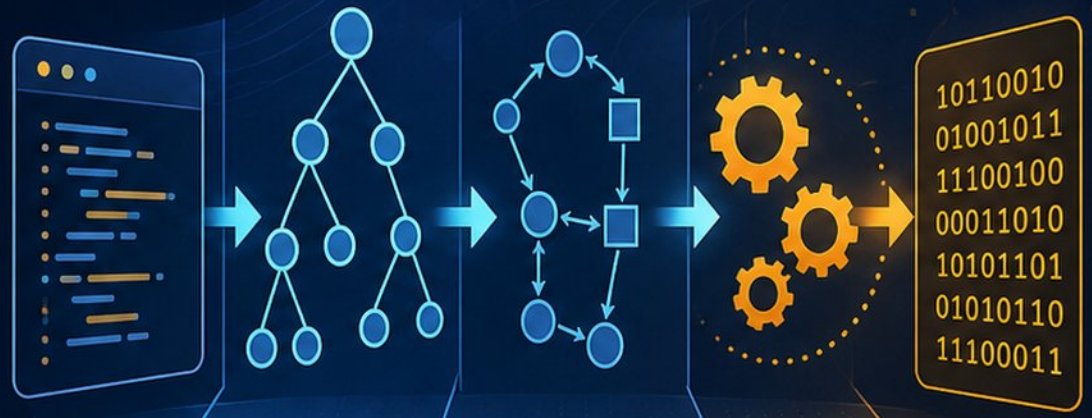
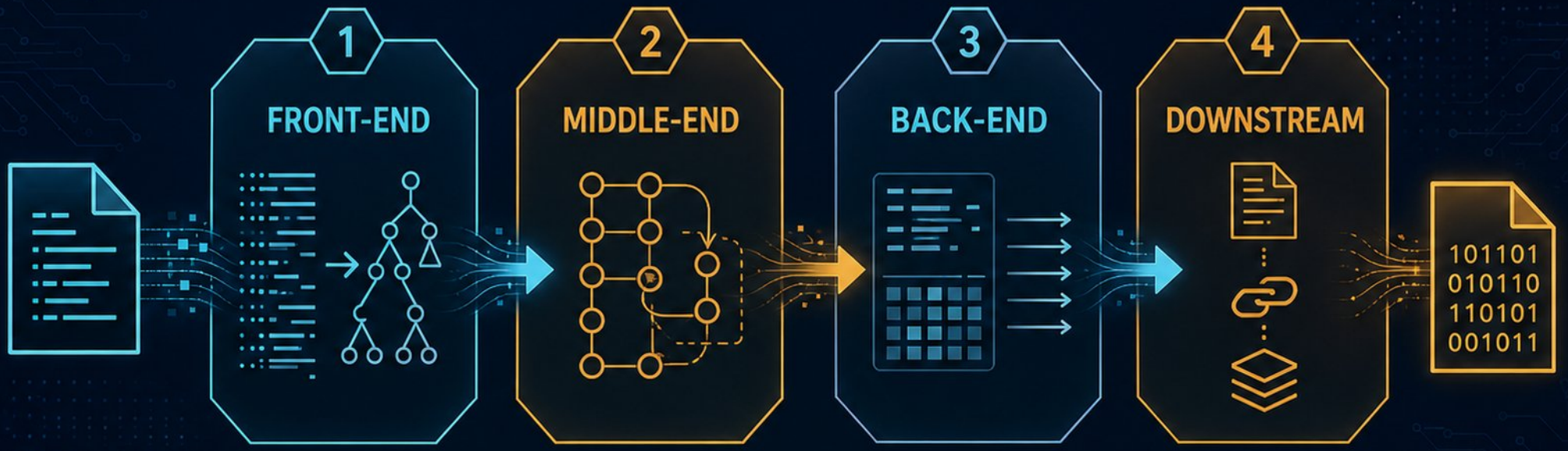
How Compilers Work: From Program Text to Machine Execution

- How does human-readable source code become machine-executable instructions?
- Compilers are multi-stage translators bridging programming languages and hardware.
- End-to-end journey: scanning → structure → meaning → optimization → code generation.
- Goal: Understand how program text is analyzed, transformed, and handed to the machine.



The Compilation Pipeline: A Bird's-Eye View

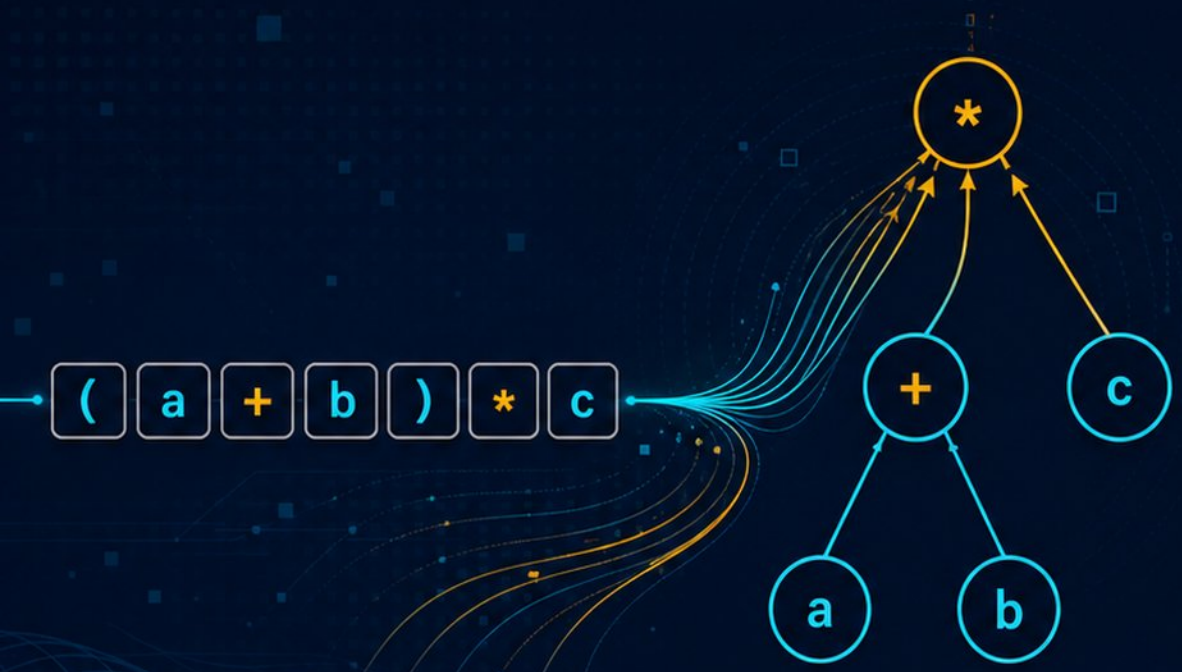


- **Front-end:** lexical analysis, parsing, semantic analysis
- **Middle-end:** IR generation and optimization (LLVM IR/GIMPLE)
- **Back-end:** instruction selection, register allocation, code emission
- **Downstream:** assembler, linker, loader produce the final executable
- **AOT** compiles entirely before execution; **JIT** compiles at runtime

Lexical Analysis: Scanning Text into Tokens

- Tokens are atomic units: keywords, identifiers, literals, operators, delimiters.
- Regular expressions define token patterns; finite automata perform efficient recognition.
- Lexing converts code like `result = a + b * 2` into a token stream.
- Scanner generators (flex/lex) trade readability versus hand-written performance and control.

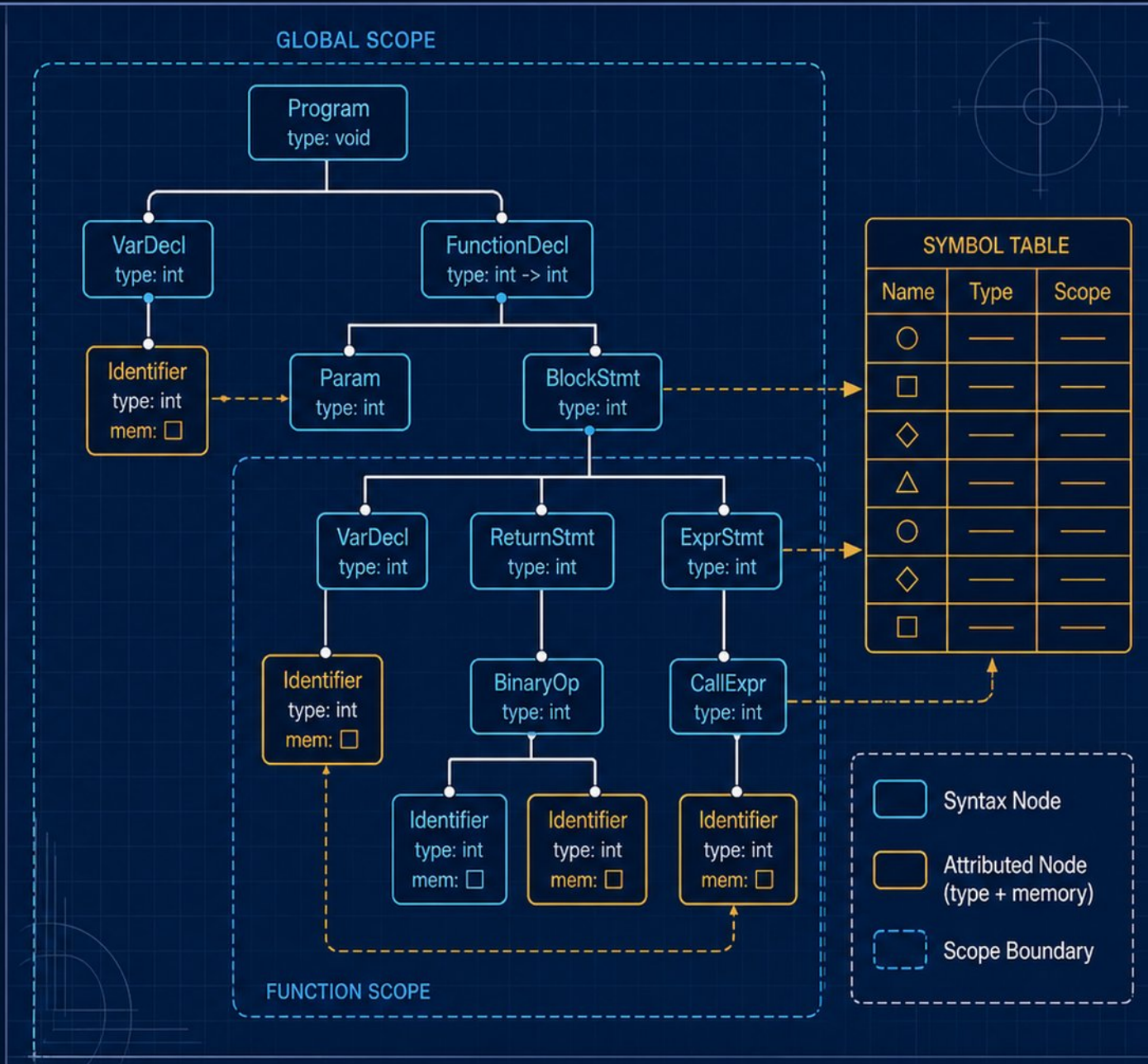
Syntax Analysis: Parsing Structure from Tokens



- Transform linear token sequence into hierarchical parse trees and abstract syntax trees (ASTs)
- Top-down (recursive descent) vs. bottom-up (LR) parsing: understand when each approach fits best
- Define syntax formally with context-free grammars for expressions, statements, and blocks
- Handle real-world issues: ambiguity, operator precedence, and robust error recovery
- Modern tools: ANTLR, tree-sitter for incremental parsing, and hand-crafted recursive descent parsers

Semantic Analysis: Adding Meaning to Trees

- Distinguishes meaning from syntax: type checking, scope resolution, and declaration enforcement
- Symbol tables track names, types, and scopes across the entire program
- ASTs are decorated with types and memory locations to form attributed ASTs
- Catches type mismatches, undeclared variables, and ambiguous overloads

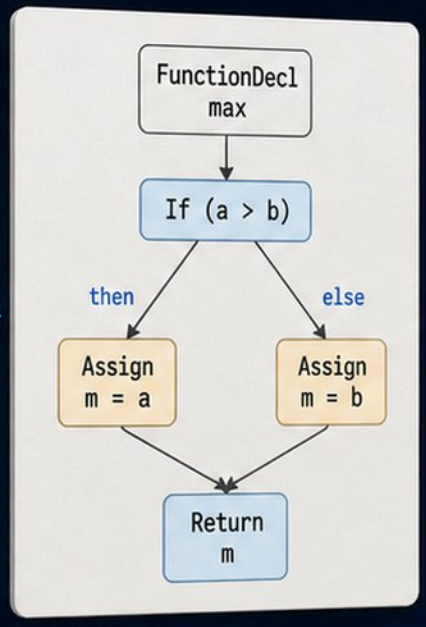


Intermediate Representation: The Compiler's Internal Language

1) Source Code

```
int max(int a, int b) {
  int m;
  if (a > b)
    m = a;
  else
    m = b;
  return m;
}
```

2) AST (Abstract Syntax Tree)



3) Three-Address Code (Linear IR)

```
1   param a
2   param b
3   t1 = a > b
4   if t1 goto L1
5   goto L2
6
7  L1:  m = a
8       goto L3
9  L2:  m = b
10      goto L3
11 L3:  return m
12
```

4) LLVM IR (SSA Form with Phi Nodes)

```
entry:
  %cmp = icmp sgt i32 %a, %b
  br i1 %cmp, label %then, label %else

then:
  %m1 = add i32 %a, 0
  br label %merge

else:
  %m2 = add i32 %b, 0
  br label %merge

merge:
  %m = phi i32 [ %m1, %then ], [ %m2, %else ]
  ret i32 %m
```

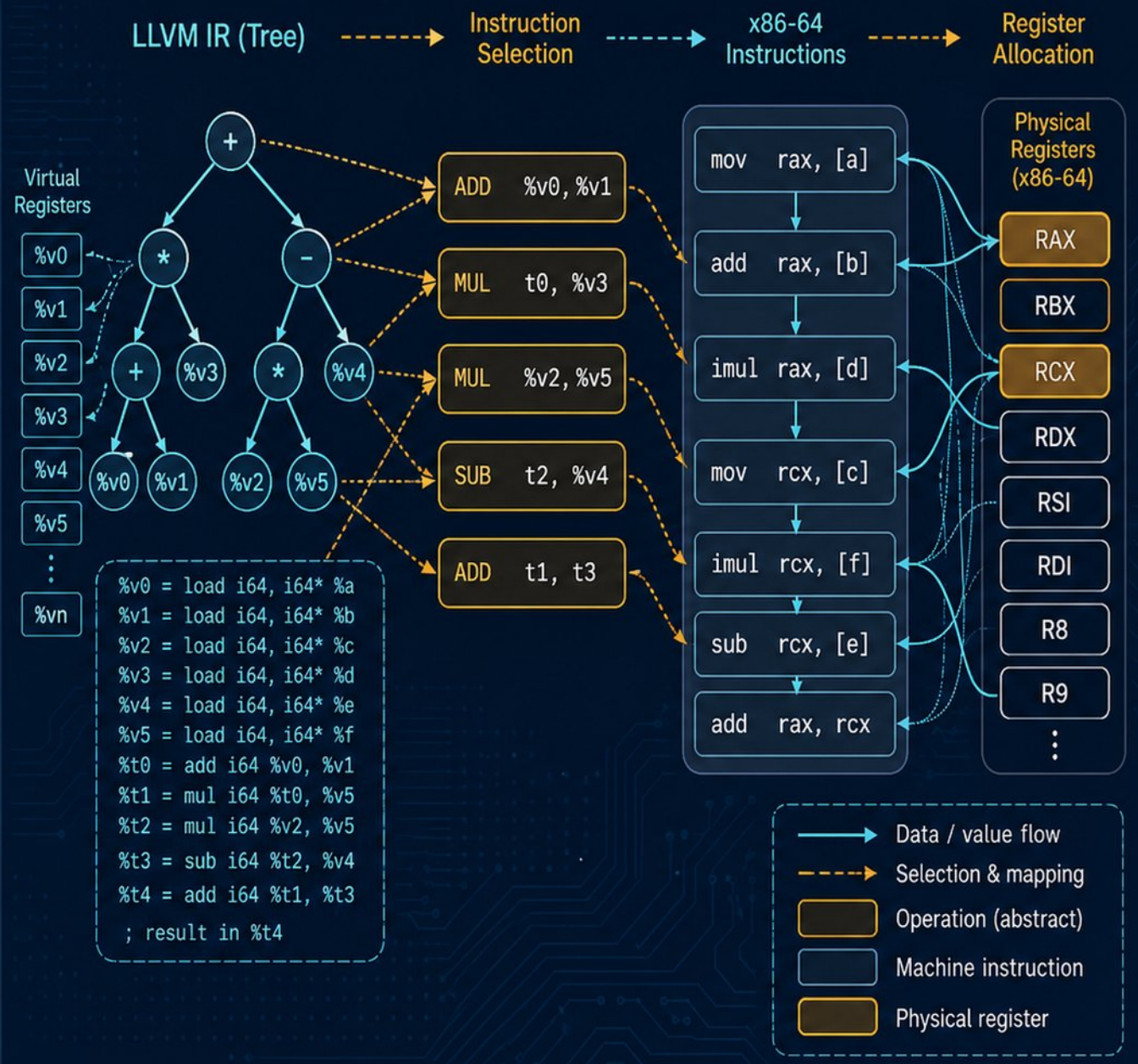
- Decouples **N** language front-ends from **M** hardware back-ends, enabling shared optimizations
- **Linear IR:** three-address code, bytecode. **Graph IR:** data-flow graphs, SSA form
- **SSA:** each variable assigned exactly once; **phi** nodes handle control-flow merges
- Example: AST lowered to three-address code, then to LLVM IR with **phi** nodes
- LLVM IR, GCC GIMPLE, Java bytecode, WebAssembly—different abstraction and trade-offs

Optimization: Making Programs Faster, Smaller, and More Efficient

- Compilers outperform naive translation by exploiting structure and target knowledge
- Scope: peephole → intraprocedural → interprocedural → link-time (LTO)
- Key passes: constant folding, CSE, dead code elimination, loop-invariant motion, inlining
- GCC/LLVM levels: -O0 (debug) → -O1/-O2 (balance) → -O3 (aggressive) → -Os (size)

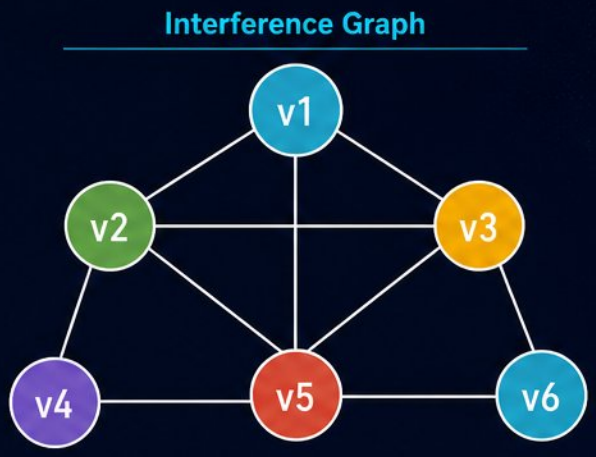
Code Generation: From IR to Machine Instructions

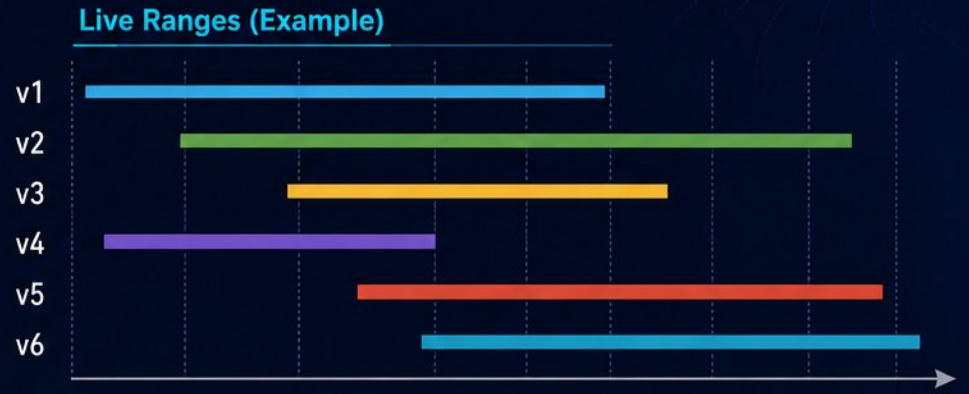
- Final transformation: target-independent IR specific ISA operations
- Instruction selection uses tree-pattern matching and tiling heuristics
- Instruction scheduling reorders to exploit pipelines and avoid stalls
- Register allocation maps virtual registers to physical via live-range analysis and graph coloring
- Trace LLVM IR for an arithmetic expression down to x86-64 assembly



Register Allocation Deep Dive: Graph Coloring and Live Ranges

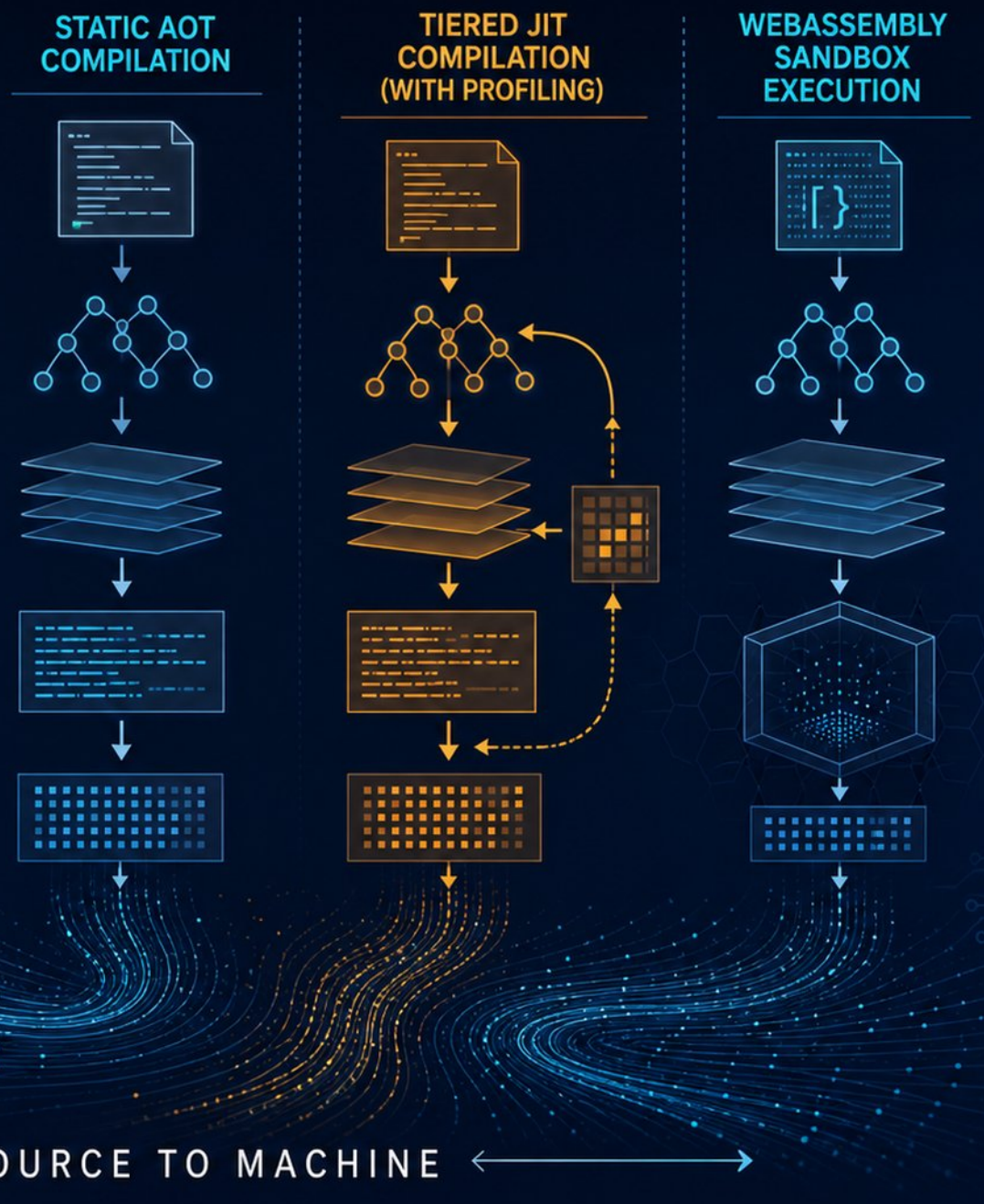
- Registers are the **fastest storage**, but severely limited in number
- **Interference graph:** nodes = virtual regs; edges = overlapping lifetimes
- **Graph coloring** assigns physical registers; spills go to memory if tight
- **Key algorithms:** Chaitin-style coloring, linear scan for JIT speed
- **Modern advances:** LLM-assisted allocation; exact methods handle 16+ registers



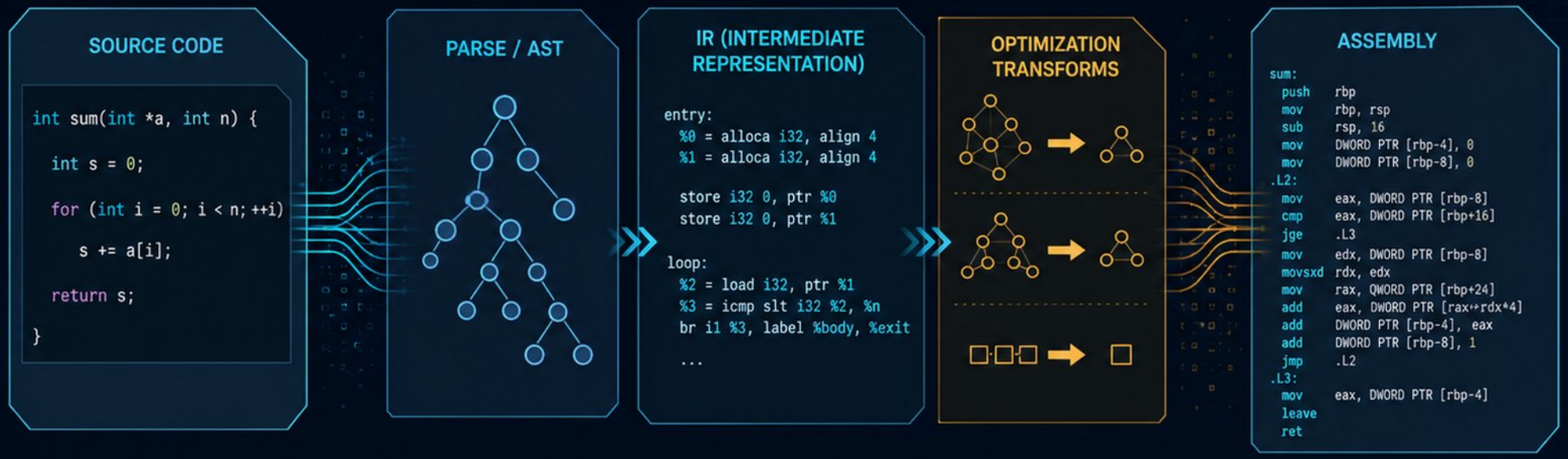


Beyond Static Compilation: JIT, PGO, and Emerging Targets

- JIT compilation trades compile time for profiling: V8/HotSpot use tiered JIT with de-optimization
- Profile-Guided Optimization feeds runtime frequencies back into static builds for better decisions
- WebAssembly 3.0 adds GC, Memory64, exception handling; runs in browsers and server-side
- GPU/ML targets: PTX, SPIR-V, and MLIR dialects progressively lower ops to hardware kernels

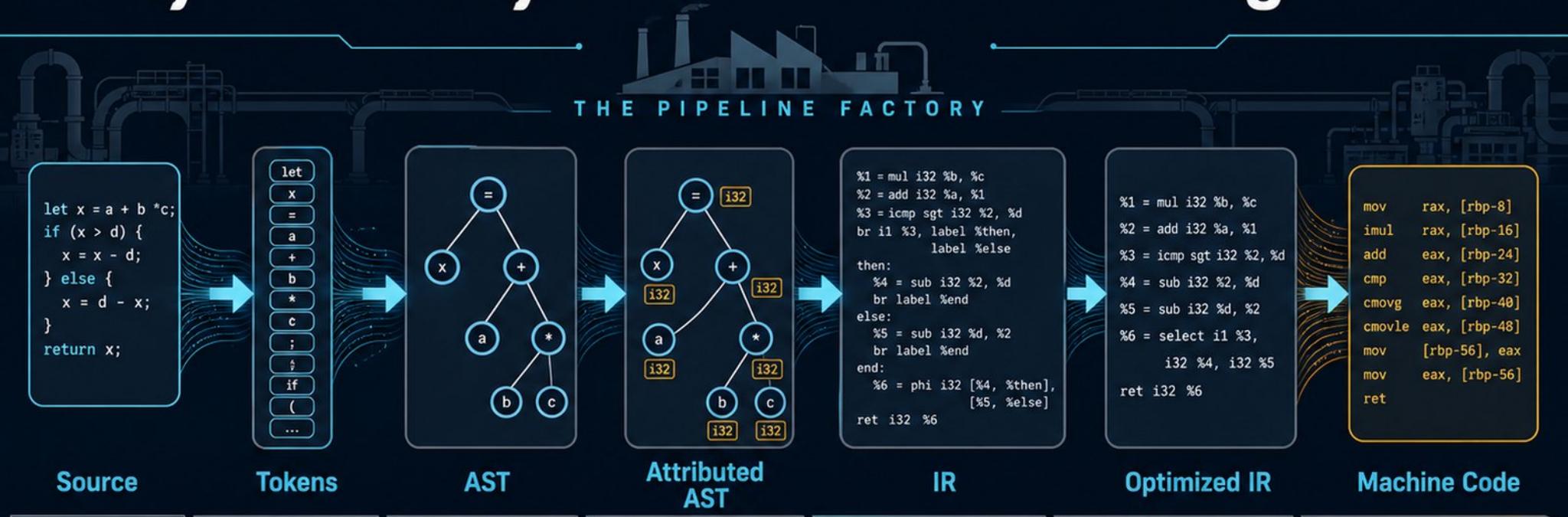


Seeing the Compiler in Practice: Tools, Artifacts, and Exploration



- Compiler outputs: AST dumps (`-ast-dump`), IR notes (`-fdump-tree-*`), and assembly (`-S`)
- Compiler Explorer (godbolt.org) lets you interactively compare source, IR, and assembly
- Live demo: compile a simple C function at `-O0`, `-O2`, `-O3`; observe the generated IR and assembly side-by-side
- LLVM toolchain: `opt` for optimization passes, `llc` for IR-to-assembly, `lli` for JIT execution

Key Takeaways and Further Learning Paths



IR (SSA form) enables one front-end to target dozens of architectures via **LLVM/GCC**

Books: Crafting Interpreters (hands-on), Dragon Book (depth), Kaleidoscope (LLVM IR)

Practice: write an expression compiler, study chibicc, experiment in Compiler Explorer

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/fb76abeb/public