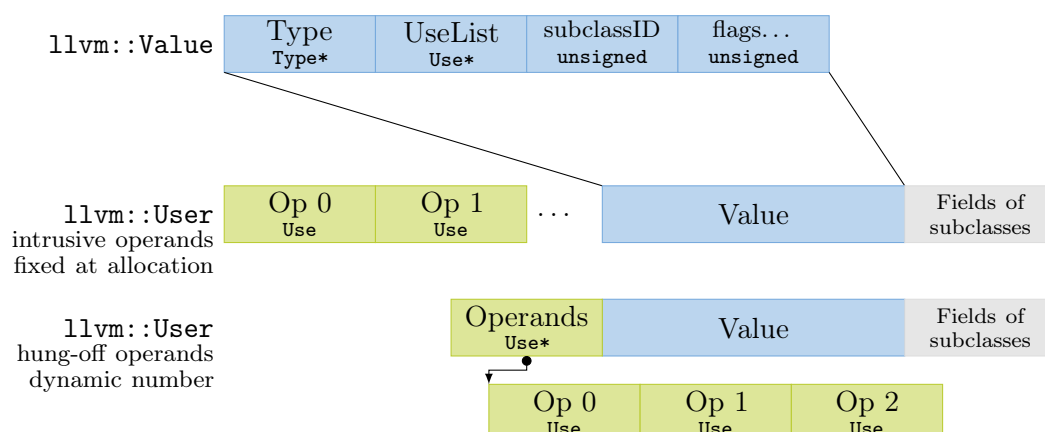
## [Slide 185] LLVM: Use Tracking

- Values track their users
 

```
llvm::Value* v = /* ... */;
for (llvm::User* u : v->users())
    if (auto i = llvm::dyn_cast<llvm::Instruction>(u))
        // ...
```
- Allows for easy replacement:
 

```
inst->replaceAllUsesWith(replVal);
```

## [Slide 186] LLVM IR Implementation: Value/User

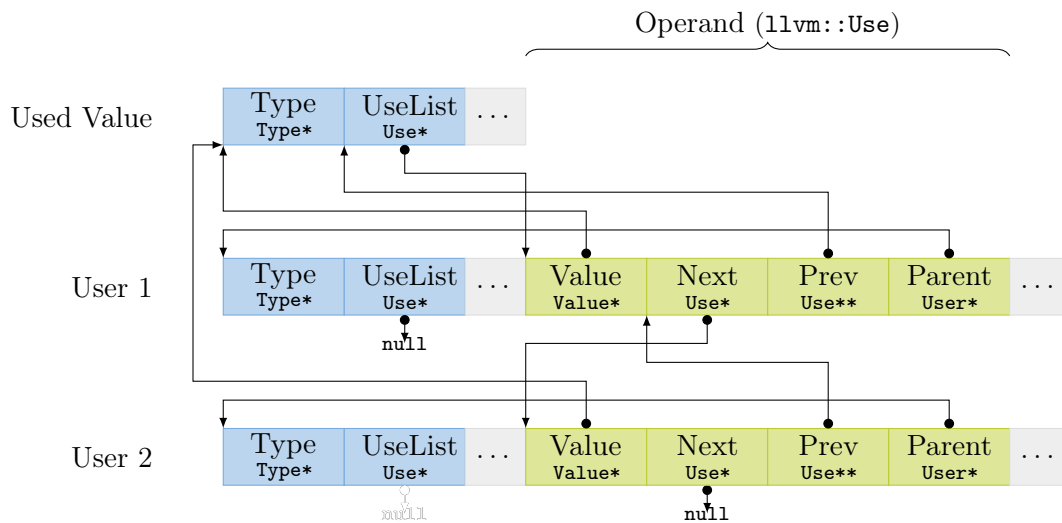


`PHINode` additionally stores  $n$  `BasicBlock*` after the operands, but aren't users of blocks.

Every LLVM `Instruction` is a separate heap allocation. As the number of operands is typically known when constructing the instruction, they are allocated *before* the instruction data structure (this is implemented by `User`).

It can happen that the number of operands increases beyond the allocated storage, for example, when a `PHINode` gets more operands than initially expected. In such cases, the operand list gets hung off into a separate allocation.

As a special case, `PHINode` needs to store the associated `BasicBlocks` in addition to the merged values. The blocks are stored after the operands, but are not operands themselves. Therefore, `PHINode` operands are always hung off.

**[Slide 187] LLVM IR Implementation: Use**

The use list is a doubly-linked list. Starting from `Value::UseList`, one can find all used by following the `Use::Next` pointer. The `Use::Prev` pointer does not point to the previous `Use`, but the previous `Use::Next` pointer or the `Value::UseList` — this way, unlinking does not need to distinguish the special case of the beginning of the use list.

A `Use` also has a pointer to the actual `Value`, so that when inspecting an operand one can actually find the operand itself.

There is also a `Parent` pointer, which points to the `User` which owns the operand: when iterating over the use list, this is the only way to find out which instruction (`User`) uses the value.

In sum, an LLVM-IR operand is quite large, using 32 bytes on a 64-bit system. In addition to every instruction being a separate heap allocation and every operand update requires updating the use list (less data locality), the IR data structures are (in absolute terms) not very efficient — despite being fairly optimized for the use cases they serve.

**[Slide 188] LLVM IR Implementation: Instructions/Blocks**

- `Instruction` and `BasicBlock` have pointers to parent and next/prev
  - Linked list updated on changes and used for iteration
  - Instructions have cached *order* (integer) for fast “comes before”
- `BasicBlock` successors: blocks used by terminator
- `BasicBlock` predecessors:
  - Iterate over users of block – these are terminators (and `blockaddress`)
  - Ignore non-terminators, parent of using terminator is predecessor
  - Same predecessor might be duplicated ( $\rightsquigarrow$  `getUniquePredecessor()`)
- Finding first non- $\phi$  requires iterating over  $\phi$ -nodes

**[Slide 189] Valid Transforms?****In-Class Exercise:**

Which of these transformations  $\text{src} \rightarrow \text{tgt}$  are valid?

```
define float @src1(float %x) {
    %a = fadd float %x, 0.0
    ret float %a
}

define i1 @src2(i8 %x) {
    %xm1 = add i8 %x, -1
    %y = xor i8 %x, %xm1
    %r = icmp ule i8 %x, %y
    ret i1 %r
}

define i8 @src3(i8 %x) {
    %div = sdiv i8 %x, 4
    ret i8 %div
}

define float @tgt1(float %x) {
    ret float %x
}

define i1 @tgt2(i8 %x) {
    %ctpop = call i8 @llvm.ctpop(i8 %x)
    %r = icmp ult i8 %ctpop, 2
    ret i1 %r
}

define i8 @tgt3(i8 %x) {
    %div = ashr i8 %x, 2
    ret i8 %div
}
```

*Solution on page 115.*

---

**[Slide 190] Alive2**

- Seemingly correct transformations might be wrong in edge cases
  - Unexpected transformations can be correct
- ⇒ Prove correctness of transformations
- Alive2<sup>3</sup>: translation validation tool for LLVM-IR
  - Convert LLVM-IR into SMT formulas
  - Proves that target is a refinement of source function

Alive2 transforms LLVM-IR functions, including memory accesses and control flow, into SMT formulas and uses the Z3 SMT solver to prove that the target function is indeed a refinement of the source function, i.e., that the target is more defined than the source. There is limited support for **undef** values (which are not tracked per-bit but just per-value) and loops (which are completely unrolled for a fixed number of iterations).

- InstCombine transformations require Alive2 proofs nowadays<sup>4</sup>

**[Slide 191] Value Tracking**

- Many transformations benefit from value ranges and known bits
  - E.g., fold `trunc+shift` into smaller shift if shift amount is known small
- `computeKnownBits`: recursively collect information from instrs
  - Arithmetic-logical instrs, casts, several intrinsics, etc.

---

<sup>3</sup>NP Lopes et al. “Alive2: bounded translation validation for LLVM”. in: *PLDI*. 2021, pp. 65–79. URL: [users.cs.utah.edu/~regehr/alive2-pldi21.pdf](https://users.cs.utah.edu/~regehr/alive2-pldi21.pdf).

<sup>4</sup>Random Example: [https://alive2.llvm.org/ce/z/xe\\_vb2](https://alive2.llvm.org/ce/z/xe_vb2)



- Bitwise instructions easier to reason about  $\rightsquigarrow$  preferred canonicalization
  - E.g.  $(1 \ll X) - 1 \rightarrow (-1 \ll X) \wedge -1$
  - E.g.  $\text{sext}(X) \rightarrow \text{zext}(X)$  if  $X$  is known non-negative
- Also use knowledge from front-end (but how encoded?)

## 6.3. Flags

### [Slide 192] Attribute, Flags, Assume

- Parameter/return attributes: constrain value ranges
  - `range(from, to)`, `nonnull`, `noundef`, ...
  - Values outside of the range are `poison`
- Instruction flags: encode additional knowledge about operation
  - `add/sub/mul/trunc nsw/nuw`: no signed/unsigned wrap
    - ▮ This can be used, e.g., to restrict the value range or get more precise information about loop trip counts, memory access offsets.
  - `or disjoint`: combination of disjoint bits
    - ▮ This permits back-ends to replace the `or` with an `add`, which can be folded into other operations (e.g., `leaq` on x86).
  - `udiv/sdiv/lshr/ashr exact`: divisor multiple of dividend
    - ▮ Permit replacing division with shift and implies that the bits that are shifted out are zero.
  - `getelementptr nuw/nusw/inbounds`: no wrap/stays inside allocation
    - ▮ Bounds information primarily improves alias analysis: if two base pointers are known to refer to different allocations, they will never be the same after two `inbounds` computations.
  - `zext/uitofp nneg`: operand is non-negative
    - ▮ When a value is known to be non-negative, the instruction is converted into the canonical unsigned form. However, some targets might want to undo this transformation in the back-end, e.g., because sign-extension is cheaper (RISC-V) or the floating-point conversion is cheaper for signed integers (x86), but it might be difficult to re-prove the non-negativeness after other transformations. This flag retains this information and permits relaxing the operations later on.
  - `icmp samesign`: operands have the same sign

When both operands are known to have the same sign, `icmp` is canonicalized into the unsigned comparison predicate. However, when there are multiple comparisons of the same value with some being signed while others are unsigned, it is more difficult to aggregate information. This flag retains this information and indicates that the signed and unsigned variants of the comparison predicate can be used interchangeably.

- Violation is `poison`

### [Slide 193] Assume

- Intrinsic `llvm.assume(i1)`
- Conditional immediate undefined behavior if argument is `false`
- Requires extra instruction for condition  $\rightsquigarrow$  might be problematic
  - Instructions use other values, etc.  $\rightsquigarrow$  prevent optimizations
- Also supports operand bundles for `nonnull`, `align`, `separate_storage`

Here's an example which demonstrate the use of `separate_storage` assumptions:

```
struct Buffer {  
    char *data;  
    void write(char c) {  
        *data++ = c;  
    }  
};  
void writeLE(unsigned x, Buffer &buf) {  
    buf.write(x);  
    buf.write(x >> 8);  
    buf.write(x >> 16);  
    buf.write(x >> 24);  
}
```

The resulting machine code will repeatedly load the `data` pointer from `buf`, because `data` might alias with `buf`, because in C/C++ `char*` can alias with everything. When exposing the information that inside `write()` the pointers `this` and `this->data` do not alias, the four stores can get combined.

```
// Inside write:  
__builtin_assume_separate_storage(this, data);  
// LLVM IR: call void @llvm.assume(i1 true) [ "separate_storage"(ptr %this, ptr %data) ]
```

Full example: <https://godbolt.org/z/TMGrTco9T>

### [Slide 194] Valid Transforms?

---

#### In-Class Exercise:

Which of these transformations `src`→`tgt` are valid?

```

define i8 @src1(i8 %x) {
  %div = udiv exact i8 1, %x
  ret i8 %div
}
define i1 @src2(i8 %x, i8 %y) {
  %cmp = icmp sge i8 %x, %y
  %cmpeq = icmp samesign eq i8 %x, 127
  %r = select i1 %cmp, i1 %cmpeq, i1 false
  ret i1 %r
}
define i8 @src3(i8 %x, i8 %y, i8 %z) {
  %xy = add nsw i8 %x, %y
  %xyz = add nsw i8 %xy, %z
  ret i8 %xyz
}
define i8 @tgt1(i8 %x) {
  ret i8 1
}
define i1 @tgt2(i8 %x, i8 %y) {
  %cmpeq = icmp samesign eq i8 %x, 127
  ret i1 %cmpeq
}
define i8 @tgt3(i8 %x, i8 %y, i8 %z) {
  %xz = add nsw i8 %x, %z
  %xyz = add nsw i8 %xz, %y
  ret i8 %xyz
}

```

*Solution on page 115.*

### [Slide 195] Fast-Math Flags

- Floating-point arithmetic typically follows IEEE 754
  - Except for NaN, which is more relaxed
- The behavior of floating-point instructions w.r.t. NaN varies strongly between different targets (e.g., taking one of the operands, returning a canonical NaN, converting signalling NaN to quiet NaN). Therefore, the semantics are specified rather generic for LLVM-IR.
- Implication: almost impossible to optimize
  - non-associative, NaNs, infinity, negative zero, no reciprocals, no contraction
  - Transformations can result in vastly different results!
- Additionally: floating-point exceptions
  - Typically ignored, but some users might be interested
- Value/operation flags to permit transformations locally
- `strictfp` function attribute allows more control through constrained floating-point intrinsics

### [Slide 196] Branch Weights

- Annotation of branch with expected probabilities
  - Sources: programmer annotations, execution profiles
- Added as metadata to branch instructions
- Can compute block execution frequencies

```

define i32 @f(i32 %x) {
; ...
  %cmp = icmp eq i32 %x, 0
  br i1 %cmp, label %then, label %else, !prof !5
; ...
}

!5 = ![! "branch_weights", ! "expected", i32 1, i32 2000]

```

## 6.4. Inter-Procedural Optimizations

### [Slide 197] Function Argument Optimizations

- Inferring argument attributes
- Eliminate dead arguments
  - Optimistic liveness analysis of arguments (assume dead)
  - Create new function with fewer arguments
- Promote pointer arguments to value arguments
  - Esp. relevant for C++ pass-by-reference
  - Only possible when both caller/callee CPU features match

### [Slide 198] Other Inter-Procedural Optimizations

- Inlining / Outlining
- Function Specialization
- Optimization of global variables
  - Marking as constant, delete dead variables, attributes, etc.
- Inter-procedural SCCP
- Devirtualization / call-target speculation
- Hot-Cold Splitting
- ...

### [Slide 199] Optimization Remarks

- Understanding optimization decisions can be difficult
  - Complex code base, abstraction, heuristics, ...
- Optimization remarks: passes can report information
  - Inlining decisions: provide cost values/thresholds for decisions
  - Others: GVN, LICM, FastISel
- Output also provided as YAML or a binary format for analysis
- Lots of verbose output, hard to identify important points
- Clang: `-Rpass=<pass> -Rpass-analysis=<pass> -Rpass-missed=<pass>`

### [Slide 200] Other Aspects not Covered Here

- Analyses: alias analysis, MemorySSA, Scalar Evolution (SCEV), target-specific analyses/heuristics, ...
- Optimizations: various loop transformations (distribute, fuse, flatten, polyhedral optimizations), value propagation, vectorization, reassociation, ...
- Loop-Closed SSA canonical form
  - Values defined inside loop only used in loop or in  $\phi$ -node in exit
  - Simplifies tracking uses of values defined in loops

## 6.5. Running and Writing LLVM Passes

### [Slide 201] Using LLVM (New) Pass Manager

```
void optimize(llvm::Function* fn) {
    llvm::PassBuilder pb;
    llvm::LoopAnalysisManager lam{};
    llvm::FunctionAnalysisManager fam{};
    llvm::CGSCCAnalysisManager cgam{};
    llvm::ModuleAnalysisManager mam{};
    pb.registerModuleAnalyses(mam);
    pb.registerCGSCCAnalyses(cgam);
    pb.registerFunctionAnalyses(fam);
    pb.registerLoopAnalyses(lam);
    pb.crossRegisterProxies(lam, fam, cgam, mam);

    llvm::FunctionPassManager fpm{};
    fpm.addPass(llvm::DCEPass());
    fpm.addPass(llvm::createFunctionToLoopPassAdaptor(llvm::LoopRotatePass()));
    fpm.run(*fn, fam);
}
```

### [Slide 202] Writing a Pass for LLVM's New PM – Part 1

```
#include "llvm/IR/PassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"

class TestPass : public llvm::PassInfoMixin<TestPass> {
public:
    llvm::PreservedAnalyses run(llvm::Function &F,
                               llvm::FunctionAnalysisManager &AM) {
        // Do some magic
        llvm::DominatorTree *DT = &AM.getResult<llvm::DominatorTreeAnalysis>(F);
        // ...
        llvm::errs() << F.getName() << "\n";
        return llvm::PreservedAnalyses::all();
    }
};
// ...
```

### [Slide 203] Writing a Pass for LLVM's New PM – Part 2

```
extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
    return { LLVM_PLUGIN_API_VERSION, "TestPass", "v1",
        [] (llvm::PassBuilder &PB) {
            PB.registerPipelineParsingCallback(
                [] (llvm::StringRef Name, llvm::FunctionPassManager &FPM,
                   llvm::ArrayRef<llvm::PassBuilder::PipelineElement>) {
                    if (Name == "testpass") {
                        FPM.addPass(TestPass());
                        return true;
                    }
                    return false;
                }
            );
        }
    };
}
```

```
    });  
  } };  
}  
  
c++ -shared -o testpass.so testpass.cc -llvm -fPIC  
opt -S -load-pass-plugin=$PWD/testpass.so -passes=testpass input.ll
```

### [Slide 204] Optimizations in LLVM – Summary

- LLVM provides a wide range of analyses and optimizations
- Optimizations organized in flexibly composeable passes
- Simplification centered around canonicalization
- Much knowledge inferred from assumptions (and `poison`/UB)
- SSA simplifies many transforms through explicit def–use chains
- Only few non-simplifying transformation passes

### [Slide 205] Optimizations in LLVM – Questions

- Why are some passes executed multiple times during optimization?
- How does the simplification pipeline relate to inlining?
- What is the key benefit of canonicalizing the IR?
- How does LLVM's `replaceAllUsesWith` work?
- What is the benefit of instr. flags like `nsw`?
- Why are floating-point optimizations problematic?
- How is C++ `[[likely]]` represented in LLVM-IR?

## 7. Vectorization

### [Slide 207] Parallel Data Processing

- Sequential execution has inherently limited performance
  - Clock rate, energy consumption/cooling, data path lengths, speed of light, ...

The speed of light is around  $3 \cdot 10^8 \frac{m}{s}$ . For a 4 GHz processor, during one clock cycle light and therefore also the electricity charges in the circuit can travel *at most* 75 cm. This doesn't consider the latencies of transistors, resistance, and other physical properties, which further restrict the path length.

- Parallelism is the key to substantial and scalable perf. improvements
- Modern systems have many levels of parallelism:
  - Multiple nodes/systems, connected via network
  - Different compute units (CPU, GPU, etc.), connected via PCIe
  - Multiple CPU sockets, connected via QPI (Intel) or HyperTransport (AMD)
  - Multiple CPU cores
  - Multiple threads per core
  - Instruction-level parallelism (superscalar out-of-order execution)
  - Data parallelism (SIMD)

### [Slide 208] Single Instruction, Multiple Data (SIMD)

- Idea: perform same operations on multiple data in parallel
- First computer with SIMD operations: MIT Lincoln Labs TX-2, 1957<sup>1</sup>

The MIT TX-2 supported computations on one 36-bit integer, two 17-bit integers, four 9-bit integers, and the pair of one 27-bit and one 9-bit integer.

- Wider use in HPC in 1970s with vector processors (Cray et al.)
  - Ultimately replaced by much more scalable distributed machines
- SIMD-extensions for multimedia processing from 1990s onwards
  - Often include very special instructions for image/video/audio processing
- Shift towards HPC and data processing around 2010
- Extensions for machine learning/AI in late 2010s

---

<sup>1</sup>W Clark et al. *The Lincoln TX-2 Computer*. Apr. 1957. URL: [http://www.bitsavers.org/pdf/mit/tx-2/TX-2\\_Papers\\_WJCC\\_57.pdf](http://www.bitsavers.org/pdf/mit/tx-2/TX-2_Papers_WJCC_57.pdf).

**[Slide 209] SIMD: Idea**

- Multiple data elements are stored in *vectors*
  - Size of data may differ, vector size is typically constant
  - Single elements in vector referred to as *lane*
- (Vertical) Operations apply the same operation to all lanes

	lane 3	lane 2	lane 1	lane 0
src 1	1	2	3	4
	+	+	+	+
src 2	1	2	3	4
	↓	↓	↓	↓
result	2	4	6	8

- Horizontal operations work on neighbored elements

**[Slide 210] SIMD ISAs: Design**

- Vectors are often implemented as fixed-size wide registers
  - Examples: ARM NEON 32×128-bit, Power QPX 32×256-bit
  - Data types and element count is defined by instruction
- Some ISAs have dynamic vector sizes: ARM VFP, ARM SVE, RISC-V V
  - Problematic for compilers: variable spill size, less constant folding
- Data types vary, e.g. i8/i16/i32/i64/f16/bf16/f32/f64/f128
  - Sometimes only conversion, sometime with saturating arithmetic
- Masking allows to suppress operations for certain lanes
  - Dedicated mask registers (AVX-512, SVE, RVV) allow for hardware masking
  - Can also apply for memory operations, optionally suppressing faults
  - Otherwise: software masking with another vector register

**[Slide 211] Historical Development of SIMD Extensions**

See Figure 7.1.

**[Slide 212] SIMD: Use Cases**

- Dense linear algebra: vector/matrix operations
  - Implementations: Intel MKL, OpenBLAS, ATLAS, ...
- Sparse linear algebra
  - Needs gather/scatter instructions
- Image and video processing, manipulation, encoding
- String operations
  - Implemented, e.g., in glibc, simdjson



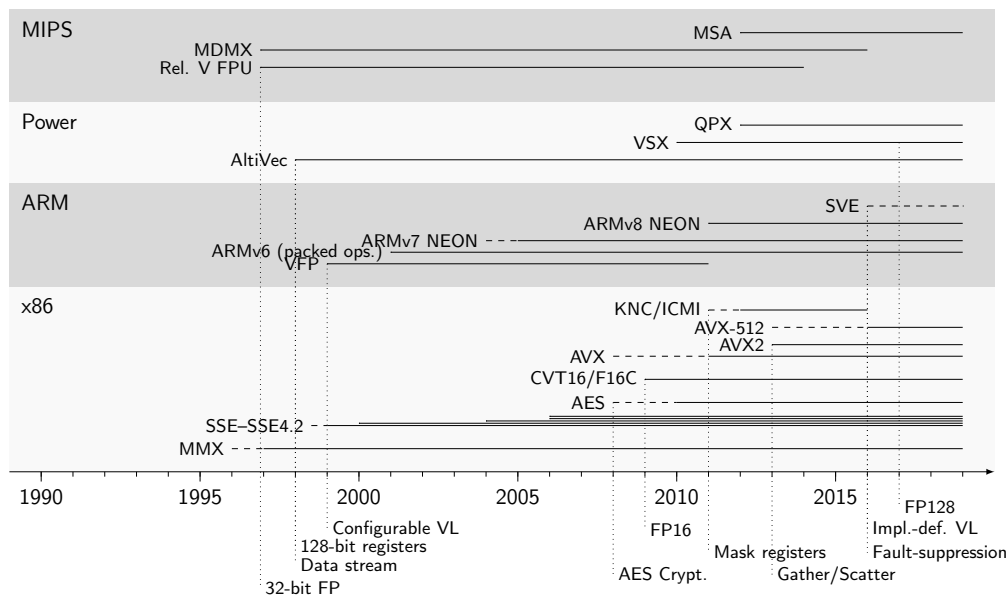


Figure 7.1.: Historical Development of SIMD extensions to contemporary ISAs.

- Cryptography

#### [Slide 213] SIMD ISAs: Usage Considerations

- Very easy to implement in hardware
  - Simple replication of functional units and larger vector registers
  - Too large vectors, however, also cause problems (AVX-512)

Large vectors (e.g., in AVX-512) cause a high space and power usage in hardware. They also need a higher memory bandwidth – on x86 systems, loading one AVX-512 register means copying an entire cache line. The performance gains of doubling the vector size are smaller than 2x for many applications, and some CPUs cannot sustain executing 512-bit operations at the maximum clock frequency due to power consumption and cooling.

- Offer significant speedups for certain applications
  - With 4x parallelism, speed-ups of  $\sim 3x$  are achievable
  - Amdahl's Law applies, unfortunately
- Caveat: non-trivial to program
  - Optimized routines provided by libraries
  - Compilers try to auto-vectorize, but often need guidance

#### [Slide 214] SIMD Programming: (Inline) Assembly

- Idea: SIMD is too complicated, let programmer handle this

- Programmer specifies exact code (instrs, control flow, and registers)
- Inline assembly allows for integration into existing code
  - Specification of register constraints and clobbers needed
- “Popular” for optimized libraries
- + Allows for best performance
- Very tedious to write, manual register allocation, non-portable
- No optimization across boundaries

### [Slide 215] SIMD Programming: Intrinsics

- Idea: deriving a SIMD schema is complicated, delegate to programmer
- Intrinsic functions correspond to hardware instructions
  - `__m128i _mm_add_epi32 (__m128i a, __m128i b)`
- Programmer explicitly specifies vector data processing instructions compiler supplements registers, control flow, and scalar processing
- + Allows for very good performance, still exposes all operations
- ~ Compiler can to some degree optimize intrinsics
  - GCC does not; Clang/LLVM does – intrinsics often lowered to LLVM-IR vectors (which also has some problems)
- Tedious to write, non-portable

### [Slide 216] SIMD Programming: Intrinsics – Example

```
float sdot(size_t n, const float x[n], const float y[n]) {
    size_t i = 0;
    __m128 sum = _mm_set_ps1(0);
    for (i = 0; i < (n & ~3ul); i += 4) {
        __m128 x1 = _mm_loadu_ps(&x[i]);
        __m128 y1 = _mm_loadu_ps(&y[i]);
        sum = _mm_add_ps(sum, _mm_mul_ps(x1, y1));
    }
    // ... take care of tail (i..<n) ...
}
```

### [Slide 217] Intrinsics for Unknown Vector Size

- Size not known at compile-time, but can be queried at runtime
  - SVE: instruction `incd` adds number of vector lanes to register
- In C: behave like an incomplete type, except for parameters/returns
- Flexible code often slower than with assumed constant vector size
- Consequences:
  - Cannot put such types in structures, arrays, `sizeof`
  - Stack spilling implies variably-sized stack
- Instructions to set mask depending on bounds: `whilelt`, ...
  - No loop peeling for tail required

---

## [Slide 218] SIMD Programming: Target-independent Vector Extensions

- Idea: vectorization still complicated, but compiler can choose instrs.
  - Programmer still specifies exact operations, but in target-independent way
  - Often mixable with target-specific intrinsics
- Compiler maps operations to actual target instructions
- If no matching target instruction exists, use replacement code
  - Inherent danger: might be less efficient than scalar code
- Often relies on explicit vector size

## [Slide 219] GCC Vector Extensions

---

### In-Class Exercise:

Compile<sup>a</sup> the following operations and observe how the output changes:

- Add 16-byte vectors of element type `uint32_t`
- Multiply 8-byte vectors of element type `uint32_t/uint8_t`
- Divide 64-byte vectors of element type `uint32_t/long double`

```
// compile with: clang -O3 -S --target=x86_64 file.c -o -  
// also try --target=aarch64  
#include <stdint.h>  
typedef uint32_t vecty __attribute__((vector_size(16)));  
vecty op(vecty a, vecty b) {  
    return a + b;  
}
```

*Solution on page 115.*

---

<sup>a</sup><https://godbolt.org/z/43v98rdno>

---

## [Slide 220] LLVM-IR: Vectors

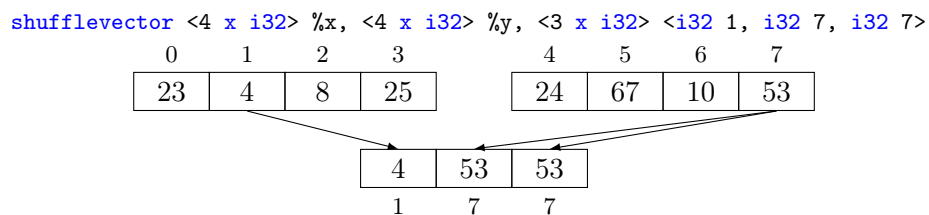
- `<N x ty>` – fixed-size vector type, e.g. `<4 x i32>`
  - Valid element type: integer, floating-point, pointers
  - Memory layout: densely packed (i.e., `<8 x i2> ≈ i16`)
- `<vscale x N x ty>` – scalable vector, e.g. `<vscale x 4 x i32>`
  - Vector with a multiple of N elements
  - Intrinsic `@llvm.vscale.i32()` – get runtime value of `vscale`
- Most arithmetic operations can also operate on vectors
- `insertelement/extractelement`: modify single element
  - Example: `%4 = insertelement <4 x float> %0, float %1, i32 %2`
  - Index can be non-constant value

Note, however, that few ISAs actually support vector indexing with a dynamic index. The two most widely used architectures, x86-64 and

AArch64, do not. During instruction selection, such IR operations will get lowered to a stack spill and a load of the element from memory.

### [Slide 221] LLVM-IR: shufflevector

- Instruction to reorder values and resize vectors
- `shufflevector <n x ty> %x, <n x ty> %y, <m x i32> %mask`
  - %x, %y – values to shuffle, must have same size
  - %mask – element indices for result (0..<n refer to %x, n..<2n to %y)
  - Result is of type <m x ty>



### [Slide 222] shufflevector: Examples

#### In-Class Exercise:

What do these instructions do and what is the result type?

1. `%a = insertelement <4 x i32> poison, i32 %x, i64 0`  
`%r = shufflevector <4 x i32> %a, <4 x i32> poison,`  
`<4 x i32> zeroinitializer`
2. `%r = shufflevector <4 x i32> %a, <4 x i32> %b,`  
`<4 x i32> <i32 0, i32 5, i32 2, i32 7>`
3. `%r = shufflevector <4 x i16> %a, <4 x i16> %b,`  
`<8 x i32> <i32 0, i32 1, i32 2, i32 3, i32 4, i32 5, i32 6, i32 7>`

*Solution on page 116.*

### [Slide 223] LLVM-IR with Vector Instructions: Example

#### In-Class Exercise:

- Transform this LLVM-IR function into scalar, idiomatic C code

```
define void @foo(ptr %0, ptr %1) {
  br label %3
3: %4 = phi i64 [ 0, %2 ], [ %12, %3 ]
  %5 = phi <4 x i64> [ <i64 0, i64 1, i64 2, i64 3>, %2 ], [ %13, %3 ]
  %6 = getelementptr inbounds i64, ptr %1, i64 %4
  %7 = load <4 x i64>, ptr %6, align 8
  %8 = icmp slt <4 x i64> %7, %5
  %9 = add nsw <4 x i64> %7, %5
  %10 = select <4 x i1> %8, <4 x i64> %9, <4 x i64> zeroinitializer
  %11 = getelementptr inbounds i64, ptr %0, i64 %4
  store <4 x i64> %10, ptr %11, align 8
```

---

```

%12 = add nuw i64 %4, 4
%13 = add <4 x i64> %5, <i64 4, i64 4, i64 4, i64 4>
%14 = icmp eq i64 %12, 2048
br i1 %14, label %15, label %3
15: ret void
}

```

*Solution on page 116.*

---

## [Slide 224] LLVM-IR: Lowering Intrinsics

- Intrinsics translated to native LLVM-IR if possible
- + Allows optimizations
- Intent of programmer might get lost

```

#include <immintrin.h>
__m128 func(__m128 a, __m128 b) {
    __m128 rev = _mm_shuffle_epi32(a + b, 0x1b);
    return _mm_round_ps(rev, _MM_FROUND_TO_NEG_INF);
}

define <4 x float> @func(<4 x float> %0, <4 x float> %1) {
    %3 = fadd <4 x float> %0, %1
    %4 = shufflevector <4 x float> %3, <4 x float> poison, <4 x i32> <i32 3, i32 2, i32 1, i32 0>
    %5 = tail call <4 x float> @llvm.x86.sse41.round.ps(<4 x float> %4, i32 1)
    ret <4 x float> %5
}

declare <4 x float> @llvm.x86.sse41.round.ps(<4 x float>, i32 immarg)

```

## [Slide 225] SIMD Programming: Single Program, Multiple Data (SPMD)

- So far: manual vectorization
- Observation: same code is executed on multiple elements
- Idea: tell compiler to vectorize handling of single element
  - Splice code for element into separate function
  - Tell compiler to generate vectorized version of this function
  - Function called in vector-parallel loop
- Needs annotation of variables
  - Varying: variables that differ between lanes
  - Uniform: variables that are guaranteed to be the same (basically: scalar values that are broadcasted if necessary)

## [Slide 226] SPMD: Example (OpenMP)

```

#pragma omp declare simd
int foo(int x, int y) {
    return x + y;
}

```

- Compiler generates version that operates on vector

```

foo:
    add edi, esi
    mov eax, edi
    ret

```

```
_ZGVxN4vv_foo:
    padd xmm0, xmm1
    ret
```

### [Slide 227] SPMD: Example (OpenMP)

```
#pragma omp declare simd uniform(y)
int foo(int x, int y) {
    return x + y;
}
```

- Uniform: always same value

```
foo:
    add edi, esi
    mov eax, edi
    ret
```

```
_ZGVxN4vu_foo:
    movd xmm1, eax
    pshufd xmm2, xmm1, 0
    padd xmm0, xmm2
    ret
```

### [Slide 228] SPMD: Example (OpenMP) – if/else

```
#pragma omp declare simd
int foo(int x, int y) {
    int res;
    if (x > y) res = x;
    else res = y - x;
    return res;
}
```

- Diverging control flow: all paths are executed

```
foo:
    mov eax, esi
    sub eax, edi
    cmp edi, esi
    cmovg eax, edi
    ret
_ZGVxN4vv_foo:
    movdqa xmm2, xmm0
    pcmpgtd xmm0, xmm1
    psubd xmm1, xmm2
    pblendvb xmm1, xmm2, xmm0
    movdqa xmm0, xmm1
    ret
```

### [Slide 229] SPMD to SIMD: Handling if/else

- Control flow solely depending on uniforms: nothing different
- Otherwise: control flow may diverge

- 
- Different lanes may choose different execution paths
  - But: CPU has only one control flow, so all paths must execute
  - Condition becomes mask, mask determines result
  - After insertion of masks, linearize control flow
    - Relevant control flow now encoded in data through masks
  - Problem: side-effects prevent vectorization

#### [Slide 230] SPMD to SIMD: Handling Loops

- Uniform loops: nothing different
- Otherwise: need to retain loop structure
  - “active” mask added to all loop iterations
  - Loop only terminates once all lanes terminate (active is zero)
  - Lanes that terminated early need their values retained
- Approach also works for nested loops/conditions
- Irreducible loops need special handling<sup>2</sup>

#### [Slide 231] SPMD Implementations on CPUs

- OpenMP SIMD functions
  - Need to be combined with `#pragma omp simd` loops
- Intel ispc<sup>3</sup> (Implicit SPMD Program Compiler)
  - Extension of C with keywords `uniform`, `varying`
  - Still active and interesting history<sup>4</sup>
- OpenCL on CPU
  - Very similar programming model
  - But: higher complexity for communicating with rest of application

#### [Slide 232] SIMD Programming: SPMD on CPUs

- Semi-explicit vectorization
- Programmer chooses level of vectorization
  - E.g., inner vs. outer loop
- Compiler does actual work
- + Allows simple formulation of complex control flow
- Compilers often fail at handling complex control flow well
  - Loops are particularly problematic

---

<sup>2</sup>R Karrenberg and S Hack. “Whole-function vectorization”. In: *CGO*. 2011, pp. 141–150.

<sup>3</sup>M Pharr and WR Mark. “ispc: A SPMD compiler for high-performance CPU programming”. In: *InPar*. 2012, pp. 1–13.

<sup>4</sup><https://pharr.org/matt/blog/2018/04/30/ispc-all>

### [Slide 233] SIMD Programming: Auto-vectorization

- Idea: programmer is too incompetent/busy, let compiler do vectorization
- Inherently difficult and problematic, after decades of research
  - Recognizing and matching lots of patterns
  - Instruction selection becomes more difficult
  - Compiler lacks domain knowledge about permissible transformations
- Executive summary of the state of the art:
  - Auto-vectorization works well for very simple cases
  - For “medium complexity”, code is often suboptimal
  - In many cases, auto-vectorization fails on unmodified code

### [Slide 234] Auto-vectorization Strategies

- Loop Vectorization
  - Try to transform loop body into vectors with  $n$  lanes
  - Often needs tail loop for remainder that doesn't fill a vector
  - Extremely common
- Superword-level Parallelism (SLP)
  - Vectorize constructs outside of loops
  - Detect neighbored stores, try to fold operations into vectors

### [Slide 235] Loop Vectorization: Strategy

- Only consider innermost loop (at first)
1. Check legality: is vectorization possible at all?
    - Only vectorizable data types and operations used
    - No loop-carried dependencies, overlapping memory regions, etc.
  2. Check profitability: is vectorization beneficial?
    - Consider: runtime checks, gather/scatter, masked operations, etc.
    - Needs information about target architecture
  3. Perform transformation

### [Slide 236] Outer Loop Vectorization

- Vectorizing the innermost loop not always beneficial
  - Example 1: inner loop has only few iterations
  - Example 2: inner loop has loop-carried dependencies
- Thus: need to consider outer loops as well
  - Also: vectorization on multiple levels might be beneficial
- Very limited support in compilers, if any



---

### [Slide 237] Auto-vectorization is Hard

- Biggest problem: data dependencies
  - Resolving loop-carried dependencies is difficult
- Memory aliasing
  - Overlapping arrays, or – worse – loop counter
- Which loop level to vectorize? Multiple?
- Loop body *might* impact loop count
- Function calls, e.g. for math functions
- Strided memory access (e.g., only every n-th element)
- Choosing vectorization level (outer loop *might* be better)
- Is vectorization profitable *at all*?
- Often black box to programmer, preventing fine-grained tuning

### [Slide 238] Auto-Vectorization: Examples

---

#### In-Class Exercise:

Compile<sup>a</sup> the functions from `ex06.txt` with vectorization remarks.

```
clang -S -emit-llvm -O3 -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize  
-Rpass-missed=loop-vectorize
```

- Does vectorization occur?
- What additional output is provided in the optimization remarks?
- If so: what is vectorized? How?
- Does the result match your expectation?

<sup>a</sup><https://godbolt.org/z/aqbb937cb>

---

### [Slide 239] Vectorization – Summary

- SIMD is an easy way to improve performance numbers of CPUs
- Most general-purpose ISAs have one or more SIMD extensions
- Recent trend: variably-length vectors
- Inline Assembly: easiest for compiler, but extremely tedious
- Intrinsics: best trade-off towards performance and usability
- Target-independent operations: slightly increase portability
- SPMD: strategy dominant for GPU programming
- Auto-vectorization: very hard, unsuited for complex code

### [Slide 240] Vectorization – Questions

- Why do modern CPUs provide SIMD extensions?
- Why do variable-length SIMD extensions have higher runtime costs?
- How are SIMD intrinsics lowered to LLVM-IR?

- What is the downside of target-independent vector operations?
- How can if/else/for constructs be vectorized?
- What is the difference between a uniform and a varying variable?
- Why is auto-vectorization often sub-par to manual optimization?

## 8. Instruction Selection

### 8.1. Overview

#### [Slide 242] Code Generation – Overview

- Instruction Selection
  - Map IR to assembly
  - Keep code shape and storage; change operations
- Instruction Scheduling
  - Optimize order to hide latencies and reduce register pressure
  - Keep operations unmodified
- Register Allocation
  - Map virtual to architectural registers and stack
  - Adds operations (spilling), changes storage

#### [Slide 243] Instruction Selection (ISel) – Overview

- Find machine instructions to implement abstract IR
- Typically separated from scheduling and register allocation
- Input: IR code with abstract instructions
- Output: lower-level IR code with target machine instructions

```
i64 %10 = add %8, %9
i8 %11 = trunc %10
i64 %12 = const 24
i64 %13 = add %7, %12
store %11, %13

i64 %10 = ADD %8, %9
STRB %10, [%7+24]
```

#### [Slide 244] ISel – Typical Constraints

- Target offers multiple ways to implement operations
  - `imul x, 2, add x, x, shl x, 1, lea x, [x+x]`
- Target operations have more complex semantics
  - E.g., combine truncation and offset computation into store
  - Can have multiple outputs, e.g. `value+flags`, `quotient+remainder`
- Target has multiple register sets, e.g. GP and FP/SIMD
  - Important to consider even before register allocation
- Target requires specific instruction sequences
  - E.g., for macro fusion

- Often represented as pseudo-instructions until assembly writing

RISC-V, for example, is an architecture which specifies several instruction sequences that are intended to be macro-fused by the processor for improved performance.

### [Slide 245] Optimal ISel

- Find *most performant* instruction sequence with same semantics (?)
  - I.e., no program with better “performance” exists
  - Performance  $\approx$  instructions associated with specific costs
- Problem: optimal code generation is **undecidable**

The halting problem can be reduced to optimal code generation: if a program continues forever, the program has to be optimized to a simple endless loop to be optimal.

- Alternative: optimal *tiling* of IR with machine code instructions
  - IR as dataflow graph, instr. tiles to optimally cover graph
  - $\mathcal{NP}$ -complete<sup>1</sup>

The cited paper shows how SAT can be reduced to optimal tiling of a DAG.

- Additional complication: many different ways to express same computation

### [Slide 246] Avoiding ISel Altogether

Use an interpreter

- + Fast “compilation time”, easy to implement
- Slow execution time
- Best if code is executed once

Interpreters are in many cases a good alternative to writing code generators — the latter are highly platform-dependent and involve several hard problem, while the former are easy to write in portable languages.

## 8.2. Macro Expansion

### [Slide 247] Macro Expansion

- Expand each IR operation with corresponding machine instrs

---

<sup>1</sup>DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54.  
URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.

<code>%5 = add %1, 12345</code>	→	<code>%5a = movz 12345</code> <code>%5 = add %1, %5a</code>
<code>%6 = and %2, 7</code>	→	<code>%6 = and %2, 7</code>
<code>%7 = shl %5, %6</code>	→	<code>%7a = lsl %5, %6</code> <code>%7b = cmp %6, 64</code> <code>%7 = csel %7a, xzr, %7b, lo</code>

**[Slide 248] Macro Expansion**

- Oldest approach, historically also does register allocation
  - Also possible by walking AST
- + Very fast, linear time, simple to implement, easy to port
- Inefficient and large output code
- Used by, e.g., LLVM FastISel, Go, GCC

**[Slide 249] Peephole Optimization**

- Plain macro expansion leads to suboptimal results
- Idea: replace inefficient instruction sequences<sup>2</sup>
- Originally: physical window over assembly code
  - Replace with more efficient instructions having same effects
  - Possibly with allocated registers
- Extension: do expansion before register allocation<sup>3</sup>
  - Expand IR into Register Transfer Lists (RTL) with temporary registers
  - While *combining*, ensure that each RTL can be implemented as single instr.

GCC's RTL is heavily inspired by the original idea of register transfer lists. However, due to severe limitations when optimizing code, the implementation was later expanded and some RTL passes also make use of SSA form.

**[Slide 250] Peephole Optimization**

- Originally covered only adjacent instructions
- Can also use logical window of data dependencies
  - Problem: instructions with multiple uses
  - Needs more sophisticated matching schemes for data deps. ⇒ Tree-pattern matching
- + Fast, also allows for target-specific sequences
- Pattern set grows large, limited potential
- Widely used today at different points during compilation

<sup>2</sup>WM McKeeman. "Peephole optimization". In: *CACM* 8.7 (1965), pp. 443–444. URL: <https://dl.acm.org/doi/pdf/10.1145/364995.365000>.

<sup>3</sup>JW Davidson and CW Fraser. "Code selection through object code optimization". In: *TOPLAS* 6.4 (1984), pp. 505–526. URL: <https://dl.acm.org/doi/pdf/10.1145/1780.1783>.

## 8.3. Tree Covering

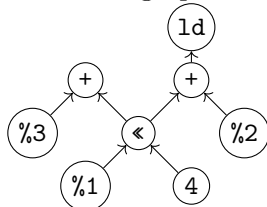
### [Slide 251] ISel as Graph Covering – High-level Intuition

- Idea: represent program as data flow graph
- Tree: expression, comb. of single-use SSA instructions (local ISel)
- DAG: data flow in basic block, e.g. SSA block (local ISel)
- Graph: data flow of entire function, e.g. SSA function (global ISel)
- ISA “defines” *pattern set* of trees/DAGs/graphs for instrs.
- Cover data flow tree/DAG/graph with least-cost combination of patterns
  - Patterns in data flow graph may overlap
  - For non-global ISel: values used outside of block must be generated

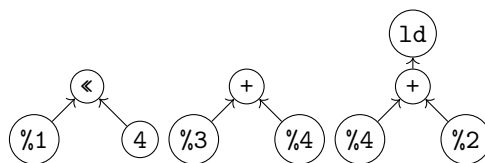
### [Slide 252] Tree Covering: Converting SSA into Trees

- SSA form:  
%4 = shl %1, 4  
%5 = add %2, %4  
%6 = add %3, %4  
%7 = load %5  
live-out: %6, %7

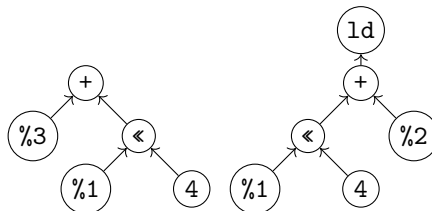
- Data flow graph:



- Method 1: Edge Splitting



- Method 2: Node Duplication



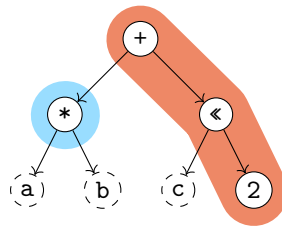
**[Slide 253] Tree Covering: Patterns**

	Pattern	Cost	Instruction
$P_0$	$GP_{R1} \rightarrow \ll(GP_{R2}, K_1)$	1	<code>lsl <math>R_1</math>, <math>R_2</math>, #<math>K_1</math></code>
$P_1$	$GP_{R1} \rightarrow +(GP_{R2}, GP_{R3})$	1	<code>add <math>R_1</math>, <math>R_2</math>, <math>R_3</math></code>
$P_2$	$GP_{R1} \rightarrow +(\ll(GP_{R2}, K_1), GP_{R3})$	2	<code>add <math>R_1</math>, <math>R_2</math>, <math>R_3</math>, lsl #<math>K_1</math></code>
$P_3$	$GP_{R1} \rightarrow +(\ll(GP_{R2}, K_1), GP_{R2})$	2	<code>add <math>R_1</math>, <math>R_3</math>, <math>R_2</math>, lsl #<math>K_1</math></code>
$P_4$	$GP_{R1} \rightarrow \text{ld}(GP_{R2})$	2	<code>ldr <math>R_1</math>, [<math>R_2</math>]</code>
$P_5$	$GP_{R1} \rightarrow \text{ld}+(\ll(GP_{R2}, GP_{R3}))$	2	<code>ldr <math>R_1</math>, [<math>R_2</math>, <math>R_3</math>]</code>
$P_6$	$GP_{R1} \rightarrow \text{ld}+(\ll(GP_{R2}, \ll(GP_{R3}, 3)))$	3	<code>ldr <math>R_1</math>, [<math>R_2</math>, <math>R_3</math>, lsl #3]</code>
$P_7$	$GP_{R1} \rightarrow \text{ld}+(\ll(GP_{R2}, 3), GP_{R3})$	3	<code>ldr <math>R_1</math>, [<math>R_3</math>, <math>R_2</math>, lsl #3]</code>
$P_8$	$GP_{R1} \rightarrow *(GP_{R2}, GP_{R3})$	3	<code>madd <math>R_1</math>, <math>R_2</math>, <math>R_3</math>, xzr</code>
$P_9$	$GP_{R1} \rightarrow +(*(\ll(GP_{R2}, GP_{R3}), GP_{R4}))$	3	<code>madd <math>R_1</math>, <math>R_2</math>, <math>R_3</math>, <math>R_4</math></code>
$P_{10}$	$GP_{R1} \rightarrow K_1$	1	<code>mov <math>R_1</math>, <math>K_1</math></code>
$\vdots$	$\vdots$	$\vdots$	$\vdots$

**[Slide 254] Tree Covering: Greedy/Maximal Munch**

- Top-down always take largest pattern
- Repeat for sub-trees, until everything is covered

+ Easy to implement, fast  
 – Result might be non-optimum

**[Slide 255] Tree Covering: Greedy/Maximal Munch – Example**

Matching Patterns:

- +:  $P_1$  – cost 1 – covered nodes: 1
- +:  $P_2$  – cost 2 – covered nodes: 3
- +:  $P_9$  – cost 3 – covered nodes: 2
- \*:  $P_8$  – cost 3 – covered nodes: 1 – best

Total cost: 5

```
madd %1, %a, %b, xzr
add %2, %1, %c, lsl #2
```

**[Slide 256] Tree Covering: with LR-Parsing?**

- Can we use (LR-)parsing for instruction selection? Yes!<sup>4</sup>
  - Pattern set = grammar; IR (in prefix notation) = input

**Advantages**

- Possible in linear time
- Can be formally verified
- Implementation can be generated automatically

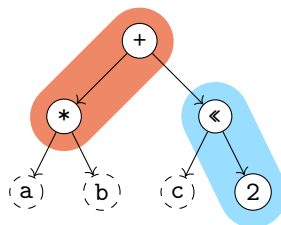
**Disadvantages**

- Constraints must map to non-terminals
  - Constant ranges, reg types, ...
- CISC: handle all operand combinations
  - Large grammar (impractical)
  - Refactoring into non-terminals
- Ambiguity hard to handle optimally

**[Slide 257] Tree Covering: Dynamic Programming<sup>5</sup>**

- Step 1: compute cost matrix, bottom-up for all nodes
  - Matrix: tree node  $\times$  register bank (different patterns might yield the same result in different register banks)
  - Cost is sum of pattern and sum of children costs
  - Always store cheapest rule and cost
- Step 2: walk tree top-down using rules in matrix
  - Start with goal, follow rules in matrix
- Time linear w.r.t. tree size

**[Slide 258] Tree Covering: Dynamic Programming – Example**



---

<sup>4</sup>RS Glanville and SL Graham. “A new method for compiler code generation”. In: *POPL*. 1978, pp. 231–254. URL: <https://dl.acm.org/doi/pdf/10.1145/512760.512785>.

<sup>5</sup>AV Aho, M Ganapathi, and SWK Tjiang. “Code generation using tree matching and dynamic programming”. In: *TOPLAS* 11.4 (1989), pp. 491–516. URL: <https://dl.acm.org/doi/pdf/10.1145/69558.75700>.



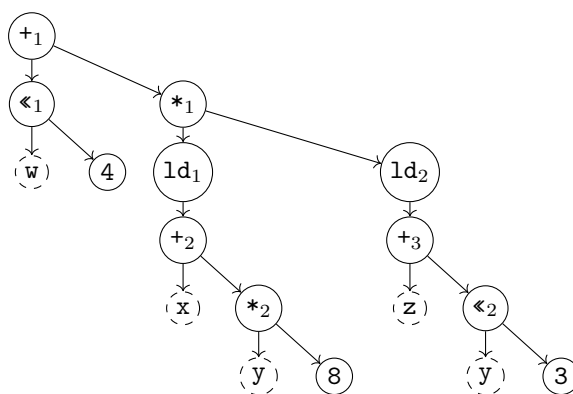
Node: +  
 Pattern:  $P_9: GP \rightarrow +(* (GP, GP), GP)$   
 Pat. Cost: 3  
 Cost Sum: 4

Node		+	*	«	2
GP	Cost	4	3	1	1
	Pattern	$P_9$	$P_8$	$P_0$	$P_{10}$

### [Slide 259] Tree Covering: Dynamic Programming – Example

#### In-Class Exercise:

Find a tree covering using the dynamic programming algorithm.



*Solution on page 116.*

### [Slide 260] Tree Covering: Dynamic Programming – Off-line Analysis

- Cost analysis can actually be *precomputed*<sup>6</sup>
- Idea: annotate each node with a state based on child states
- Lookup node label from precomputed table (one per register bank)
- Significantly improves compilation time
- But: Tables can be large, need to cover all possible (sub-)trees
- Variation: dynamically compute and cache state tables<sup>7</sup>

<sup>6</sup>A Balachandran, DM Dhamdhere, and S Biswas. “Efficient retargetable code generation using bottom-up tree pattern matching”. In: *Computer Languages* 15.3 (1990), pp. 127–140.

<sup>7</sup>MA Ertl, K Casey, and D Gregg. “Fast and flexible instruction selection with on-demand tree-parsing automata”. In: *PLDI* 41.6 (2006), pp. 52–60.

**[Slide 261] Tree Covering**

- + Efficient: linear time to find local optimum
- + Better code than pure macro expansion
- + Applicable to many ISAs
- Common sub-expressions cannot be represented
  - Need either edge split (prevents using complex instructions) or node duplication (redundant computation  $\Rightarrow$  inefficient code)
- Cannot make use of multi-output instructions (e.g., `divmod`)

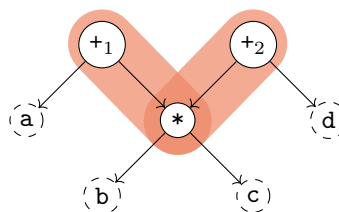
**8.4. DAG Covering****[Slide 262] DAG Covering**

- Idea: lift restriction of trees, operate on data flow DAG
  - Reminder: an SSA basic block already forms a DAG
- Trivial approach: split into trees ☹️
- Least-cost covering is  $\mathcal{NP}$ -complete<sup>8</sup>

**[Slide 263] DAG Covering: Adapting Dynamic Programming I<sup>9</sup>**

- Step 1: compute cost matrix, bottom-up for all nodes
    - As before; make sure to visit each node once
  - Step 2: iterate over DAG top-down
    - Respect that multiple roots exist: start from all roots
    - Mark visited node/register combinations: avoid redundant emit
- + Linear time
- Generally not optimal, only for specific grammars

“Roots” in this context refers to all values that are required by other basic blocks and therefore must be computed and stored in some register.

**[Slide 264] DAG Covering: Adapting Dynamic Programming I – Example**

---

<sup>8</sup>DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54. URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.

<sup>9</sup>MA Ertl. “Optimal code selection in DAGs”. In: *POPL*. 1999, pp. 242–249. URL: <https://dl.acm.org/doi/pdf/10.1145/292540.292562>.

Total cost: 6  
 madd %1, %b, %c, %a  
 madd %2, %b, %c, %d  
 Optimal cost: 5     $\rightsquigarrow$  non-optimal result

	Node	+ <sub>2</sub>	+ <sub>1</sub>	*
GP	Cost	3	3	3
	Pattern	$P_9$	$P_9$	$P_8$

### [Slide 265] DAG Covering: Adapting Dynamic Programming II<sup>10</sup>

- Step 1: compute cost matrix, bottom-up (as before)
  - Step 2: iterate over DAG top-down (as before)
  - Step 3: identify overlaps and check whether split is beneficial
    - Mark nodes which should not be duplicated as *fixed*
  - Step 4: as step 1, but skip patterns that *include* fixed nodes
  - Step 5: as step 2
- + Probably fast? “Near-optimal”?
- Generally not optimal, superlinear time

### [Slide 266] DAG Covering: ILP<sup>11</sup>

- Idea: model ISel as integer linear programming (ILP) problem
- $P$  is set of patterns with cost and edges,  $V$  are DAG nodes
- Variables:  $M_{p,v}$  is 1 iff a pattern  $p$  is rooted at  $v$

$$\begin{aligned}
 &\text{minimize} && \sum_{p,v} p.\text{cost} \cdot M_{p,v} \\
 &\text{subject to} && \forall r \in \text{roots}. \sum_p M_{p,r} \geq 1 \\
 & && \forall p, v, e \in p.\text{edges}(v). M_{p,v} - \sum_{p'} M_{p',e} \leq 0 \\
 & && M_{p,v} \in \{0, 1\}
 \end{aligned}$$

Minimize cost for all matched patterns s.t. every root has a match and every input of a match has a match.

- + Optimal result
- Practicability beyond small programs questionable (at best)

### [Slide 267] DAG Covering: Greedy/Maximal Munch

- Top-down, start at roots, always take largest pattern
- Repeat for remaining roots until whole graph is covered

<sup>10</sup>DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54.  
 URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.

<sup>11</sup>DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54.  
 URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.

- + Easy to implement, reasonably fast
- Result often non-optimal
- Used by: LLVM SelectionDAG

### 8.5. Graph Covering

#### [Slide 268] Graph Covering

- Idea: lift limitation of DAGs, cover entire function graphs
- Better handling of predication and VLIW bundling
  - E.g., hoisting instructions from a conditional block
- Allows to handle instructions that expand to multiple blocks
  - `switch`, `select`, etc.
- May need new IR to model control flow in addition to data flow
- In practice: only used by adapting methods showed for DAGs
- Used by: Java HotSpot Server, LLVM GlobalISel (all tree-covering)

### 8.6. ISel in Practice

#### [Slide 269] Flawed Assumptions

- Cost model is fundamentally flawed
- ⇒ “Optimal” ISel doesn’t really mean anything
- Out-of-order execution: costs are not linear
  - Instructions executed in parallel, might execute for free
  - Possible contention of functional units
- Register allocator will modify instructions
- “Bad” instructions boundaries increase register requirements
  - More stack spilling  $\rightsquigarrow$  much slower code!

#### [Slide 270] Instruction Selection in Practice

- Most compilers use some form of greedy tree/DAG pattern matching
- Later stages use peephole optimizations
  - Basically also tree/DAG matching on machine operations
- Distinction between tree/DAG/graph matching somewhat artificial<sup>12</sup>

Problem in practice: implementing the huge amount of required patterns

- LLVM X86 back-end has 73k lines C++ for lowering + auto-generated patterns
- Needs lots of handling for corner cases, e.g. immediates
- Coming up with the patterns is often non-trivial

---

<sup>12</sup>My personal opinion.

To illustrate the last point: x86 has no instruction to reverse bits of an integer. However, with the GFNI extension, the instruction `vgf2p8affineqb` was introduced, which can be configured in a way that it performs a bit-reverse operation<sup>a,b</sup> (Would you come up with this pattern?)

<sup>a</sup><https://github.com/llvm/llvm-project/pull/81764>

<sup>b</sup>Example: <https://godbolt.org/z/34fW6hc64>

## 8.7. LLVM Instruction Selection

### [Slide 271] LLVM Back-end: Overview

- LLVM-IR → Machine IR: instruction selection + scheduling
  - MIR is SSA-representation of target instructions
  - Selectors: SelectionDAG, FastISel, GlobalISel
  - Also selects register bank (GP/FP/...) – required for instruction
  - Annotates registers: calling convention, encoding restrictions, etc.
- MIR: minor (peephole) optimizations
- MIR: register allocation
- MIR: prolog/epilog insertion (stack frame, callee-saved regs, etc.)
- MIR → MC: translation to machine code

### [Slide 272] LLVM MIR Example

```
define i64 @fn(i64 %a,i64 %b,i64 %c) {
  %shl = shl i64 %c, 2
  %mul = mul i64 %a, %b
  %add = add i64 %mul, %shl
  ret i64 %add
}
# YAML with name, registers, frame info
body: |
  bb.0 (%ir-block.0):
    liveins: $x0, $x1, $x2

    %2:gpr64 = COPY $x2
    %1:gpr64 = COPY $x1
    %0:gpr64 = COPY $x0
    %3:gpr64 = MADDXrrr %0, %1, $x2r
    %4:gpr64 = ADDXrs killed %3, %2, 2
    $x0 = COPY %4
    RET_ReallyLR implicit $x0
llc -march=aarch64 -stop-after=finalize-isel
```

### [Slide 273] LLVM MIR Example

#### In-Class Exercise:

Analyze the Machine IR of the following code<sup>a</sup>. (Also consult the reference<sup>b</sup>.)

- What is the difference between physical and virtual registers?
- What do `killed` and `implicit-def` mean?
- How do branches and calls differ from LLVM-IR?

```
// clang --target=aarch64 -c -mllvm -stop-after=finalize-isel -O1 -o -  
// Also try -O0, -O2, -g, and --target=x86_64.
```

```
[[clang::noinline]] int foo(int n) {  
    int r = 1;  
    while (n) { r *= n << n; n--; }  
    return r;  
}  
int bar(int n) { return foo(n) + n; }
```

<sup>a</sup><https://godbolt.org/z/zj9WM84Wh>

<sup>b</sup><https://llvm.org/docs/MIRLangRef.html>

---

### [Slide 274] LLVM: Instruction Selectors

#### FastISel

- Uses macro expansion
- Low compile-time
- Code quality poor
- Only common cases
- Otherwise: fallback to SelectionDAG
- Default for `-O0`

#### SelectionDAG

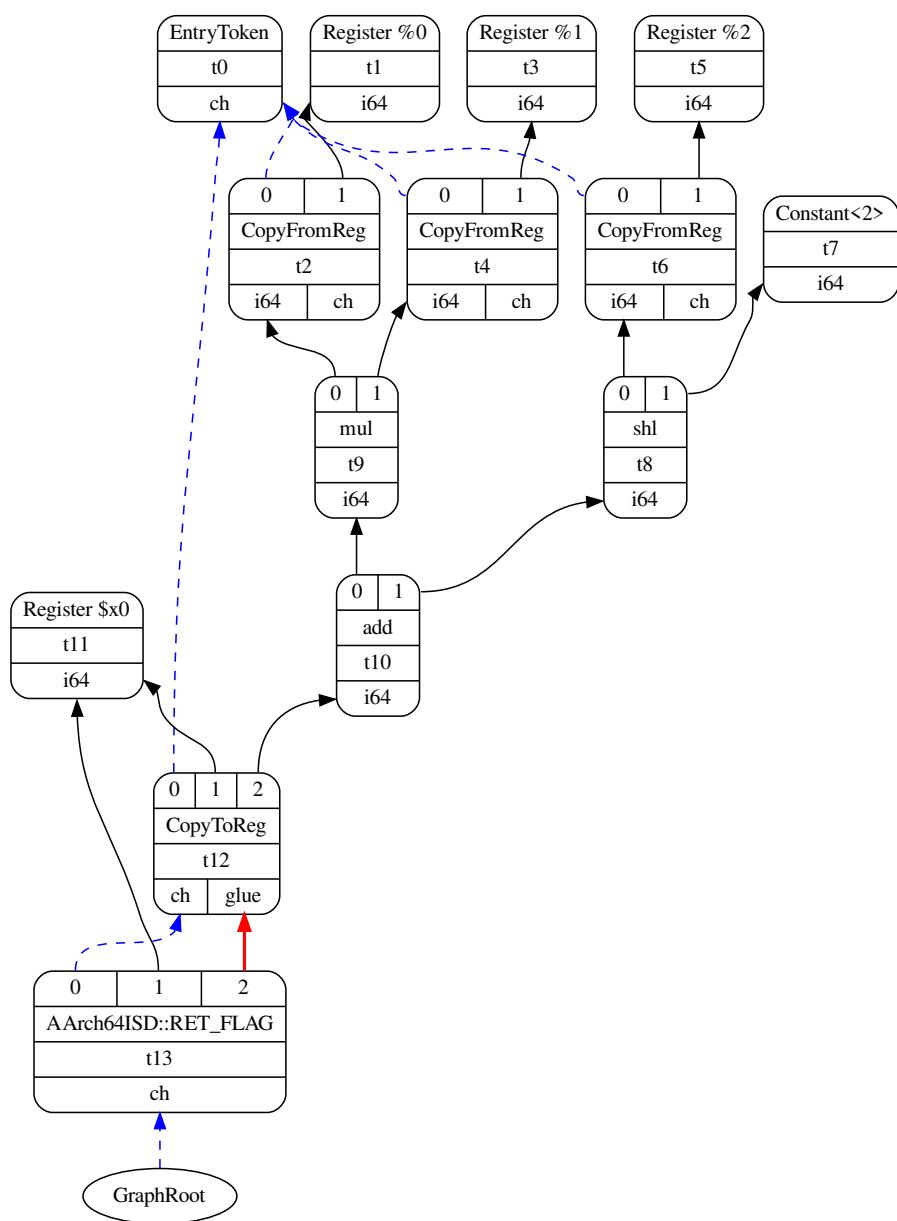
- Converts each block into separate DAGs
- Greedy tree matching
- Slow, but good code
- Handles all cases
- No cross-block opt. (done in DAG building)
- Default

#### GlobalISel

- Conv. to generic-MIR then legalize to MIR
- Reuses SD patterns
- Faster than SelDAG
- Few architectures
- Handles many cases, SelDAG-fallback
- Default AArch64 `-O0`

### [Slide 275] LLVM SelectionDAG: IR to ISelDAG

- Construct DAG for basic block
  - EntryToken as ordering chain
- Legalize data types
  - Integers: promote or expand into multiple



isel input for fn:

Figure 8.1.: ISel DAG right before instruction selection.

- Vectors: widen or split (or scalarize)
- Legalize operations
  - E.g., conditional move, etc.
- Optimize DAG, e.g. some pattern matching, removing unneeded sign/zero extensions

See Figure 8.1.

```
llc -march=aarch64 -view-isel-dags
```

Note: needs LLVM debug build

### [Slide 276] LLVM SelectionDAG: ISelDAG to DAG

- Mainly pattern matching
- Simple patterns specified in TableGen
  - Matching/selection compiled into bytecode
  - `SelectionDAGISel::SelectCodeCommon()`
- Complex selections done in C++
- Scheduling: linearization of graph

See Figure 8.2.

```
llc -march=aarch64 -view-sched-dags
```

Note: needs LLVM debug build

### [Slide 277] LLVM GlobalISel

- DAGs do have limitations w.r.t. instruction selection
- Graph loses original scheduling information (why problematic?)

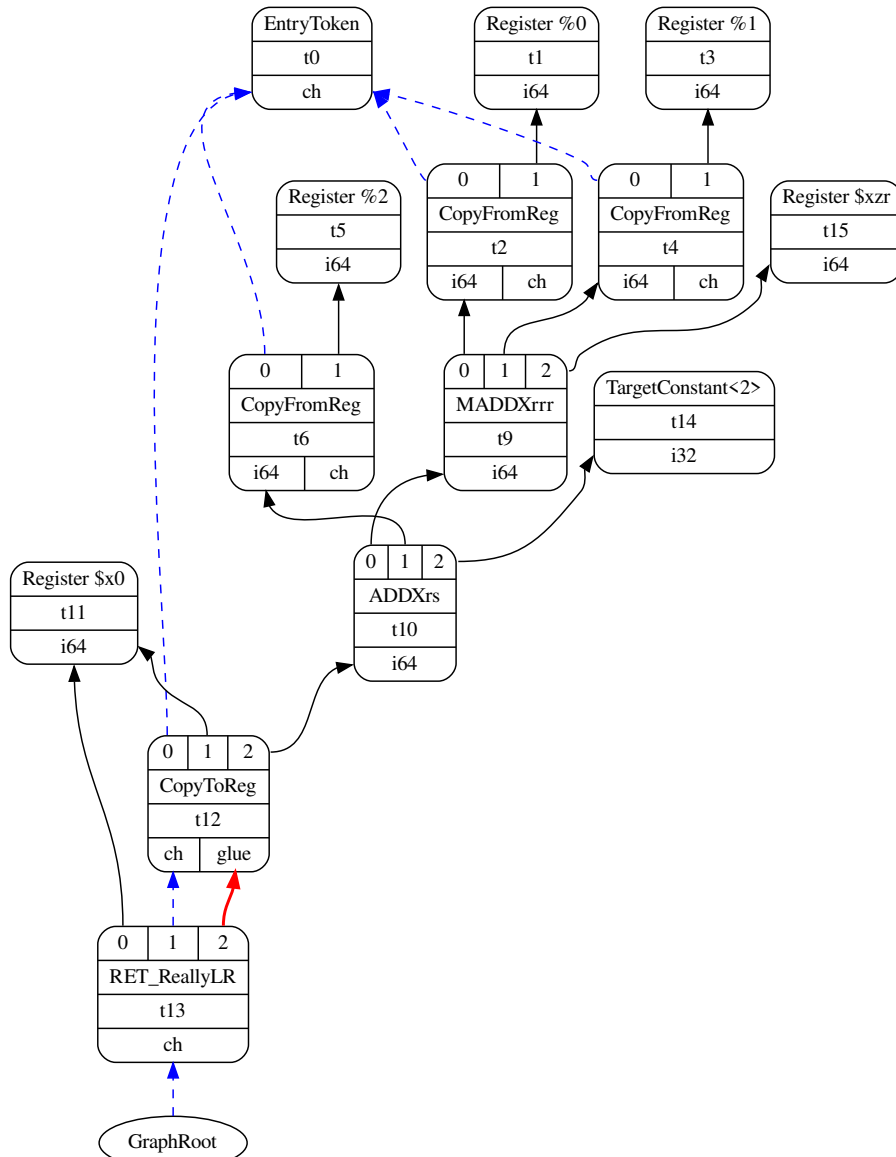
The original instruction ordering, e.g., as occurring in the input source, is lost during instruction selection. This gives programmers less opportunities to influence the order of operations in the resulting machine code and also can contribute to frequent jumps between lines when stepping through the code in a debugger.

- Building a separate IR is expensive
- Core idea of GlobalISel: do legalization and ISel in MachineIR
- MIR is target-specific  $\rightsquigarrow$  introduce generic operations
- Permit matching/combining instructions from different blocks

### [Slide 278] LLVM GlobalISel: Passes

- IRTranslator: translate LLVM-IR into gMIR
  - One-to-one mapping, except call and switch
- PreLegalizerCombiner
- Localizer: try to improve placement of constant materialization
- Legalizer: replace illegal instructions with legal gMIR instrs





scheduler input for fn:

Figure 8.2.: X86 DAG after Instruction Selection

Illegal instructions are instructions that the target does not support, e.g. due to unsupported types. For example, AArch64 only supports 32/64-bit integer arithmetic, so operations on smaller integers (e.g., `i1`) need to be promoted to a legal type; likewise for larger integers, which need to be split up.

- `PostLegalizerCombiner`
- `RegBankSelect`: assign register bank (reduce copies between banks)

While moves between registers of the same register file are very cheap on modern CPUs, moves between different register files (e.g., general-purpose and floating-point registers) remain somewhat expensive.

- `InstructionSelect`

### [Slide 279] Instruction Selection – Summary

- Instruction Selection: transform generic into arch-specific instructions
- Often focus on optimizing tiling costs
- Target instructions often more complex, e.g., multi-result
- Macro Expansion: simple, fast, but inefficient code
- Peephole optimization on sequences/trees to optimize
- Tree Covering: allows for better tiling of instructions
- DAG/Graph Covering: support multi-res instrs., but  $\mathcal{NP}$ -complete
- Practical problems are very different from theory

### [Slide 280] Instruction Selection – Questions

- What is the (nowadays typical) input and output IR for `ISel`?
- Why is good instruction selection important for performance?
- Why is peephole optimization beneficial for nearly all `ISel` approaches?
- How can peephole opt. be done more effectively than on neighboring instrs.?
- What are options to transform an SSA-IR into data flow trees?
- Why is a greedy strategy not optimal for tree pattern matching?
- When is DAG covering beneficial over tree covering?
- Which `ISel` strategies does LLVM implement? Why?
- What are advantages/disadvantages of `SelDAG` and `GlobalISel`?

## A. Exercise Solutions

### [Slide 22]

Some suggestions:

- Sequences of `+/-</>`
- `[-]` → set zero
- `[>]` → find next zero (`memchr`)
- `[->+>+<]` → add to next two siblings, set zero
- `[->+++<]` → add 3 times to next sibling, set zero
- ...

### [Slide 28]

Indirect/direct threading and computed `goto`/tail calls are orthogonal.

- Computed `goto` is a non-standard feature. A lot of state can be kept in registers across different bytecode operations, this is done automatically by the compiler. The possibility of indirect jumps inside the function may prevent some compiler optimizations.
- Tail calls (largely) rely on standard features and the code is typically more maintainable (e.g., operations can be implemented in separate functions). Some state can be kept in registers across operations, but this has to be implemented manually; the usable number of preservable registers is defined by the calling convention and is architecture-dependent. Non-standard calling convention (e.g., `ghccc`) can be used to increase this limit. Compilers cannot perform optimization across the different functions.
- Indirect threading permits a small opcode (e.g., one byte as in this example), which leads to higher code density and therefore increased data locality.
- Direct threading avoids the indirection to the opcode handler at the cost of larger bytecode (in the example, an operation grows from 2 to 16 bytes). On modern out-of-order CPUs, avoiding the indirection might not lead to significant performance differences.

### [Slide 33]

As a simple approach, check the array boundaries at every pointer move, when reaching bounds, re-allocate to larger buffer and copy the elements. As pointer moves occur often, the bounds checks are rather expensive.

To improve efficiency, it is possible to use virtual memory by padding the buffer with known-unmapped pages (guard pages) and intercepting accesses to these pages (e.g., through a signal handler). On an access, the buffer can grow and/or be moved (e.g.,

through `mremap`). Care must be taken to ensure that the guard pages are large enough (the length of the program should be sufficient).

### [Slide 52]

Example for input: `a = 3 * 2 + 1;`

	Rec. Depth 1	Rec. Depth 2	Rec. Depth 3
<code>minPrec</code>	1		
<code>lhs</code>	<code>a</code>		
<code>op (prec/assoc)</code>	<code>= (2/r)</code>		
<code>minPrec</code>	1	2	
<code>lhs</code>	<code>a</code>	3	
<code>op (prec/assoc)</code>	<code>= (2/r)</code>	<code>* (12/l)</code>	
<code>minPrec</code>	1	2	13
<code>lhs</code>	<code>a</code>	3	2
<code>op (prec/assoc)</code>	<code>= (2/r)</code>	<code>* (12/l)</code>	<code>+ (11/l)</code>
<code>minPrec</code>	1	2	
<code>lhs</code>	<code>a</code>	<code>3*2</code>	
<code>op (prec/assoc)</code>	<code>= (2/r)</code>	<code>+ (11/l)</code>	
<code>minPrec</code>	1	2	12
<code>lhs</code>	<code>a</code>	<code>3*2</code>	1
<code>op (prec/assoc)</code>	<code>= (2/r)</code>	<code>+ (11/l)</code>	<code>; (0/-)</code>
<code>minPrec</code>	1	2	
<code>lhs</code>	<code>a</code>	<code>(3*2)+1</code>	
<code>op (prec/assoc)</code>	<code>= (2/r)</code>	<code>; (0/-)</code>	
<code>minPrec</code>	1		
<code>lhs</code>	<code>a=((3*2)+1)</code>		
<code>op (prec/assoc)</code>	<code>; (0/-)</code>		

### [Slide 60]

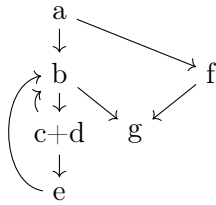
There are two ways of implementing a scoped hash table:

- Chain of hash maps:  $Scope = (Map[Name \rightarrow Type] \text{ names}, Scope \text{ parent})$ . This is, however, very slow for deeply nested scopes, as all hash maps of the parent scopes must be queried. Hash map lookups are fairly expensive.
- Hash map of lists:  $Map[Name \rightarrow List[Tuple[Depth, Type]]]$ . For every identifier, the type at a given scope nesting depth is stored. Invalidation can be implemented with an epoch counter for every depth. The downside is that this hash map can grow very large, as entries are never removed.

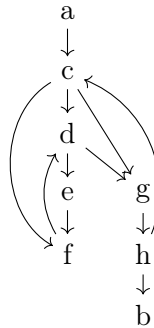
---

### [Slide 83]

Control flow graph for fn1:



Control flow graph for fn2:



### [Slide 99]

```
phis(v1, v2) {
entry:
    v3 = mul v1, v2 // a * b
    v4 = mul v2, v2 // b * b
    v5 = cmpgt v1, v4 // a > b * b
    br v5, ifthen, ifelse
ifthen:
    br header
header:
    phi1 = phi(ifthen: v1, body: v7) // a
    phi2 = phi(ifthen: 1, body: phi2) // c
    v6 = cmpgt phi1, 0 // a > 0
    br v6, body, cont
body:
    v7 = sub phi1, phi2 // a - c
    br header
cont:
    br ret
ifelse:
    v8 = mul v2, v2 // b * b
    br ret
ret:
    phi3 = phi(cont: phi1, ifelse: v8) // a
    ret phi3
}

swap(v1, v2, v3) {
entry:
    br header
header:
    phi1 = phi(entry: v3, body: v5) // c
    // Note: all phi nodes execute concurrently. Therefore, these two phi nodes swap a and b.
    phi2 = phi(entry: v1, body: phi3) // a
    phi3 = phi(entry: v2, body: phi2) // b
    v4 = cmpgt phi1, 0 // a > 0
    br v4, body, cont
body:
```

```
    v5 = sub phi1, 1 // c - 1
    br header
cont:
    ret phi2
}
```

[Slide 128]

```
int sw(int x) {
    switch (x) {
        case 4:
        case 100: return 12;
        case 5: return 32;
        case 8: return 9;
        default: return x;
    }
}
```

[Slide 129]

```
int sw(int x) {
    switch (x) {
        case 4:
        case 10: return 12;
        case 5: return 32;
        case 8: return 9;
        default: return x;
    }
}
```

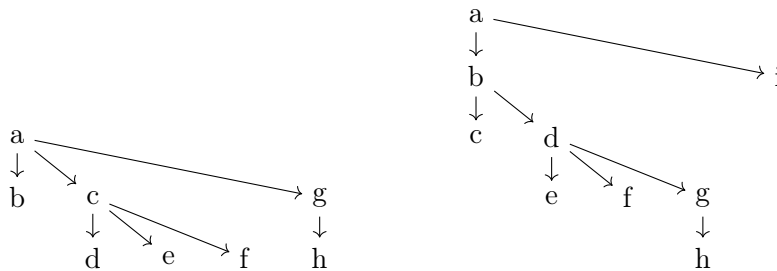
[Slide 149]

The algorithm never visits the `%else` block and therefore can prove that `%j4` and thus `%j2` are always the constant 1.

```
define i32 @fn() {
entry:
    br label %loop
loop:
    %k2 = phi i32 [ 0, %entry ], [ %k3, %ifmerge ]
    %kcond = icmp slt i32 %k2, 100
    br i1 %kcond, label %loopbody, label %ret
loopbody:
    br label %then
then:
    %k3 = add i32 %k2, 1
    br label %ifmerge
ifmerge:
    br label %loop
ret:
    ret i32 1
}
```

---

[Slide 157]



[Slide 171]

The first CFG has no reducible loop, so the algorithm as written will not find any loops.

The second CFG contains two reducible loops:

- Loop 1: d (header), e, f, g
- Loop 2: b (header), c, d, e, f, g, h

[Slide 189]

You can use Alive2 as validation tool: [https://alive2.llvm.org/ce/z/\\_PuqaK](https://alive2.llvm.org/ce/z/_PuqaK)

1. The neutral element of floating-point addition is `-0.0`; adding `+0.0` to `-0.0` results in `+0.0`, which is not a generally correct transformation.
2. This transformation, although non-obvious, is correct.
3. `sdiv` rounds towards zero, but `ashr`, so for e.g. `-3`, `src3` returns 0 while `tgt3` return `-1`.

[Slide 194]

You can use Alive2 as validation tool: <https://alive2.llvm.org/ce/z/HgpRmK>

1. Correct: `udiv exact` results in `poison` when 1 is not a multiple of the parameter, i.e., when the parameter is not 1. Thus, always returning 1 makes the target function more defined than the source function.
2. Not correct: `%cmpeq` is `poison` if `%x` is negative. The transformation would be correct if the `samesign` flag was dropped.
3. Not correct: integer addition is associative, but the `nsw` flag needs to be dropped.

[Slide 219]

The results diverge strongly depending on whether the operations is natively supported by the architecture. For some operations, the output is as intended:

```
// typedef uint32_t vecty __attribute__((vector_size(16)));
// vecty op(vecty a, vecty b) { return a + b; }
op:
    padd xmm0, xmm1
    ret
```

For some operations, some legalization needs to be applied, e.g., to widen the elements to the nearest supported size:

```
// typedef uint8_t vecty __attribute__((vector_size(8)));
// vecty op(vecty a, vecty b) { return a * b; }
op:
  punpcklbw xmm0, xmm0 // zero extend elements to 16 bit
  punpcklbw xmm1, xmm1
  pmullw xmm0, xmm1 // 16-bit multiplication
  pand xmm0, xmmword ptr [rip + .LCPI0_0] // {0x00ff, 0x00ff, ...}
  packuswb xmm0, xmm0
  ret
```

Other operations need to be scalarized, because the hardware doesn't support the vectorized operation, e.g. for division. This is of course *less* efficient than directly writing the scalarized code: First, the legalization of vector operations only happens inside the back-end and therefore earlier transformations will only see (and probably ignore) the vectorized code. Second, there needs to be code to extract the elements from the vector and re-insert them at the end.

The performance impact of using this type of vector addition is not visible in the source program; even if it looks optimized, the output might be worse than for unvectorized source code. This is part of the reason why using this style of vector programming is not very popular.

### [Slide 222]

1. Broadcast %x to all lanes of a <4 x i32>.
2. Interleave two vectors into a <4 x i32>, taking the elements 0 and 2 from the first vector and elements 1 and 3 from the second vector.
3. Concatenate two vectors into a <8 x i16>.

### [Slide 223]

The LLVM-IR was generated from this code:

```
// clang-18 -emit-llvm -O1 -g0 -mavx2 -mllvm -force-target-max-vector-interleave=1
void foo(long* restrict a, const long* b) {
#pragma clang loop vectorize(enable) interleave(disable)
  for (long i = 0; i < 2048; i++) {
    a[i] = b[i] < i ? i + b[i] : 0;
  }
}
```

### [Slide 259]

Dynamic Programming Table:

Node	+ <sub>1</sub>	« <sub>1</sub>	4	* <sub>1</sub>	ld <sub>1</sub>	+ <sub>2</sub>	* <sub>2</sub>	8	ld <sub>2</sub>	+ <sub>3</sub>	« <sub>2</sub>	3
Cost	13	1	1	12	6	4	4	1	3	2	1	1
Pattern	$P_9$	$P_0$	$P_{10}$	$P_8$	$P_4$	$P_9$	$P_8$	$P_{10}$	$P_6$	$P_1$	$P_0$	$P_{10}$

Resulting instructions:



---

```
lsl r0, rW, 4
mov r1, 8
madd r2, rY, r1, rX
ldr r3, [r2]
ldr r4, [rZ, rY, lsl #3]
madd r5, r3, r4, r0
```



# Bibliography

- [AC71] FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. URL: <https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf>.
- [AGT89] AV Aho, M Ganapathi, and SWK Tjiang. “Code generation using tree matching and dynamic programming”. In: *TOPLAS* 11.4 (1989), pp. 491–516. URL: <https://dl.acm.org/doi/pdf/10.1145/69558.75700>.
- [BCS97] P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997. URL: <https://www.cs.rice.edu/~keith/Promo/CRPC-TR94517.pdf.gz>.
- [BDB90] A Balachandran, DM Dhamdhere, and S Biswas. “Efficient retargetable code generation using bottom-up tree pattern matching”. In: *Computer Languages* 15.3 (1990), pp. 127–140.
- [Bel73] JR Bell. “Threaded Code”. In: *CACM* 16.6 (1973), pp. 370–372. URL: <https://dl.acm.org/doi/pdf/10.1145/362248.362270>.
- [Bra+13] M Braun et al. “Simple and efficient construction of static single assignment form”. In: *CC*. 2013, pp. 102–122. URL: [https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9\\_6.pdf](https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9_6.pdf).
- [Cla+57] W Clark et al. *The Lincoln TX-2 Computer*. Apr. 1957. URL: [http://www.bitsavers.org/pdf/mit/tx-2/TX-2\\_Papers\\_WJCC\\_57.pdf](http://www.bitsavers.org/pdf/mit/tx-2/TX-2_Papers_WJCC_57.pdf).
- [Cyt+91] R Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *TOPLAS* 13.4 (1991), pp. 451–490. URL: <https://dl.acm.org/doi/pdf/10.1145/115372.115320>.
- [DF84] JW Davidson and CW Fraser. “Code selection through object code optimization”. In: *TOPLAS* 6.4 (1984), pp. 505–526. URL: <https://dl.acm.org/doi/pdf/10.1145/1780.1783>.
- [Die82] PF Dietz. “Maintaining order in a linked list”. In: *STOC*. 1982, pp. 122–127. URL: <https://dl.acm.org/doi/pdf/10.1145/800070.802184>.
- [ECG06] MA Ertl, K Casey, and D Gregg. “Fast and flexible instruction selection with on-demand tree-parsing automata”. In: *PLDI* 41.6 (2006), pp. 52–60.
- [EG03] MA Ertl and D Gregg. “The structure and performance of efficient interpreters”. In: *JILP* 5 (2003), pp. 1–25. URL: <http://www.jilp.org/vol5/v5paper12.pdf>.
- [Ert99] MA Ertl. “Optimal code selection in DAGs”. In: *POPL*. 1999, pp. 242–249. URL: <https://dl.acm.org/doi/pdf/10.1145/292540.292562>.

- [Gar02] K Gargi. “A sparse algorithm for predicated global value numbering”. In: *PLDI*. 2002, pp. 45–56.
- [Geo05] L Georgiadis. “Linear-Time Algorithms for Dominators and Related Problems”. PhD thesis. Princeton University, Nov. 2005. URL: <https://www.cs.princeton.edu/techreports/2005/737.pdf>.
- [GG78] RS Glanville and SL Graham. “A new method for compiler code generation”. In: *POPL*. 1978, pp. 231–254. URL: <https://dl.acm.org/doi/pdf/10.1145/512760.512785>.
- [Hab13] J Haberman. *Parsing C++ is literally undecidable*. 2013. URL: <https://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html>.
- [Hav97] P Havlak. “Nesting of reducible and irreducible loops”. In: *TOPLAS* 19.4 (1997), pp. 557–567. URL: <https://dl.acm.org/doi/pdf/10.1145/262004.262005>.
- [KG08] DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54. URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.
- [KH11] R Karrenberg and S Hack. “Whole-function vectorization”. In: *CGO*. 2011, pp. 141–150.
- [Kud17] J Kuderski. “Dominator Trees and incremental updates that transcend times”. In: *LLVM Dev Meeting*. Oct. 2017. URL: [https://llvm.org/devmtg/2017-10/slides/Kuderski-Dominator\\_Trees.pdf](https://llvm.org/devmtg/2017-10/slides/Kuderski-Dominator_Trees.pdf).
- [LA04] C Lattner and V Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *CGO*. 2004, pp. 75–86. URL: <http://www.llvm.org/pubs/2004-01-30-CGO-LLVM.pdf>.
- [LM69] ES Lowry and CW Medlock. “Object code optimization”. In: *CACM* 12.1 (1969), pp. 13–22. URL: <https://dl.acm.org/doi/pdf/10.1145/362835.362838>.
- [Lop+21] NP Lopes et al. “Alive2: bounded translation validation for LLVM”. In: *PLDI*. 2021, pp. 65–79. URL: [users.cs.utah.edu/~regehr/alive2-pldi21.pdf](https://users.cs.utah.edu/~regehr/alive2-pldi21.pdf).
- [LT79] T Lengauer and RE Tarjan. “A fast algorithm for finding dominators in a flowgraph”. In: *TOPLAS* 1.1 (1979), pp. 121–141. URL: <https://dl.acm.org/doi/pdf/10.1145/357062.357071>.
- [McK65] WM McKeeman. “Peephole optimization”. In: *CACM* 8.7 (1965), pp. 443–444. URL: <https://dl.acm.org/doi/pdf/10.1145/364995.365000>.
- [PM12] M Pharr and WR Mark. “ispc: A SPMD compiler for high-performance CPU programming”. In: *InPar*. 2012, pp. 1–13.
- [Ram99] G Ramalingam. “Identifying loops in almost linear time”. In: *TOPLAS* 21.2 (1999), pp. 175–188. URL: <https://dl.acm.org/doi/pdf/10.1145/316686.316687>.
- [Tar73] R Tarjan. “Testing flow graph reducibility”. In: *STOC*. 1973, pp. 96–107. URL: <https://dl.acm.org/doi/pdf/10.1145/800125.804040>.

- [Tro+21] M Trofin et al. *MLGO: a Machine Learning Guided Compiler Optimizations Framework*. 2021. arXiv: 2101.04808 [cs.PL]. URL: <https://arxiv.org/abs/2101.04808>.
- [Vel03] TL Veldhuizen. *C++ templates are Turing complete*. 2003. URL: <http://port70.net/~nsz/c/c%2B%2B/turing.pdf>.
- [WZ91] MN Wegman and FK Zadeck. “Constant propagation with conditional branches”. In: *TOPLAS* 13.2 (1991), pp. 181–210. URL: <https://dl.acm.org/doi/pdf/10.1145/103135.103136>.