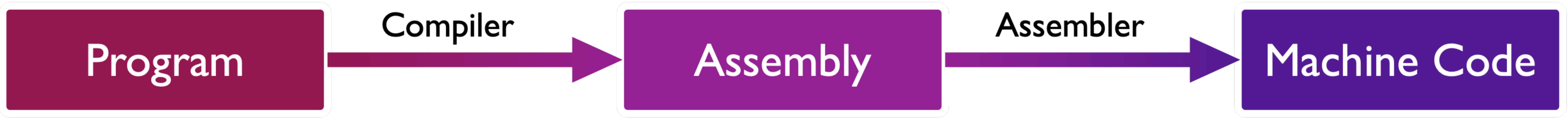
Types of Compilers

traditionally …

- Any program that translates one representation [of a program] to another can be thought of as a compiler.
- But we can think of a few different types of compilers for high level programming languages, based on what kind of representations they translate to

1. High level language → assembly language (e.g., llvm)
2. High level language → machine-independent code (e.g., javac)
3. Machine-independent code → assembly (e.g., Java's JIT compiler)
4. High level language → high level language (e.g., domain-specific languages, source-to-source optimizers)
5. Low level language → low level language (e.g., Apple's Rosetta 2)

high-level to assembly



- Compiler converts program into assembly

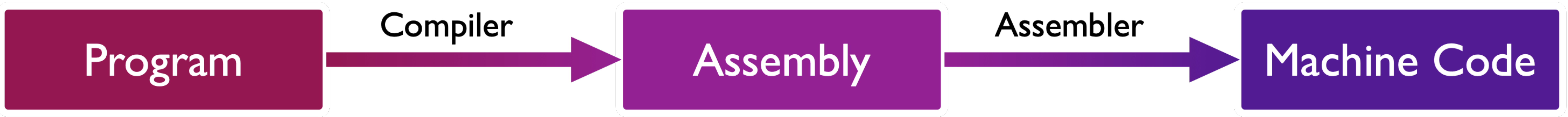
```
t1 = t2 + 1  →  addi r1 r2 1
```

- Assembler is a machine-specific translator that converts assembly into machine code

```
addi r1 r2 1  →  000000000001 00010 000 00001 0010011
```

- Conversion is usually one-to-one with some exceptions
  - Program locations
  - Variable names

high-level to assembly



Program

Compiler

Assembly

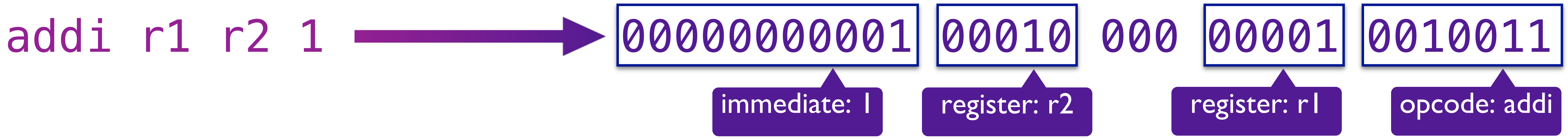
Assembler

Machine Code

- Compiler converts program into assembly

```
t1 = t2 + 1 → addi r1 r2 1
```

- Assembler is a machine-specific translator that converts assembly into machine code



```
addi r1 r2 1 → 000000000001 00010 000 00001 0010011
```

immediate: 1

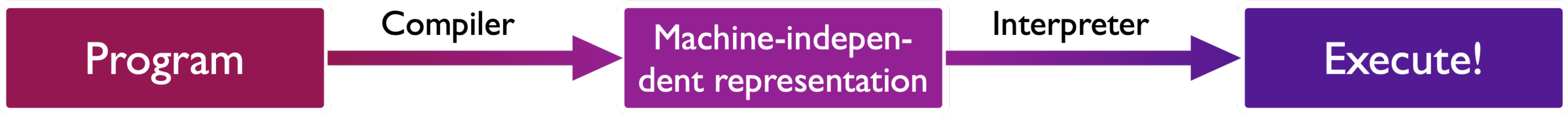
register: r2

register: r1

opcode: addi

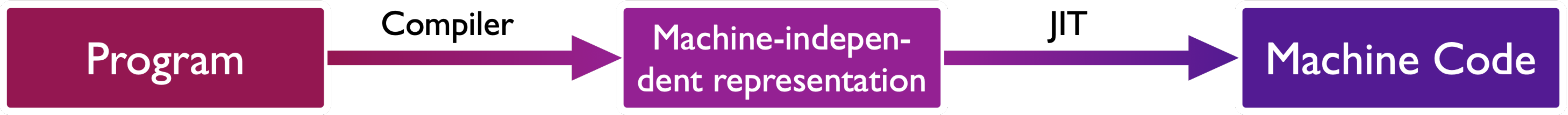
- Conversion is usually one-to-one with some exceptions
  - Program locations
  - Variable names

high-level to machine-independent



- Compiler converts program into machine-independent representation
- Interpreter then processes and executes this representation "on-the-fly"
  - Operations are "executed" by invoking methods of the interpreter, rather than directly executing on the machine
- Compiler and interpreter can be separate
  - e.g., javac translates Java programs into Java *bytecode*, Java interpreter executes bytecode
  - Bytecode is like assembly language, but not tied to a specific machine
- May have a single program (just called an "interpreter" then)
  - e.g., most scripting languages, like python, Perl.
- Aside: what are the pros and cons of the interpreter-based approach?

machine-independent to assembly



- First part works just like with an interpreter: convert program to machine-independent representation
- Replace the interpreter with *another compiler*
- This *just-in-time* compiler (JIT) compiles code *while the program executes*
  - As JIT, compiled ("native") code takes over from interpreted code
- Is this better or worse than a compiler that generates machine code directly from the program?
  - What code does JIT compile?

high-level to high-level

- Some times, the goal of a compiler is not to generate code to run, but to just generate another representation
- Modernize legacy code
    - Air Force’s conversion from COBOL to Java
- Reuse programming tools
    - Translate restricted, domain-specific language (e.g., SQL) to general-purpose language
- Keep program in the same high-level language
    - Many optimizing compilers just rewrite the source code of a language

Low-level to low-level

- Modernize legacy machine code
  - Rosetta: PowerPC → x86
  - Rosetta 2: x86-64 → ARM64
- Compatibility and Performance

next: what are the phases of a compiler?

Or: What translations does a compiler do to compile?