Instruction Selection in LLVM

Table of Contents

---

1. Introduction

Instruction selection is a key phase in the backend of a compiler. It maps the intermediate representation (IR) of a program to machine instructions from a specific instruction set architecture (ISA). In the context of LLVM, instruction selection is modular, target-aware, and designed to maximize code efficiency while preserving correctness.

LLVM's instruction selection is part of the *CodeGen* backend, which includes:

- **Instruction Selection**: Translates LLVM IR to target-specific instructions.
- **Instruction Scheduling**: Reorders instructions to optimize pipeline usage.
- **Register Allocation**: Maps virtual registers to physical ones.

The primary challenge of instruction selection is to find the best target instructions that implement the IR operations, considering the peculiarities of the target ISA. This process is tightly coupled with optimizations that take advantage of instruction-level parallelism, register usage, and addressing modes.

Moreover, LLVM supports multiple instruction selection frameworks, such as the older SelectionDAG-based approach and the newer GlobalISel infrastructure. SelectionDAG represents the program in a form amenable to tree pattern matching and is highly optimized for mature targets. In contrast, GlobalISel provides a more flexible and extensible pipeline designed to support new targets more easily and to improve maintainability.

Instruction selection is not just about correctness—it plays a significant role in performance. Poor instruction selection can lead to bloated binaries and inefficient use of the target machine's features. The effectiveness of this phase directly impacts the quality of the generated machine code, influencing power consumption, execution time, and overall system performance.

Additionally, the interaction of instruction selection with earlier and later stages of the compiler pipeline is significant. For example, earlier optimizations may affect the granularity and shape of IR instructions, which in turn impacts how instruction selection patterns match. Similarly, the choices made during instruction selection constrain the possibilities available to the register allocator and scheduler.

LLVM allows backend developers to define target-specific patterns and cost models, enabling fine-grained control over instruction selection. This flexibility is critical when targeting ISAs like RISC-V, which aim for

simplicity and extensibility. LLVM's modularity makes it a suitable platform for experimenting with novel instruction selection approaches as well.

Structure of the Report

This report is structured to provide both foundational understanding and advanced insights into instruction selection within LLVM, particularly in the context of the RISC-V architecture. Section 2 discusses how ISA design shapes instruction selection strategies, with concrete examples. Sections 3 to 6 examine individual components and mechanisms within LLVM's instruction selector, including peephole optimizations, simplification, matching, and tree-pattern matching. Section 7 presents a survey of instruction selection methodologies relevant to LLVM and RISC-V, exploring both historical and modern approaches. Section 8 investigates the role of AI in instruction selection, while Section 9 discusses future directions and ongoing innovations. The final section compiles a set of references for further reading.

---

2. The Impact of ISA Design on Instruction Selection

The design of the target ISA significantly influences the complexity and strategy of instruction selection. ISAs with rich addressing modes, complex instructions (like x86), or RISC-style simplicity (like RISC-V) impose different constraints and opportunities.

Motivating Example

Consider the following C code:

```
int a = b + c * d;
```

The corresponding LLVM IR might look like:

```
%1 = mul i32 %c, %d
%2 = add i32 %b, %1
```

In RISC-V, which is a load-store architecture with three-address instructions, the selection might yield:

```
mul t0, c, d
add a, b, t0
```

In a more complex ISA (e.g., x86), the multiplication might be fused into an addressing mode.

Design Dimensions That Affect Selection

- **Instruction Set Orthogonality**: Whether instructions can be freely combined.
- **Addressing Modes**: Rich modes may allow folding memory operations.
- **Register Constraints**: Some ISAs (e.g., ARM) have specific registers for certain operations.
- **Immediate Field Widths**: Affect whether constants can be embedded.

Instruction selectors need to model these constraints explicitly. For instance, in TableGen, one must annotate the legality and encoding features of instructions, including their operand classes and their expected behaviors. The selection logic, when generated from these patterns, leverages this metadata to emit valid and optimized sequences.

Different ISAs often lead to divergent strategies in instruction selection. In RISC-V, which favors simplicity and uniformity, instruction selection tends to be straightforward. However, the lack of complex addressing

modes means that the selector has to emit multiple simple instructions where more powerful ISAs could use a single one. This tradeoff emphasizes the importance of good scheduling and peephole optimization to eliminate redundancy introduced during initial selection.

Moreover, RISC-V's modular ISA—where optional extensions such as multiplication or atomic operations are enabled independently—requires instruction selectors to handle capability checks dynamically. LLVM addresses this by generating multiple patterns guarded by predicates that reflect the active sub-target features. This approach allows the compiler to generate correct code across varying configurations of the RISC-V ISA.

In addition to instruction set differences, microarchitectural characteristics such as pipeline depth, register file size, and memory latency also influence the optimality of selected instructions. Some instructions may be correct but introduce latency penalties due to pipeline hazards or cache behavior. Thus, instruction selection is inherently tied to both the abstract ISA and the physical characteristics of the processor.

---

3. Peephole Optimization

Peephole optimizations are small, local transformations applied after or during instruction selection. They aim to improve efficiency without changing program semantics. These optimizations examine a small window (or "peephole") of instructions and look for patterns that can be replaced with more efficient sequences.

Example

Consider:

```
%1 = add i32 %x, 0
```

This can be eliminated entirely, as adding zero is a no-op.

LLVM's `peephole-opt` pass includes many such transformations, such as:

- Constant folding
- Redundant instruction elimination
- Algebraic simplifications

These transformations are crucial for cleaning up inefficient patterns that might result from generic instruction selection strategies. For example, if the selector conservatively generates a load and then a shift to perform multiplication by a power of two, peephole optimization can often recognize the idiom and replace it with a single shift or LEA instruction (in complex ISAs).

Peephole optimizations can be applied:

- Before instruction selection (on IR)
- After instruction selection (on MachineInstrs)

In the LLVM backend, this often happens in the `PostISelDAG` phase, which takes the selected DAG and simplifies instruction sequences. This post-processing step helps reduce register pressure, instruction count, and overall binary size.

Peephole optimizations also serve as a secondary defense mechanism. While more sophisticated passes like loop unrolling and vectorization attempt global optimizations, peephole passes can catch cases that slipped

through, especially after transformations have disturbed previously optimized regions. In some cases, multiple passes may be applied iteratively until a fixed point is reached, ensuring maximal simplification.

These optimizations are not limited to arithmetic simplifications. They may also involve recognizing register-to-register copies, replacing them with moves, or identifying redundant load/store pairs and eliminating them. By operating at the instruction level, peephole passes help close the gap between high-level optimization and low-level code efficiency.

---

4. The Simplifier

The simplifier is a component of LLVM's SelectionDAG infrastructure. It reduces and canonicalizes the DAG (Directed Acyclic Graph) before instruction selection. Its primary purpose is to expose canonical forms and eliminate unnecessary complexity that could hinder effective pattern matching.

Tasks of the Simplifier

- Merge redundant operations
- Normalize instruction forms
- Simplify constant expressions
- Propagate known values and types

Example

```
%1 = xor i32 %x, -1  ; bitwise NOT
```

Can be simplified to a NOT node in the DAG, which might map to a specific RISC-V pseudo instruction or a combination of XOR with -1. This helps reduce the number of patterns required to handle semantically equivalent expressions.

The simplifier improves matching efficiency and avoids redundant pattern definitions. For instance, commutative operations like `add` and `mul` can be reordered consistently so that a single pattern definition matches more cases. This normalization reduces pattern explosion and improves match quality.

In practice, this simplification phase is essential for reducing the search space that the matcher must consider. For example, if a node is expressed in multiple equivalent ways (e.g., `a + b` vs. `b + a`), simplification ensures that only one canonical form reaches the pattern matcher, reducing both compile time and the required number of patterns.

The simplifier can also propagate known constants and values, eliminating entire branches of computation. For instance, a multiply-by-zero operation can be collapsed to a constant, avoiding any code generation for that sub-tree and reducing register pressure and instruction count.

In SelectionDAG, simplification occurs both during DAG construction and just before pattern matching. This dual-phase design ensures that DAGs remain manageable and well-formed throughout their lifetime. It also interacts closely with legalization, as certain illegal DAG nodes must be transformed into legal forms before simplification and selection proceed.

---

5. The Matcher

LLVM uses pattern matching to map IR or DAG nodes to machine instructions. This is typically done via TableGen-generated matchers, which encode instruction selection rules declaratively.

Key Concepts

- **Patterns**: Specified in `.td` files
- **Operands**: Match IR values or constants
- **Constraints**: Ensure legality on the target

Example

In TableGen (RISC-V):

```
def : Pat<(add GPR:$rs1, GPR:$rs2), (ADD GPR:$rs1, GPR:$rs2)>;
```

This says: match an IR `add` node with two general-purpose registers and map it to the `ADD` instruction.

The matcher traverses the DAG in bottom-up order, choosing the best match according to cost models. It aims to find the lowest-cost instruction sequence that covers the input DAG. The selection process can be customized via predicates and target-specific cost functions.

TableGen matchers rely heavily on pattern specificity and order. The more specific a pattern is, the earlier it is considered. This allows backends to express highly optimized instruction sequences for common idioms, such as loop counters, memory offsets, or conditional branches, without overriding more general fallback rules.

When multiple matches apply, LLVM uses cost-based heuristics to select the most efficient one. These costs can be based on instruction latency, size, or other target-specific metrics. Therefore, the matcher is not just syntactic—it embodies performance trade-offs tuned for each architecture.

Matchers also support custom C++ logic to handle cases not expressible in declarative patterns. For example, matching variable-length vector instructions or predicated operations might require control-flow-sensitive information, which is beyond the scope of pattern matching alone.

---

6. Selection via Tree-Pattern Matching

LLVM's SelectionDAG approach uses tree-pattern matching to identify suitable instruction sequences. This technique is based on classical compiler algorithms, where expression trees are covered using instruction templates.

DAG Representation

IR instructions are converted into DAGs, where:

- Nodes represent operations (add, mul, etc.)
- Edges represent data dependencies

SelectionDAG differs from basic trees in that it supports shared subexpressions and side-effect modeling. DAGs allow more accurate modeling of real programs and facilitate reuse of computation across instruction patterns.

Matching Process

- **Combine**: Group multiple IR nodes into one machine instruction
- **Select**: Use pattern definitions to emit instructions

Example

```
int x = (a + b) << 2;
```

Might produce DAG nodes:

- `add a, b`
- `shl result, 2`

A combined pattern can match this to:

```
ADD t0, a, b
SLLI x, t0, 2
```

If the target has a fused `LEA`-like instruction, the matcher may emit that instead.

This technique is inspired by classical tree-covering algorithms and is highly effective for RISC-style targets. The matcher reduces complex trees into minimal-cost tilings using dynamic programming, ensuring both correctness and optimality under local constraints.

The main challenge with tree-pattern matching is expressiveness. While it's efficient for regular ISAs, it struggles with irregular instructions or semantic conditions. For these cases, LLVM supports custom C++ matchers or post-selection transformations to ensure correct code generation.

Tree-pattern matching also integrates tightly with instruction legalization and register allocation. The selected instructions must satisfy operand constraints and be legal on the target. If not, additional transformations or fallback patterns are triggered to recover valid code sequences.

LLVM developers can augment the matcher with target-specific patterns for idioms such as arithmetic simplifications, memory addressing, or loop increment expressions. These patterns help close the gap between IR semantics and efficient instruction sequences on specific architectures.

---

7. Survey of Instruction Selection Methodologies

Instruction selection has undergone significant evolution over the decades. In the context of LLVM and RISC-V, a range of methodologies have been explored, each with different trade-offs in terms of flexibility, performance, and maintainability. This section provides a mini-survey of the major instruction selection methodologies used or studied in LLVM, enriched with small examples to illustrate their operation.

7.1 Macro Expansion

Macro expansion is the simplest method for instruction selection, where each intermediate representation (IR) operation maps directly to a fixed machine instruction or sequence. This approach ignores context and global structure, often resulting in inefficient code.

Example:

```
%1 = add i32 %a, %b
```

Maps directly to:

```
ADD t0, a, b
```

This method is easy to implement but cannot exploit complex instruction patterns or eliminate redundant computations.

7.2 Tree Pattern Matching

Tree pattern matching involves representing IR operations as trees and covering these trees using pre-defined instruction patterns. LLVM's SelectionDAG uses this method extensively. Patterns are expressed in TableGen and are matched using dynamic programming to select the lowest-cost instruction tiling.

Example:

```
int x = (a + b) << 2;
```

Corresponding LLVM IR:

```
%1 = add i32 %a, %b
%2 = shl i32 %1, 2
```

Tree pattern matcher recognizes this as a sequence and selects:

```
ADD t0, a, b
SLLI x, t0, 2
```

7.3 DAG-Based Instruction Selection with Simplification

SelectionDAG generalizes tree matching to DAGs, which better model SSA-based IR and allow common subexpression elimination. The DAG is simplified before selection to canonicalize and optimize nodes.

Example:

```
%1 = xor i32 %a, -1  ; Bitwise NOT
```

Simplifier converts this to a NOT node, enabling pattern selection:

```
NOT t0, a
```

7.4 Global Instruction Selection (GlobalISel)

GlobalISel is LLVM's modern instruction selection framework designed to overcome limitations of SelectionDAG. It operates directly on Machine IR (MIR) and separates instruction legalization, selection, and register allocation phases.

Example:

```
%1 = sub i32 %x, %y
```

GlobalISel performs:

```
SUB a0, x, y
```

7.5 Bottom-Up Rewriting Systems

Bottom-up rewriting systems such as BURG define a set of rewrite rules and use dynamic programming to select minimal-cost instruction sequences. Though not directly used in LLVM, its concepts influence the TableGen matcher.

Example: IR Tree:

```
     +
    / \
   *   2
  / \
 a   3
```

Rewritten into:

```
MUL t0, a, 3
ADDI x, t0, 2
```

7.6 Heuristic and Cost-Based Selection

Heuristic models guide selection when multiple patterns match. LLVM uses target-specific cost models to pick instructions that optimize for size, latency, or power consumption.

Example:

```
%1 = add i32 %x, 1
```

Two options:

```
ADDI x, x, 1
OR x, x, 1
```

Cost models guide the optimal choice based on context.

7.7 Instruction Selection via Machine Learning

Recent methods explore AI-driven selection using models trained on large datasets. These can predict sequences of instructions based on IR input, using techniques like supervised learning or reinforcement learning.

Example: Given:

```
%1 = and i32 %a, 0xFF
```

An ML model may emit:

```
ZEXT x, a
```

If supported by the target.

These methodologies illustrate the rich design space in instruction selection, particularly in LLVM. For RISC-V, the emphasis is on minimalism, which makes pattern matching and cost-aware strategies especially important.

---

8. AI and Instruction Selection

Recent research explores AI-based techniques to improve or replace traditional instruction selection. These approaches seek to address the growing complexity of instruction sets and optimization opportunities that are difficult to encode manually.

Motivations

- Handle complex ISAs with large pattern spaces
- Improve adaptability across architectures
- Explore trade-offs in instruction selection quality

AI techniques can generalize from past compiler outputs or performance data to learn effective mappings from IR to machine instructions. This is especially valuable for domain-specific architectures or systems where hand-tuned instruction selectors are not feasible.

Techniques

- **Supervised Learning**: Models are trained on pairs of IR input and correct machine instruction output. This approach mimics traditional instruction selectors but learns to generalize beyond explicitly encoded patterns.
- **Reinforcement Learning**: The selector learns by interacting with a simulated environment, receiving rewards based on the performance of emitted code (e.g., execution time, size).
- **Graph Neural Networks (GNNs)**: These are used to model the IR as a graph and reason about dependencies and structures in a way that generalizes well across programs.

Example

Given an IR operation:

```
%1 = mul i32 %a, 4
```

A learned model may suggest a shift instead:

```
SLLI x, a, 2
```

This transformation is common in hand-optimized code and can be learned from data.

Challenges

- Ensuring semantic correctness
- Handling edge cases and target-specific constraints
- Integration with LLVM’s deterministic and maintainable infrastructure

LLVM researchers are investigating hybrid models that combine traditional rule-based selection with AI-guided heuristics. Such approaches retain the reliability of pattern-based systems while gaining flexibility from learned models.

While still experimental, AI-driven instruction selection holds promise for optimizing compilation pipelines and improving code generation in areas where expert-tuned rules are infeasible or overly rigid.

---

9. The Future of Instruction Selection

LLVM's current instruction selection frameworks—SelectionDAG and GlobalISel—represent different points in a long trajectory of evolution. As the diversity of hardware targets continues to increase, and the demand for high performance and energy efficiency grows, instruction selection strategies are expected to become more dynamic, intelligent, and context-sensitive.

Global Instruction Selection (GlobalISel)

GlobalISel is one of the major efforts in LLVM aimed at modernizing instruction selection. It provides better modularity and separation of concerns, making it easier to develop and maintain backend code. Its phases (legalization, selection, and register bank assignment) allow clearer abstraction boundaries and facilitate customization per target.

In the future, GlobalISel may become the default infrastructure, phasing out SelectionDAG in most targets. Its MIR-centric approach also makes it more suitable for machine learning integration and JIT use cases.

Domain-Specific Instruction Selection

With the rise of domain-specific accelerators (e.g., AI/ML chips, DSPs, quantum co-processors), instruction selectors will need to support highly irregular ISAs with specialized semantics. This may require instruction selectors that are either programmable or adaptive to dynamic constraints.

LLVM is already experimenting with domain-specific idiom matching, where backends include custom patterns to recognize high-level IR forms like matrix multiplications or convolution layers. These patterns are then lowered to specialized instructions or intrinsic calls.

Feedback-Guided and Profile-Driven Selection

Instruction selection may increasingly rely on runtime feedback or offline profiling. Profile-guided optimization (PGO) already informs inlining and loop unrolling decisions; similar data could influence instruction choice. For example, branch frequency data could prioritize compact branch encodings or fuse instruction sequences in hot paths.

LLVM’s MachineFunction and MachineLoop infrastructures are well-positioned to support feedback-directed instruction selection. Integration of execution traces could enable dynamic recompilation strategies in managed runtime environments.

Integration with Machine Learning

As discussed in Section 8, AI-based instruction selection is a promising frontier. Future compilers may integrate learned cost models or use end-to-end pipelines where instruction selection is informed by

historical compilation outcomes. This opens the door to autotuning and architecture-specific specializations that adapt over time.

However, this transition requires addressing challenges of correctness, explainability, and integration into deterministic compiler pipelines. Compiler-as-a-service platforms may lead this adoption due to their scale and ability to train models across millions of compilation instances.

In summary, the future of instruction selection in LLVM lies at the intersection of architectural diversity, compiler modularity, and adaptive intelligence. Continued advances in static analysis, hardware modeling, and AI will play a pivotal role in shaping this evolution.

---

10. References

- Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.
- LLVM Project Documentation: https://llvm.org/docs/
- LLVM TableGen Tutorial: https://llvm.org/docs/TableGen/index.html
- LLVM GlobalISel Infrastructure: https://llvm.org/docs/GlobalISel/index.html
- RISC-V Unprivileged ISA Specification: https://riscv.org/specifications/
- Nobre, A., Pereira, F. M. Q., & Sarkar, V. (2020). "Machine Learning-Based Instruction Selection." *ACM Transactions on Architecture and Code Optimization*.
- Davidson, J. W., & Fraser, C. W. (1984). "The Design and Application of a Retargetable Peephole Optimizer." *ACM Transactions on Programming Languages and Systems*.
- Proebsting, T. A. (1995). "Optimizing an ANSI C Interpreter with Superoperators." *POPL '95 Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
- LLVM Developer Policy: https://llvm.org/docs/DeveloperPolicy.html

---