Lecture 2: Instruction Selection

CESE4085 Modern Computer Architecture Course
Part 2, L7.2
Carlo Galuzzi

TU Delft

LECTURE CONTENTS

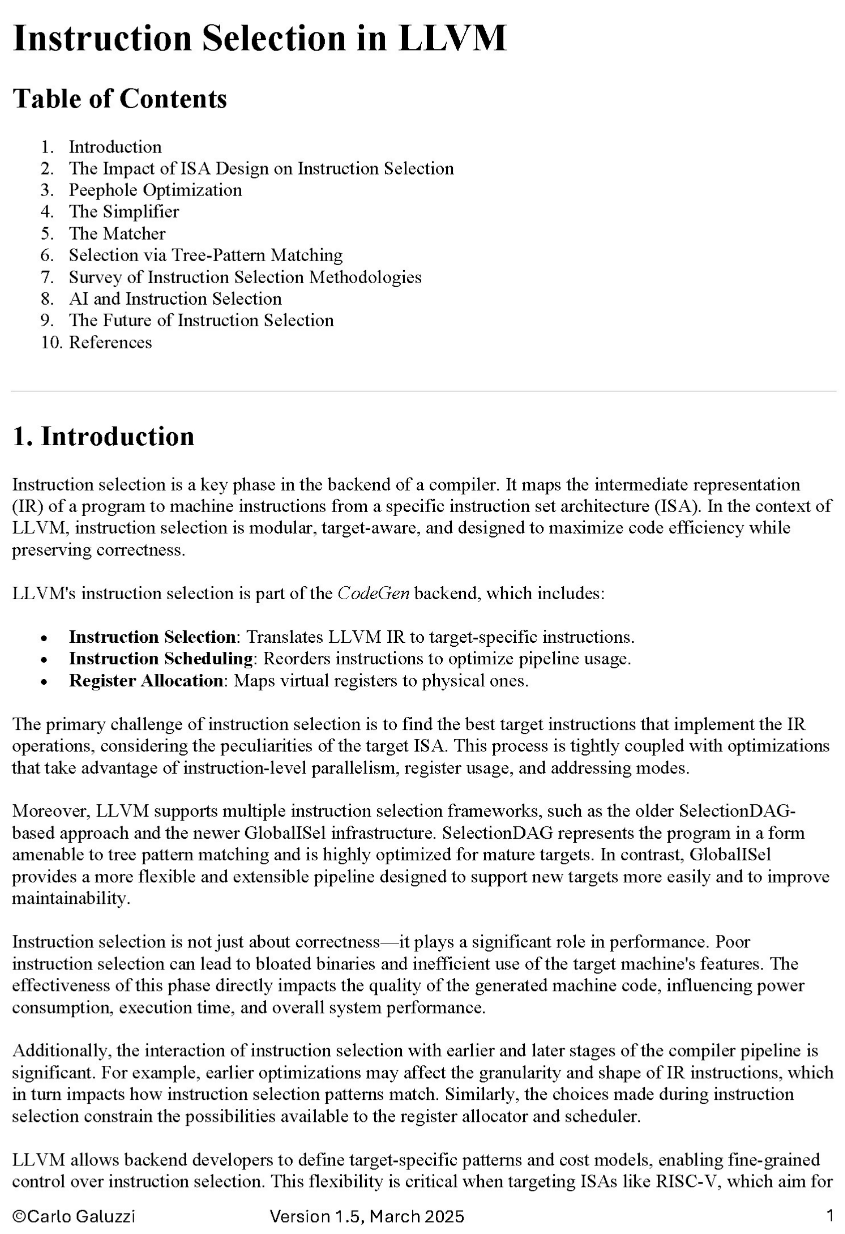
- What is Instruction Selection?
- LLVM Code Generation
- Instruction Selection and the Target
- Pattern Matching and Optimization
- Approaches to Instruction Selection
- Case Study: RISC-V
- Modern and Future Techniques

LEARNING GOALS:

Instruction selection is a bridge between high-level logic and low-level execution, and it touches on both architecture and compiler internals. We'll look at how LLVM performs this task, what design decisions matter. By the end of this lecture, among other goals, you will be able to:

- Understand the role of instruction selection in a compiler
- Analyze how processor design affects instruction selection
- Appreciate the importance of simplification and optimization
- Compare traditional and modern methods
- Apply this knowledge in practical settings

EXTRA READING MATERIAL IN BRIGHSPACE



Instruction Selection in LLVM

Table of Contents

1. Introduction
2. The Impact of ISA Design on Instruction Selection
3. Peephole Optimization
4. The Simplifier
5. The Matcher
6. Selection via Tree-Pattern Matching
7. Survey of Instruction Selection Methodologies
8. AI and Instruction Selection
9. The Future of Instruction Selection
10. References

---

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.

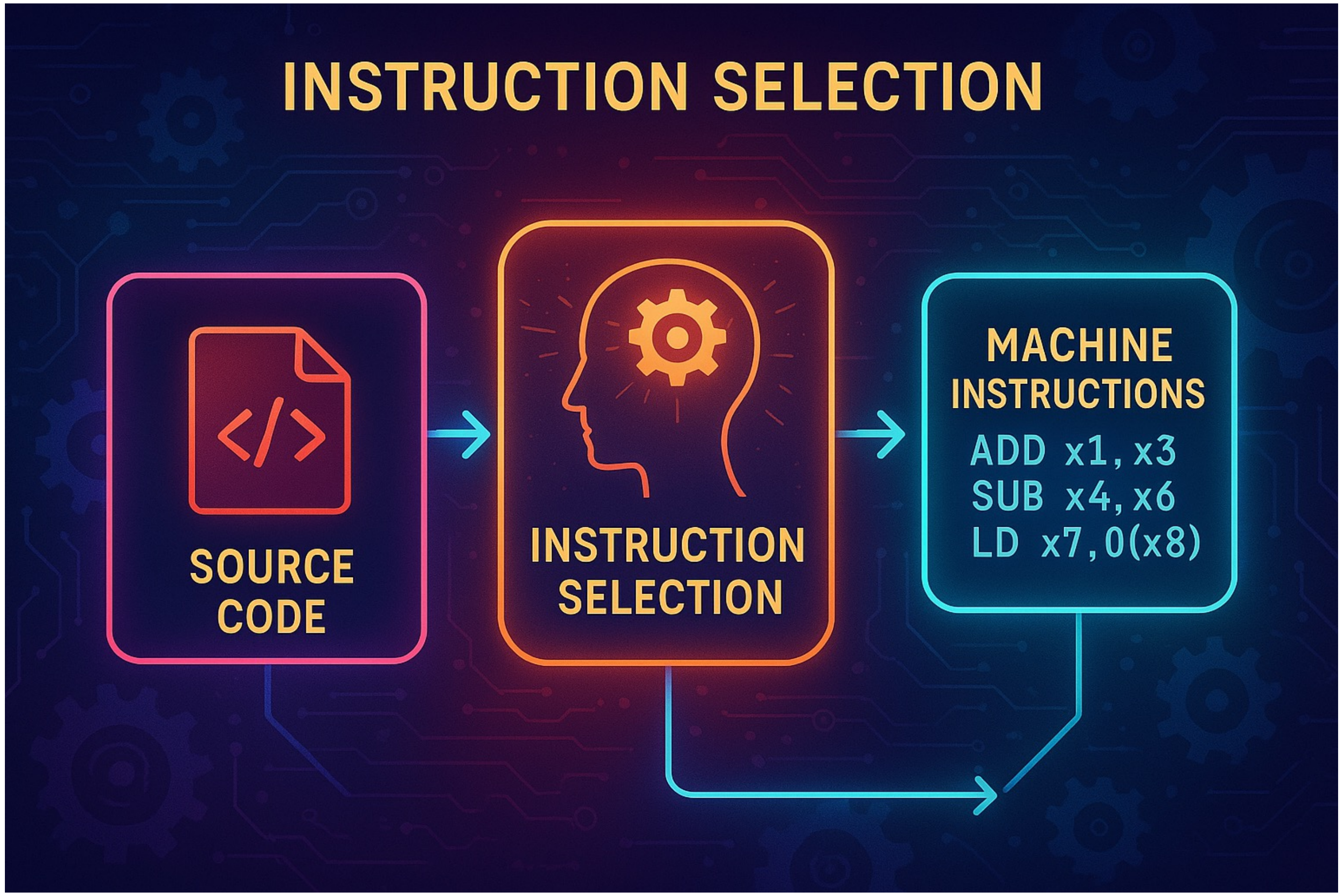
Moreover, LLVM supports multiple instruction selection frameworks, such as the older SelectionDAG-based approach and the newer GlobalISel infrastructure. SelectionDAG represents the program in a form amenable to tree pattern matching and is highly optimized for mature targets. In contrast, GlobalISel 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

Instruction Selection



(AI Generated)

WHAT IS INSTRUCTION SELECTION?

When your code is compiled, it doesn't magically become executable. Somewhere in the process, the compiler needs to take abstract operations—like adding two integers—and turn them into actual CPU instructions.

These instructions have to follow the rules of the hardware while still doing what your code intended.

- IR (Intermediate Representation) is portable across targets. For example: `%1 = add i32 %a, %b`

- This needs to become a valid machine instruction. On RISC-V:

```
add t0, a, b
```

The goal is to match meaning while using instructions efficiently and legally.

LLVM CODE GENERATION

LLVM is designed as a modular compiler framework. It breaks the job of compiling into three major chunks, each responsible for a specific part of the process.

Instruction selection sits in the backend and is one of the final, most hardware-specific transformations.

1. **Front-End** – Parses source code and outputs LLVM IR.
2. **Mid-End** – Performs optimizations on IR.
3. **Back-End** – Translates IR to machine instructions.

Instruction selection takes over once all the general-purpose optimizations are done and the compiler knows it’s time to talk directly to the hardware.

INSTRUCTION SELECTION AND THE TARGET

Every processor has its quirks. Some are minimalist and require very explicit code, while others have fancy features that allow more complex instructions.

The compiler has to be aware of what the target machine can do and adapt its output accordingly.

- RISC-V is straightforward: simple, fixed-length instructions.
- x86 is packed with powerful, flexible instructions.

Example:

- On RISC-V, memory and arithmetic are done separately:

```
lw t1, 0(sp)
add t0, t1, t2
```

- On x86, the memory access might be fused:

```
add eax, [ebx]
```

MAPPING IR TO MACHINE CODE

The compiler doesn't just copy IR to machine code—it has to understand what each IR instruction is trying to do and then choose the best way to express that with real hardware instructions. This means translating abstract logic into legal, efficient operations that your CPU understands.

C example:

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

LLVM IR:

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

RISC-V assembly:

```
add t0, a, b
slli x, t0, 2
```

ISA DESIGN AND ITS IMPACT

Instruction selection depends heavily on what the target architecture allows. Some ISAs are very regular and simple, while others are complex and packed with features.

This changes how many steps the compiler needs to take and whether it can use shortcuts.

- RISC-V uses simple, explicit operations.
- x86 can combine operations with memory access and has many encodings.

Example:

```
arr[i] += 4;
```

Could turn into:

- RISC-V: multiple steps (load, add, store)
- x86: one instruction with a memory operand

MOTIVATING EXAMPLE: C TO RISC-V

Let’s go from source code to machine code to see what instruction selection looks like in action. This helps make things concrete by showing how abstract logic is turned into real instructions.

C:

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

IR:

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

RISC-V:

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

The compiler breaks the computation into small parts and picks instructions that match each step.

TREE AND DAG REPRESENTATIONS

Inside the compiler, computations are represented not as plain text but as structured graphs. This helps the compiler understand dependencies, optimize, and match patterns more easily. The two main representations are **trees** and **DAGs**.

- A **tree** is simple and works well for basic expressions.
- A **DAG** (Directed Acyclic Graph) can represent shared subcomputations.

Example of a tree:

```
  +
 / \
a   b
```

Example of a DAG (shared computation):

```
  +
 / \
a   b
 \ /
  +
```

DAGs help avoid recomputing the same thing twice.

SelectionDAG IN LLVM

As LLVM starts converting IR to machine code, it uses an intermediate structure called the **SelectionDAG**.

This is a data structure that helps the compiler keep track of all the computations and how they depend on one another, making it easier to match groups of instructions to efficient machine code.

- A **SelectionDAG** is a Directed Acyclic Graph used to represent operations.
- Each node represents an operation, such as `add`, `mul`, or `load`.
- Edges represent dependencies: which values are used by which operations.

Example: If you compute `(a + b) * (a + b)`, the DAG avoids recomputing `a + b` by sharing that subexpression across both inputs to `mul`.

WHY USE DAGS INSTEAD OF TREES?

Simple expression trees are easy to build and understand, but they have limitations when you need to reuse values or represent more realistic program behavior.

DAGs help the compiler avoid repeating work and produce cleaner code.

- Trees can’t represent shared values, so repeated expressions are computed multiple times.
- DAGs can share results of computations, reducing redundant code generation.

Example:

```
x = (a + b);
y = (a + b) * c;
```

In a tree: `a + b` might be recomputed. In a DAG: it’s computed once and reused.

TableGen AND DSLs IN LLVM

To define how IR should be mapped to machine instructions, LLVM uses a custom domain-specific language (DSL) called **TableGen**.

This system lets developers describe instructions, registers, and pattern rules without writing all the logic manually in C++.

- TableGen files (.td) are declarative and concise.
- They define what patterns in IR look like and how to translate them.
- The LLVM build system uses **TableGen** to generate efficient matching code.

This approach simplifies backend development and improves code consistency across targets.

STRUCTURE OF A TableGen PATTERN

TableGen patterns follow a structure that makes them easy to match. A pattern usually has a left-hand side (LHS) to match IR and a right-hand side (RHS) to produce a machine instruction.

Example:

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

- LHS: IR pattern using pseudo-registers (e.g., `GPR:$rs1`).
- RHS: Target instruction to emit.

You can also match more complex expressions like shifts, multiplies, or constants in the LHS.

MATCHING CONSTANTS AND FOLDS

LLVM allows matching constant values in patterns, which helps fold operations into compact forms.

Example:

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

Can be eliminated because adding zero has no effect.
TableGen patterns can include constants:

```
def : Pat<(add GPR:$rs1, 0), (COPY GPR:$rs1)>;
```

This replaces an add with a copy when the second operand is zero.

COST MODELS AND INSTRUCTION SELECTION

LLVM uses cost models to help choose between multiple valid instruction patterns.

These costs reflect things like instruction latency, size, or power usage.

- Patterns can be annotated with cost metadata.
- Targets may define their own cost functions.
- Lower-cost instructions are preferred during matching.

This is essential when selecting between alternative instructions that produce the same result.

MATCHING INSTRUCTION SEQUENCES

Sometimes, LLVM matches not just single operations, but small sequences of them. This is useful for optimization and instruction fusion.

Example:

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

May be combined as [(1)] :

```
SLLI_ADD x, a, b, 1 ; if supported
```

This reduces instruction count and improves throughput.

[(1)]SLLI_ADD is a hypothetical fused instruction that combines a **shift left logical immediate** (SLLI) and an **addition** (ADD) into a single operation—if supported by the target ISA. It's not an actual RISC-V instruction.

CANONICALIZATION AND DAG NORMALIZATION

Before pattern matching, LLVM tries to rewrite expressions into a canonical form. This means different but equivalent expressions are represented in a consistent way. For example:

- `add x, y` and `add y, x` become the same node.
- Constants are moved to the right side:

```
add 3, x → add x, 3.
```

This reduces the number of patterns needed and increases the chances of a match.

DAG TRANSFORMATIONS FOR OPTIMIZATION

DAG nodes can be transformed before matching to optimize the code.

These transformations can simplify operations, remove redundancy, or prepare for better matches.

Examples:

- `x * 2 → x << 1`
- `x + 0 → x`
- `and x, 255 → zext x`

These rules are often built into LLVM as general simplification passes.

PEEPHOLE OPTIMIZATION (1)

Once instructions are selected, there’s still room for improvement. Peephole optimization scans short sequences of instructions to simplify or eliminate redundancy.

- Works locally on small windows of code ("peepholes").
- Detects patterns like no-op moves, unnecessary loads/stores, or constant operations.
- Can be applied iteratively until no changes are found (reaching a fixed point).

Example:

```
add x, x, 0    ; unnecessary
```

Can be removed without affecting the program.

PEEPHOLE OPTIMIZATION (2)

Peephole optimizers often rely on simple rules, but their impact can be significant. More examples:

- Two consecutive moves:

```
mov a, b
mov b, a
```

May cancel each other out.

- Constant propagation:

```
mov a, 4
add a, 5
```

Can be folded into: `mov a, 9`

In embedded or real-time systems, such savings can reduce power, code size, and cache pressure.

LEGALIZATION AND LOWERING PHASES

In LLVM, not all IR instructions are directly legal on a given target.

Legalization rewrites these into simpler, legal parts.

- **Type legalization**: Breaks large types (e.g., `i128`) [1] into smaller parts.
- **Operation legalization**: Replaces unsupported operations with legal ones.
- **Custom lowering**: Some targets provide custom implementations for complex operations.

This ensures the DAG only contains instructions that can be matched and emitted.

[1]LLVM will split an i128 operation into operations on the lower 64 bits and the upper 64 bits separately. It then manages carry or borrow bits across the two halves to ensure correctness. This lets the compiler emulate larger operations using supported instructions.

GLOBALISEL OVERVIEW

GlobalISel is a modern instruction selector in LLVM that avoids building a **SelectionDAG**.

It works directly on Machine IR and aims to be more modular and target-independent.

- Built for modern architectures with complex needs.
- Offers clearer phases: selection, legalization, register assignment.
- Easier to extend and debug compared to SelectionDAG.

GlobalISel is LLVM’s direction for future backend development.

GlobalISel INSTRUCTION SELECTION PHASE

The first phase in GlobalISel is selection. Here, IR operations are replaced with generic machine instructions.

- Pattern matching is performed using C++ matchers or auto-generated code.
- Legal types and operations are assumed at this stage.

Example:

```
%1 = add i32 %a, %b → becomes → G_ADD %a, %b
```

This generic instruction will be legalized in the next step.

WHY INSTRUCTION SELECTION MATTERS IN PRACTICE (1)

- Once we've optimized the IR, we still need to make real instructions. This step can make or break performance.

- Instruction selection isn’t just about correctness—it’s about taking advantage of the target’s capabilities without wasting cycles or registers.

- Think of writing a shopping list. You can write “fruit” or you can write “three bananas and two apples.” Instruction selection is like being specific in a way that fits what the store offers. You don't want to ask for something the store (hardware) doesn’t sell ☺.

WHY INSTRUCTION SELECTION MATTERS IN PRACTICE (2)

Example:

```
int x = (a + b) << 1;
```

This can be compiled into:

```
add t0, a, b
slli x, t0, 1
```

But what if your processor has a single instruction that does add+shift? The selector needs to notice that.

A TALE OF TWO COMPILERS

Imagine compiling the same program using two different instruction selectors. One uses only safe, simple rules. The other is smart: it looks ahead, folds operations, and picks efficient instructions.

Both compilers start with this IR:

```
%1 = mul i32 %a, 2
%2 = add i32 %1, 1
```

- Compiler A generates:

```
mul t0, a, 2
addi x, t0, 1
```

- Compiler B knows `a * 2` is a shift:

```
slli t0, a, 1
addi x, t0, 1
```

Same program, faster code. That’s the power of good instruction selection.

REAL HARDWARE, REAL DECISIONS

Let’s say we’re targeting a RISC-V core. It doesn’t support complex memory operations or fused multiply-add. So instruction selection has to break everything into smaller, supported steps.

```
int y = arr[i] + 3;
```

Gets lowered to:

```
slli t0, i, 2 ; multiply i by 4 for word offset
add t1, t0, arr
lw t2, 0(t1)
addi y, t2, 3
```

Each step reflects both a hardware limitation and a compiler decision. A different ISA might do this in two instructions.

LOCAL VS. GLOBAL VIEW

Instruction selection often works within a single basic block. But what if we looked across blocks? Suppose we have:

```
if (x > 0) y = x + 1;
```

Locally, we just do the addition in the `if` block. But globally, we might notice that `x + 1` is needed elsewhere too, and hoist it out.

This can reduce code duplication and improve register usage. It’s like packing for a trip—planning ahead can save space and effort.

PRESSURE FROM REGISTERS

Suppose you've got just a few registers. A complex instruction like a multiply-accumulate may require three operands. If you're low on registers, that could force a spill to memory.

So, the selector might prefer two smaller instructions using fewer temporaries. This is a judgment call: do we want speed or less register pressure?

Example:

- Prefer `add + mul` separately when registers are tight.
- Prefer `madd` when registers are plentiful and performance matters.

CHOOSING BETWEEN EQUALS

Sometimes, two instruction sequences do the same thing.
How do we choose?

Example:
x = a + b + c;

You can do:

```
add t0, a, b
add x, t0, c
```

Or:

```
add t0, b, c
add x, a, t0
```

Both are valid. But what if `a` is already in the result register?
We might want the second version to save a move.

Instruction selection isn't always about math—it's about context.

FITTING THE PUZZLE

- Instruction selection is like playing Tetris.
- The pieces (IR operations) come in, and you need to fit them together efficiently.
- You can choose different blocks (instructions), rotate them (reorder), or delay their placement (scheduling). But once they land (emit), you’re locked in.
- **Great selectors anticipate the next block and leave room for future operations.**

MEMORY AND ALIASING

- When your program accesses memory, the compiler has to be careful.
- What if two pointers refer to the same location? Then reordering or combining loads and stores could cause bugs.
- So, in ambiguous cases, instruction selection plays it safe. It avoids moving memory instructions and chooses simpler, isolated operations.
- If aliasing is ruled out (e.g., `restrict` in C), the selector can be more aggressive and emit fewer, smarter instructions.

THINKING LIKE A CODE GENERATOR

Imagine being a code generator looking at this IR:

```
%1 = load i32, i32* %p
%2 = add i32 %1, 4
store i32 %2, i32* %q
```

Should you do:

```
lw t0, 0(p)
addi t1, t0, 4
sw t1, 0(q)
```

Or fuse it?

```
lw t0, 0(p)
addi t0, t0, 4
sw t0, 0(q)
```

Fusing saves a register. That's a small win, but in loops, it adds up.

Instruction selection thrives on small wins!

CLOSING THE LOOP

- Once instructions are selected, they go on to scheduling, register allocation, and eventually get turned into binary.
- But if the selection was poor, no amount of scheduling can fix it.
- That’s why instruction selection isn’t just an early backend step—it’s the moment when ideas become actions.
- Every decision here echoes down the pipeline. It’s not glamorous. But it’s where your code comes to life.

MATCHING PATTERNS IN REAL LIFE

Think of a mechanic recognizing car trouble by the sound of the engine. Compilers do something similar—they match known patterns in code and apply fixes (or optimizations).

If the compiler sees:

```
%1 = mul i32 %x, 2
```

It should think: “Aha! That’s a shift!” and emit:

```
slli t0, x, 1
```

But if the value is 3, the compiler might just use `addi + slli` to fold it. Matching is about recognition and using the cheapest path forward.

SMART FOLDING OF INSTRUCTIONS

Constant folding isn't just for early optimizations. Instruction selection can fold too, choosing simpler instructions when constants are involved.

Example:

```
x   = (a << 2) + 5;
```

- Recognize that `<< 2` is a multiply.
- Recognize that `+ 5` is an immediate.

On RISC-V:

```
slli t0, a, 2
addi x, t0, 5
```

Or maybe all at once with an address calculation if used in a load.

KEEPING IT LEGAL

Not every idea is allowed. Some combinations of instructions aren't legal on the hardware. Instruction selection has to make sure everything it emits is valid. For instance:

- You can't do add with two memory operands on most RISC architectures.
- Immediate values might be too large for an `addi`, so they need to be split.

Example:

```
x = y + 100000;
```

Might become:

```
lui t0, 1
addi x, y, t0
```

Legal code, even if it takes more than one instruction.

MAKING ROOM FOR LOOPS

Loops are hot spots. Instruction selection can help them run faster by minimizing instruction count and reducing register use.
Suppose you have:

```
for (int i = 0; i < n; i++) sum += arr[i];
```

Selectors can:

- Use offsets smartly.
- Preload values.
- Choose instructions that reduce stalls (like `addi` over `mul`).

Little tweaks here save time every iteration!

BRANCHES AND CONDITIONS

Conditionals often involve comparison followed by a jump.

Instruction selection decides whether to separate these or fuse them.

Compare:

```
slt t0, a, b
bnez t0, label
```

vs.

```
blt a, b, label
```

If the target has a fused conditional branch, using it saves space and time.

WHEN NOT TO OPTIMIZE

Sometimes, the smartest choice is to leave things alone.

Over-optimizing can make debugging harder or blow up code size.
Instruction selectors might avoid fusing if:

- It prevents good scheduling later.
- It introduces register pressure.
- The gain is negligible in cold code paths.

Optimization is about tradeoffs, not perfection.

SIMPLER ISN’T ALWAYS SLOWER

There’s a myth that fewer instructions always means faster code.

But a simple, regular sequence might be easier for the CPU to pipeline or parallelize.

Example:

```
slli t0, a, 1
add x, t0, b
```

May be faster than a fancy fused op if it aligns better with scheduling. Sometimes, predictability beats cleverness.

FROM SELECTION TO SCHEDULING

Once instruction selection finishes, the baton passes to instruction scheduling.

But the choices made here shape the rest.

Good selection:

- Leaves flexible register usage.
- Avoids unnecessary dependencies.
- Prepares the ground for reordering.

It’s like laying out furniture before decorating—get it wrong and everything feels cramped.

SELECTION PITFALLS AND GOTCHAS

It’s easy to pick the wrong pattern if you don’t check the context. Some mistakes include:

- Matching a pattern without checking alignment.
- Using instructions that need special privilege (e.g., system-level).
- Emitting operations without checking legality.

Robust instruction selection always verifies before it selects.

PLATFORM-SPECIFIC TRICKS

Each architecture has quirks. Knowing them helps selectors make smart choices.

Example (RISC-V):

- Loads/stores must be aligned.
- lui + addi is the idiom for large constants.

Example (x86):

- Use `lea` for arithmetic!

```
lea eax, [eax+4*ebx+8]
```

This does multiply, add, and offset in one go. Instruction selection can find this pattern.

DEBUG-FRIENDLY SELECTION (1)

Sometimes, the goal of compilation isn’t peak performance—it’s clarity and debuggability.

During development, instruction selection can be adjusted to produce more transparent, one-to-one mappings from IR to machine code.

- Emit simpler instructions that directly reflect the IR.
- Avoid fused or overly optimized patterns that obscure source code meaning.
- Retain temporary variables in registers rather than collapsing them.

DEBUG-FRIENDLY SELECTION (2)

This approach makes it easier for tools like GDB (GNU Debugger) or performance profilers to relate assembly code to the original source.

Example:

```
x = a + b;
```

Can remain:

```
add t0, a, b
```

Instead of fusing it into a more complex form that might save one instruction but confuse debugging.

ADAPTING TO EVOLVING ARCHITECTURES

Hardware changes. New instructions appear, old ones are deprecated, and microarchitectural features evolve. Instruction selectors must keep up.

- When a new instruction (like `zext.h` in RISC-V) is introduced, new patterns must be added to use it.
- When an instruction becomes inefficient due to pipeline changes, its cost must be updated.
- Cross-generation compatibility sometimes requires fallback rules.

Good instruction selection adapts quickly, balancing new optimizations with stability.

MANAGING PATTERN EXPLOSION

As an ISA grows, the number of possible instruction patterns increases rapidly.

This “pattern explosion” can overwhelm the selector.

- Similar patterns may overlap and cause ambiguity.
- Some patterns may rarely match in real programs.
- Larger pattern sets slow down matching and increase compile time.

A curated, measured set of high-impact patterns leads to faster, more maintainable compilers.

INSTRUCTION SELECTION FOR VLIW AND DSPS

In **Very Long Instruction Word** (VLIW) and **Digital Signal Processors** (DSPs), instruction selection plays an even more critical role.

- **VLIW** targets often bundle multiple operations together.
- **DSPs** require fine-tuned instruction placement to exploit parallelism.

Instruction selectors for these architectures must:

- Identify independent operations early.
- Match specialized multiply-accumulate or saturation patterns.
- Coordinate with the scheduler to fill instruction slots efficiently.

EMBEDDED SYSTEMS AND SELECTION PRIORITIES

In embedded systems, performance is important—but code size, power consumption, and memory usage may be more critical.

- Smaller instructions are preferred, even if they're slower.
- Selector may choose multi-instruction sequences that use fewer registers.

Example: Instead of a fused multiply-add, an embedded selector might prefer:

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

Because it's easier to spill `t0` if needed.

LOW POWER OPTIMIZATION AND CODE SELECTION

Power-aware compilers consider how instructions affect energy consumption. Selection can help minimize:

- Switching activity on buses.
- High-power instructions like floating-point divides.
- Memory traffic by reusing registers more effectively.

Power-sensitive instruction selection is becoming increasingly relevant in mobile and IoT applications.

FUZZING AND TESTING THE SELECTOR

Instruction selection must be correct, robust, and resilient. Compilers are tested through fuzzing, random IR generation, and differential testing.

- Random IR tests help expose corner cases.
- Known-good inputs are compiled and executed to verify correctness.
- Comparison against other selectors (e.g., GCC vs LLVM) can reveal subtle bugs.

This ensures the selector emits legal, predictable instructions.

ERROR RECOVERY IN SELECTION

What happens if the selector can’t find a valid pattern? LLVM typically reports a fatal error, but recovery mechanisms are possible:

- Fallback to generic expansion (macro-like patterns).
- Emit helper calls to runtime library routines.
- Retry selection after legalizing operands or types.

Instruction selection isn’t just about success—it’s about what to do when things don’t match.

DIAGNOSTICS AND LOGGING

Instruction selection can be complex and opaque. When things go wrong—or just to better understand what's going on—diagnostics and logging become essential tools.

- LLVM provides detailed debugging output using flags like -debug, -print-after, and -print-before.
- These logs show how IR is transformed at each stage and which patterns matched (or failed to match).
- You can also visualize the SelectionDAG or MIR for further insight.

Example:

```
llc -debug -print-after=instruction-select file.ll
```

This helps developers trace specific selections and spot errors or inefficiencies in the process.

HUMAN-READABLE IR AND SELECTION ANALYSIS

LLVM IR is designed to be readable by both machines and humans. This makes it easier to analyze and debug instruction selection behavior.

- **By inspecting the IR**, you can anticipate which instructions the selector will generate.
- **You can detect issues** like uncanonicalized forms (e.g., add 0, x) that prevent good matches.
- **You can also trace transformations** from original source to IR to machine code.

This transparency is invaluable for compiler developers and students alike.

REAL-WORLD SELECTION FAILURES

Even well-tested selectors can make poor choices. These misfires usually show up in the form of slow binaries, excessive register spills, or redundant instructions. Common examples:

- Using a general `mul` instead of `shl` when multiplying by a power of two.
- Emitting extra `mov` instructions due to missed operand reuse.
- Failing to fold constants into addressing modes.

These issues often arise due to subtle issues in pattern ordering, missed simplifications, or target constraints.

FUTURE OF INSTRUCTION SELECTION

The future of instruction selection is moving toward **automation**, **adaptivity**, and **tighter integration** with the rest of the backend pipeline. Emerging directions:

- **Machine learning-based selection**: Using models trained on large corpora of IR/machine code pairs.
- **Auto-tuning selectors**: Systems that refine pattern costs based on profiling feedback.
- **Unified selection + scheduling**: Combining multiple backend steps to enable global optimizations.

As architectures become more heterogeneous, selectors will also need to specialize for GPUs, accelerators, and mixed instruction sets.

COMPARISON WITH OTHER COMPILERS

LLVM is powerful, but not alone. Comparing it to other compilers reveals different approaches to the same problem.

- **GCC** uses RTL (Register Transfer Language) and hand-written matchers.
- **MLIR** enables extensible dialects and deferred lowering.
- **Academic compilers** may use graph rewriting or ML-based heuristics.

Each approach has strengths. Studying them provides perspective on what choices are possible and why.

And different compilers are also use for error checking!

SURVEY OF INSTRUCTION SELECTION METHODOLOGIES (1)

Over the years, different strategies have been developed for instruction selection. Each comes with its own strengths and trade-offs. Let’s take a step back and look at the broader landscape of methodologies.

- **Macro Expansion**: The simplest method. It directly replaces each IR instruction with a fixed sequence of machine instructions. Easy to implement but produces inefficient code.

- **Tree Pattern Matching**: Matches IR trees to instruction templates using dynamic programming. Efficient and widely used in compilers like LLVM (SelectionDAG).

SURVEY OF INSTRUCTION SELECTION METHODOLOGIES (2)

Over the years, different strategies have been developed for instruction selection. Each comes with its own strengths and trade-offs. Let’s take a step back and look at the broader landscape of methodologies.

- **DAG Covering**: Extends tree matching to allow shared subexpressions. More powerful but computationally harder.

Example:

```
x = a + b + c;
```

Macro expansion might produce 2 add instructions in sequence. Tree/DAG covering could analyze and combine them optimally based on usage.

SURVEY OF INSTRUCTION SELECTION METHODOLOGIES (3)

Over the years, different strategies have been developed for instruction selection. Each comes with its own strengths and trade-offs. Let’s take a step back and look at the broader landscape of methodologies.

- **Bottom-Up Rewriting (e.g., BURG)**: Uses grammars to select instructions bottom-up with cost annotations. Compact and formal, often used in research compilers.

- **Graph Rewriting and SSA-aware Matching**: Tries to cover entire control/data flow graphs, often used in academic tools.

SURVEY OF INSTRUCTION SELECTION METHODOLOGIES (4)

Over the years, different strategies have been developed for instruction selection. Each comes with its own strengths and trade-offs. Let’s take a step back and look at the broader landscape of methodologies.

- **Machine Learning Models**: Trained on datasets of IR+ASM to predict optimal mappings. Still experimental, but promising.

These methods differ in what they prioritize: speed, code size, maintainability, or extensibility. Modern compilers sometimes mix techniques depending on the target and compilation phase.

SUMMARY AND FINAL THOUGHTS (1)

Instruction selection is both practical and deep. It turns abstract logic into hardware instructions, shaping the performance and correctness of compiled code.

- **It requires balancing competing goals**: speed, size, power, and debug-ability.
- **It works hand-in-hand** with instruction legalization, register allocation, and scheduling.
- **It is evolving**, becoming smarter and more integrated over time.

A good understanding of instruction selection is foundational to mastering compilers.

SUMMARY AND FINAL THOUGHTS (2)

There are many matters to think about or explore where working with compilers:

- When is it worth emitting more instructions for better register use?
- How could machine learning help select better instructions?
- What trade-offs would you consider when targeting embedded vs. high-performance CPUs?
- Should we unify selection and scheduling into one step?
- How do you define “better” code when there are many valid encodings?

There are no perfect answers—only thoughtful trade-offs.

See you at the next Lecture!

Don’t forget to consult the Q&A blog on Brightspace!

BACKUP SLIDE: AI AND INSTRUCTION SELECTION

AI and machine learning are starting to influence compilers. In instruction selection, they can help choose the best instruction among many valid candidates.

- Models can be trained on IR-assembly pairs.
- They can predict instructions, tilings, or costs more accurately than static heuristics.
- Reinforcement learning can be used to explore instruction combinations that maximize performance.

This field is still young, but it’s growing fast. As AI models become more explainable and integrated, they could complement or replace parts of traditional selection.