Facile
Fake Assembly Code
for an Instructional Language Environment

Computer Science 210
Fall 2024

Instructor: Jeff Chase

1 Introduction

Facile is an architecture for a programmable abstract machine suitable for implementation as a computer processor or central processing unit (CPU). It defines a simple language for software instructions: an assembly language called facileASM. The C implementation lets you write programs in facileASM and then run them on a machine emulator (facileCPU), feed them inputs, and see their outputs.

That paragraph uses some jargon and terminology for some important concepts in this course. §2 introduces some of them. The purpose of Facile is to provide the simplest possible concrete reference example for computer architecture and assembly language (§3). The name "facile" is a backronym: it means "too easy"—ignoring complexities of the real world. Indeed, Facile is designed to be as simple as it can be. It leaves out many details of real computers: §4 has a partial summary. Even so, Facile is sufficiently powerful to illustrate some Big Ideas in programming systems and languages (§5).

Digging into the code. The Facile implementation is only a few hundred lines of C code. The code is easily readable by a novice C programmer as a model for introductory C programming. You might want to gloss over some error handling and other features that make it easy to use from the Linux command line. It uses C pointers in a few places: think of them just like Java references at first.

2 Talking about machines and systems

A computer is a machine that runs software—programs. Programmability is a uniquely powerful property. A programmable machine can perform many different tasks or functions, encoded in its software. We can choose to do something different without any change to the machine: we just change its software, which is much easier.

All of computer science is concerned with programmable machines and how to define them, build them, use them, and reason about them. In this course we focus on the point where the software meets the hardware—what real machines in use today "look like" to software. We also study how we pile *systems software* on top of the machine model to make it easier for programmers to get it done without having to spell out out every tedious detail of their programs. A wise professor once said: "system software is all the code you did not have to write to get your program to run." And there is a lot of it! This course includes an introduction to some: libraries, build tools, operating systems, and command shells.

To get into all that, we must talk about machines and their software in much more detail than an introductory programming course. We start with some basic terms. Refer to Figure 1.

CPUs and cores. The core of any computer is a processor or Central Processing Unit (CPU) that executes software. Today we often call a processor a *core*. Properly speaking, a core is a block of digital logic that implements a processor. It might appear on a computer chip along with other circuitry including other

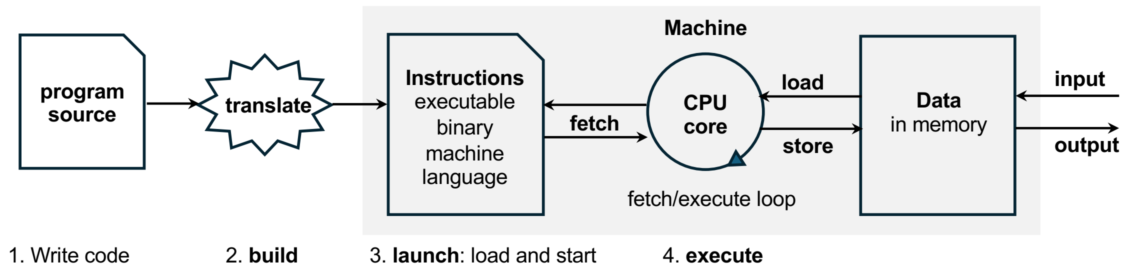

Figure 1: Programmers write program source code in programming languages with various design tradeoffs. Then various build utilities (e.g., compiler, assembler, linker) translate the source program into a binary machine language: a sequence of instructions in a machine-specific format. To launch a program P, system software first loads the binary into machine memory. It then commands a CPU processor core to begin executing the P's instructions. P controls the core: the core fetches and executes instructions in P until P terminates. Some of those instructions load data from memory or store data in memory. Various input/output devices move data in and out of memory under software control.

processor cores that replicate the identical logic template. For now we consider a machine with *one processor, one CPU, one core.* So for our purposes those terms mean the same thing and are interchangeable.

Multicore. A typical CPU chip today is a *multicore* chip: it has multiple processor cores that run in parallel. Software can harness the power of multiple cores on the same chip, or on multiple chips, to get work done faster, e.g., to compute a result in less time or to complete tasks at a higher rate. But that introduces new challenges to manage the interactions for safe and correct behavior and good performance. Multicore is a topic for later.

Instruction Set Architecture (ISA). To know how to program a machine, we have to specify precisely what it does and how it behaves, from the point of view of software. That is the purpose of a machine *architecture*, also called *Instruction Set Architecture (ISA)*. The ISA defines a set of operations (*opcodes*) that the CPU can perform. An *instruction* is a primitive command that tells the CPU an opcode to perform and what data elements (the *operands*) to apply the operation to.

Memory. A machine architecture also defines a set of *state elements*—named locations to hold data for use as operands for instructions. Within the machine, CPU cores are connected to a memory that holds code (instructions) and data for use by programs. To speed things up, the CPU also has fast temporary locations called *registers* to hold data items while in active use.

Assembly language (ASM) represents instructions in a human-readable form. By "human-readable" we mean that it is a string of characters that we can type on a keyboard and read and pronounce. For example, each instruction opcode is represented by a pronouncable *mnemonic* that suggests its action, like `add` for an instruction that adds two numbers.

You can think of ASM as a primitive programming language. ASM is "primitive" because every detail of a task must be broken down into groups of instructions—the basic steps defined by the ISA. ASM lists every instruction of the program individually, and it reflects all details of the ISA of the target machine. For example, it lets programmers access the CPU registers by name.

Assembler. Once we have an ASM program, there is still a lot of painstaking and tedious work to translate it into the form the machine can execute, which is just a bunch of numbers. Fortunately a utility program called an *assembler* does that for us. It reads the ASM as input and outputs the instructions in executable binary machine language. For example, in Figure 1, the source code could be ASM and the translator is the assembler.

Programming languages and compilers. Most programmers write their software in programming languages that are easier to use than ASM. Examples include C, Java, and Python. We call them *high-level* *languages* because they express computations using constructs that are independent of the details of any specific machine or ISA. Much of this course concerns the implementation of high-level programming languages, using C as an example. When we build C source code, the C *compiler* translates the C into ASM code for us. Then the compiler invokes the assembler to take over as before.

Some words about words

Computer systems are developed by humans, and humans are often imprecise or inconsistent in their use of language. There are lots of ways to get confused! Here are a few points to keep in mind:

- Words might have different meanings depending on context. There are only so many words to go around to describe so many concepts, so we often reuse them. For example, in Figure 1, the word *load* means to bring data in to nearby storage. In one use it means to copy a program binary into machine memory, and in another it means to retrieve a value from memory into a register.
- The meaning and usage of words may change over time. For example, in the distant past we called memory "core". (We had our reasons.) That usage survives in terms like "core dump"—to save the memory contents of a running program into a file for post-mortem debugging. We see later that the meaning of the word "word" has itself changed over time: we use *word* to refer to a unit of data or storage of a certain size, which grows as technology advances. So: you have to be adaptable in your use of language.
- We often blur or skip over certain distinctions to talk at a suitable level of detail. For example, in Figure 1 a CPU core operates on data in memory. The core is under the control of a program running on the core and telling it what to do. So we might say it is the program that operates on the data, and ignore the core as just an implementation detail. As you gain experience with computer systems you become more comfortable with such shorthands.
- Take care not to read extra meaning into the terms. Consider "assembler". Large programs may have many parts or modules that are translated separately: there is "some assembly required" to combine and connect the parts into a complete program. Naturally you might think the "assembler" does this assembly, but it is instead a different utility program called the *linker* that does it. Why do we call it an assembler then? It's an old term, and we must accept other people's terminology to get along. An assembler by any name would be as sweet: in the old days (1940-50s), humans input binary code directly by flipping manual switches on the front of the computer.

State. The word "state" also has multiple meanings. For our purpose it just means data. The "state elements" in the ISA are pieces of the machine that hold data. Generally we use the word "state" to refer to one condition from a set of possible conditions of some thing, like "state of mind". If you take Discrete Math, you might learn that Finite State Machines (FSMs) model computation as transitions among a finite set of named states. Our machine has a more powerful model for transitions, but like an FSM it has a finite set of states. At any given point in time, the machine holds exactly one combination of values in its various state elements—the state of the machine. The set of elements is finite, and each has finite space, so the set of possible combinations of values is finite. We can also talk about the state of a piece of software to mean its data and/or the current values of its data.

Static vs. dynamic. Figure 1 illustrates a distinction that is useful for talking about programs. We call a program property *static* if it is determined prior to launch—on the left side of Figure 1. In particular, a program element or property is static if it is determined or "bound" at build time, e.g., by a compiler, assembler, or linker. We say a property of a program is *dynamic* if it emerges as the program runs—on the right side of the figure. For example, systems permit programs to allocate more memory space as they run, so the size of memory space in use is dynamic.

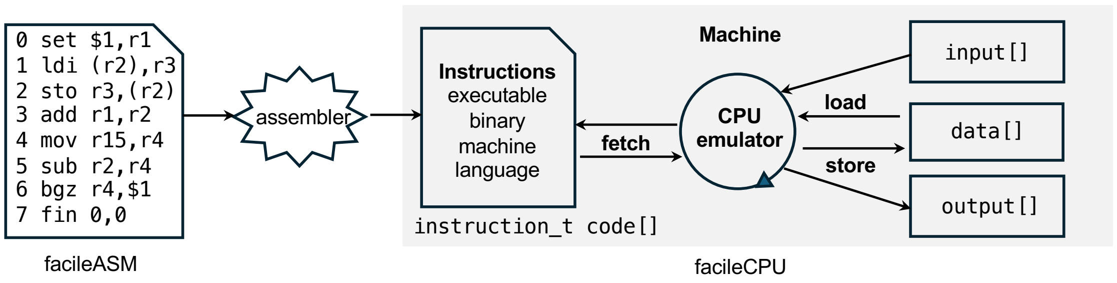

Figure 2: A facileASM program to copy input to output (echo). An assembler (§3) reads the ASM source and outputs a binary representation: an array of structs, one for each instruction. The facileCPU machine emulator interprets the binary to execute the program. Facile programs access their input and output in designated segments of memory (see §3).

Opcode	Operands	Description
`set`	`$value,rd`	Set the value in `rd` to `value` (an integer); place `value` in `rd`.
`add`	`rs,rd`	Add the value in `rs` to the value in `rd` and place the result in `rd`.
`sub`	`rs,rd`	Subtract the value in `rs` from the value in `rd` and place the result in `rd`.
`nop`	`0,0`	No operation, no effect. The operands are ignored and must be '0'.
`fin`	`0,0`	Exit/halt the program. The operands are ignored and must be '0'.

Table 1: Some simple instruction opcodes in facileASM. They illustrate two operand modes: immediate (`$value` for integer constant *value*) and register (`rs` for source value or `rd` for destination/result).

3 Facile architecture

Facile defines a simple assembly language (facileASM) with a dozen or so basic instruction types (opcodes). Figure 2 depicts an exemplary facileASM program. Let's call the program P. Facile includes an assembler that translates P to its binary representation to load into machine memory. A machine emulator (facileCPU) interprets the binary code to run P just as a "real" CPU would. This section summarizes the Facile architecture/ISA from the reference point of a program P.

Registers. Facile has 16 registers numbered `r0-r15`. Each may contain an integer value. The values default to zero, but P may modify them. There are no specific conventions for how P uses its registers.

Memory: segments. It is common to view memory as a set of named regions, called *segments*, with different purpose and use. A Facile program P launches with four memory segments: `code`, `data`, `input`, and `output`. Each segment is an array of locations addressed (indexed) starting at address/index zero up to the highest address, which is one less than the segment's size.

Code segment. Each location of the `code` segment contains a binary instruction. In the C code, each instruction is a structure of type `instruction_t` with integer fields for the opcode (`enum opcode_t`) and operands. The `code` segment is static and immutable—fixed and known before P launches. That means the code is execute-only: no instruction exists that allows P to read or modify its own code.

Operands. Each facileASM instruction has exactly two operands, called *source* and *destination*. A few opcodes always have '0' operands, which are ignored. Otherwise, the operands provide inputs for the operation, and the destination operand gives a location for the result.

Source operand. A source operand specifies a value or a location (a register or memory address) containing a value. The value is taken as an input for the operation. An instruction never writes to a source operand location: the same value continues to reside there after the instruction executes.

Destination operand. A destination operand specifies a register location or a location in memory. The instruction may write a result into the destination location. The value modifies and/or overwrites whatever value resides in that location before the instruction executes.

To illustrate, Table 1 introduces some simple opcodes. The `add` and `sub` are exemplary arithmetic operations: they operate on two inputs—the values in the source and destination registers—and leave the result in the destination. Many other arithmetic opcodes could work this way without complication, but these are enough for some simple programs. Two of the opcodes in Table 1 ignore both operands, which appear as '0': `nop` does nothing, and `fin` halts execution and completes the program—P exits. P also exits if it steps beyond the last instruction in the code segment without encountering a `fin`.

The `set` operation in Table 1 shows how to initialize a register with a "hardcoded" constant value. The source operand encodes a constant integer value directly in the instruction. The destination operand is a register to receive the value.

Operand modes. The examples in Table 1 illustrate different modes for specifying operands. The source operand of `set` uses *immediate mode*: a constant value preceded by '\$'. A *register mode* operand simply names a register. §3.1 describes *indirect mode*. Each facileASM opcode defines exactly one legal operand mode for each of its operands. Any other mode is illegal: the assembler reports an error. That restriction makes facileASM code easy to translate and interpret, and it simplifies the binary representation.

3.1 Addressing data: indirect mode

Loads and stores. In many architectures, the only way to compute on a data value in memory is to first copy (*load*) it from memory into a register. To place a value in memory, it is necessary first to get the value in a register and then to copy (*store*) the value from the register into a specified memory location. Architectures with these restrictions are called *load/store architectures*: they have special instruction opcodes to perform loads and stores, separate from any other computing.

Facile is a load/store architecture. Table 2 lists the opcodes for load and store instructions. Facile designates specific opcodes to load and store on each segment. It includes only the opcodes that make sense: there is no opcode to modify code or input, for example. Figure 3 depicts the memory segments and opcodes to access them.

Data segments. Each location of the `input, output`, and `data` segments contains an integer value. Input is present in the `input` segment before a program P launches: the input segment is static. How does P know the size of the input? As a convention, the system places the input size in `r15` before P starts. P's output is the contents of `output` when P terminates. The `data` segment is for P's temporary use as it runs. The `output` and `data` are dynamic: they start at size 0 and grow as P writes to them.

Indirect addressing. To access memory, the load and store instructions must specify the desired memory address as an operand. We view memory as an array of numbered/indexed locations; the number/index of a location is its *address*. In Facile, the instruction's opcode tells us which segment to access; the address is an index into the selected memory segment. Table 2 illustrates use of *indirect mode* operands for addressing in load and store instructions. The operand specifies the name of a register in parentheses; the value in the register is taken as the address. It allows P to compute or obtain the address as it runs.

To illustrate, Figure 2 shows an echo loop that retrieves a data value from its input at an index in register `r2`, loads the value into register `r3`, then stores the value from register `r3` to its output at the index in register `r2` (see lines 1–2).

In any real implementation, the sizes of the segments and locations and the integer values they contain must fall within predefined limits. But we ignore that detail for now: just suppose they are big enough. *Exercise:* look at the C code and determine what those limits are.

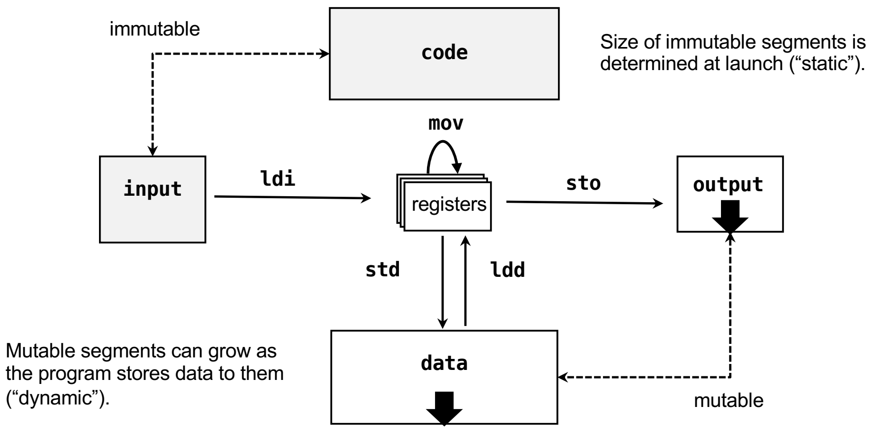
immutable
code
Size of immutable segments is determined at launch ("static").
mov
input
ldi
registers
sto
output
std
ldd
Mutable segments can grow as the program stores data to them ("dynamic").
data
mutable

Figure 3: State elements of the Facile architecture and instruction opcodes that move data among them. The state elements include the CPU registers and four memory segments: `code`, `data`, `input`, and `output`. Facile uses specific opcodes (Table 2) to operate on specific segments of memory, i.e., to copy a data value between a register and a segment. A program cannot modify its `code` or its `input`: they are *immutable*, and their sizes are fixed at launch. The `data` and `output` segments are mutable and dynamic.

Opcode	Operands	Description
`ldi`	`(rs),rd`	Retrieve value from memory at `input[rs]` and load it into register `rd`
`ldd`	`(rs),rd`	Retrieve value from memory at `data[rs]` and load it into register `rd`
`std`	`rs,(rd)`	Retrieve value from register `rs` and store it to memory at `data[rd]`
`sto`	`rs,(rd)`	Retrieve value from register `rs` and store it to memory at `output[rd]`
`mov`	`rs,rd`	Retrieve value from register `rs` and copy it into register `rd`

Table 2: Instructions to access data in facileASM. Load (`ld`) and store (`st`) opcode mnemonics have a suffix indicating which segment of memory to operate on: `d` for `data`, `i` for `input`, and `o` for `output`.

3.2 Control

Now we consider how the machine runs software programs. To launch a program P, system software first loads P's executable binary into memory. (See Step 3 in Figure 1.) Then, it commands the CPU core to execute P's first instruction, at `code[0]`.

Control. The effect of launch is to transfer control of the CPU core to the program. *Control* refers loosely to what code executes next: the next instruction to execute tells the core what to do next. After program P launches, the core continues to execute P's instructions until P completes (e.g., a `fin` instruction), or some other system event causes a transfer of control to other software. For example, P terminates if it tries to execute an impossible instruction, causing the CPU to raise a *fault*. While it is executing in P the core does whatever P says: P controls the core.

Control flow is the stream of instructions that run on the core. A core executes instructions in a sequence, a serial order: first one instruction and then the next. (The chip implementation may use internal parallelism for speed if it at least *appears* to execute in a sequence, so it behaves the same.) *A program is an ordered sequence of instructions.* Think of any program P as a list of numbered lines, with one instruction per line, as shown in Figure 2. After executing an instruction in P at some line, the processor advances (steps) to the next line in P and executes the instruction there.

That sequential behavior is familiar from the way statements execute in any imperative programming language, like C or Java. But interesting programs use *control structures* like conditional statements (`if-then-else`) and loops (`for`, `while`) to vary their control flow, e.g., in response to input. How does that work at the

Opcode	Operands	Description
`jmp`	`0,$target`	Unconditional jump to `target`—the instruction at `code[target]`.
`bez`	`rs,$target`	If the value in `rs` is zero, then jump to `target`.
`bnz`	`rs,$target`	If the value in `rs` is not zero, then jump to `target`.
`bgz`	`rs,$target`	If the value in `rs` is greater than zero, then jump to `target`.
`blz`	`rs,$target`	If the value in `rs` is less than zero, then jump to `target`.

Table 3: Jump/branch instruction opcodes in facileASM.

machine level? We need instructions to vary the control flow.

Control transfers: jumps/branches. Certain instructions—with opcodes called *jump* or *branch*—can change the index of the next instruction to execute. (The terms are interchangeable.) That causes the processor to "go to" another place in the program and execute instructions there.

A jump or branch is a *control transfer*. Consider a point in the instruction stream where a jump/branch causes the control flow (the *dynamic* order) to deviate from the order in which the instructions appear in the program (the *static* order). We say that the jump/branch is *taken*. We call the destination (the next instruction in the stream) the *target* of the branch.

Conditional control and control structures. Useful control structures need a way to check a condition before jumping, e.g., to jump only if a certain data value is zero or not zero. For this purpose, ISAs include instruction opcodes for *conditional* branch. When a conditional branch executes, the control transfer might or might not be taken. If the condition evaluates false in the program's current state, then the processor steps to the next instruction following the branch, rather than jumping to the branch target.

Table 3 summarizes the branch opcodes in facileASM. The conditional branches test a value in a source register and compare it to zero. The branch target is an immediate mode non-negative integer constant, identifying the index of the target instruction in the `code` segment. Facile does not support labels: you have to count the instructions to figure out the correct index yourself. If you get it wrong, your program executes the wrong instructions and goes off the rails. (But you will figure it out, eventually.)

To illustrate, Figure 2 shows a use of the `bgz` instruction at line 6 to implement an echo loop. It subtracts the next location index in register `r2` from the input size in register `r4`. If the result value in `r4` is greater than zero, then there is more input to process, so it jumps to the target: the `ldi` to load the next input value at line 1. Recall that a program launches with the size of the input in register `r15`. This program never modifies `r15`, but uses `mov` to copy the size value into `r4` on each pass through the loop.

In this way, software programs can specify complex input-dependent behaviors, going beyond the straight line execution of a static instruction sequence. With branches, an instruction appearing at exactly one line in the program's static order may execute many times in the dynamic order. For example, consider an infinite loop. Or some parts of the program might be skipped entirely, depending on the input.

3.3 FacileASM syntax

Lines with initial character '#' are ignored as comments. For other non-blank lines, the opcode mnemonic is exactly three characters (no upper case), appearing at the start of the line. The C code obtains each mnemonic and its mandated operand modes (`operand_mode_t`) by indexing `op_templates[]` with the opcode (`opcode_t`). See `facile.h` and `asm.c`.

The mnemonic is followed by exactly one space character before the operands. The source operand is first, followed by a comma, then the destination operand, with no space between them and no trailing characters. The operands conform to the syntax for the mandated operand mode:

- An *immediate* operand is an integer value preceded by character '$'.

- A *register* operand is specified as character 'r' followed by an integer constant in range 0-15.
- An *indirect* operand is specified as a register operand in parentheses.
- Ignored/null operands (mode `NO_OPERAND`) must appear as '0'.

4 Facile simplifications

Facile is restrictive relative to real commercial CPU architectures, like ARM or x64. It eliminates certain common conveniences and features. These restrictions make a programmer's work harder. In particular, it might take more instructions to represent a given computational task. The purpose of the restrictions is to make it easy to describe and implement Facile and and easy for you to understand it. Facile is a toy for purposes of instruction only (oof).

Here are some of the key features to expect in full-featured ISA/ASM but missing from Facile. We discuss most of them later in the course.

- **Negative numbers and floats.** Facile supports only ints. Negative integers work here but we do not talk about them (yet).
- **Data items of varying sizes and ranges.** ASM commonly uses different opcodes or mnemonic opcode suffixes to indicate if the instruction operates on a data unit of 1, 2, 4, or 8 bytes. In Facile everything is an integer that is "big enough". We are not concerned with saving space for small values.
- **Logical operators and shifts.** These operators manipulate the binary representations of data. Facile can add them as new instructions without changing anything else.
- **Offsets for indirect addressing.** ASM typically permits an instruction to add a fixed immediate offset to the address, or even multiply the offset by some factor or add it to a base address from another register. These modes are useful to address arrays and data structures. Your facileASM code needs extra instructions to compute the exact address itself and place it in a register.
- **Condition codes.** Most CPUs have a special set of internal boolean state elements that record status of the most recent previous operation, e.g., whether the result was zero or not zero. Conditional branch instructions query these condition codes. In facile you must copy a result into a register and test it there, as in the echo example (Figure 2).
- **Many more opcodes for conditional execution.** There are many possibilities for instructions to branch or to store data values conditionally. We often say of x64: whatever you want to do, "there's an op for that".
- **Procedure (function) call/return.** Modern architectures have special instructions to call functions, pass arguments, and return control to the caller with a return value. The facile instructions can emulate that, but it is inconvenient.
- **Branch target labels.** Real assemblers let you tag a line of ASM code with a unique string label, for use in a branch target. The assembler replaces it in the binary with the index of the target instruction.
- **Register subfields.** Legacy architectures like x86 have long histories through many revisions, and take care to preserve backward compatibility so old code can still run. They permit you to use "old" opcodes and register names to operate on smaller data units in subfields of the registers. In many cases compilers choose them because the binary code is more compact. You may encounter them in assembly code, forcing you to learn the variations or look them up as they appear.

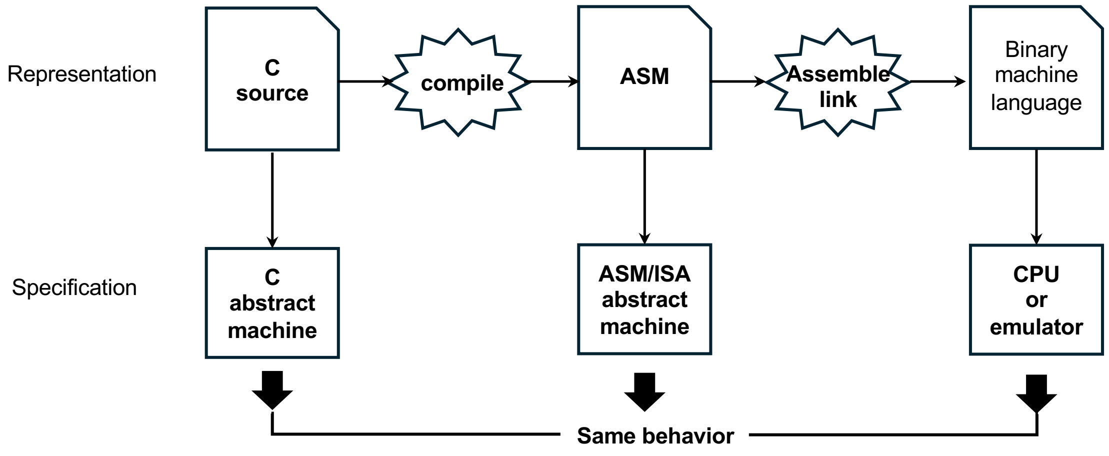

Figure 4: Programs have multiple representations, e.g., in the C programming language, ASM for a particular ISA, or binary ready to execute on a machine. The meaning or *semantics* of each representation is specified by an abstract machine that defines the behavior of all constucts in each language. Build utilities (e.g., compiler, assembler) translate from one representation to another. They are designed to transform the program in a way that preserves its semantics. The objective is to run the binary program on a machine implementation that matches the specification and produces the expected behavior for all valid programs.

5 Abstraction: A peek at some Big Ideas

Figure 4 illustrates some lessons from this quick tour. We have seen that any given program P has multiple representations. Each representation is encoded in some language. Translators convert P from one language to another. Are all the representations of P the same program? Yes, in the sense that P's behavior is defined by the meaning of the source language (e.g., C). The *syntax* (the language of expression) is different, but we presume that the translation steps preserve the *semantics* (the meaning).

Why so many languages? They have different purpose and goals. Machine language (ISA) is designed to be implementable in hardware that runs P fast. At the same time it must have sufficient expressive power to represent all programs of interest. Computer architecture is the study of ISAs for real computers and their implementations and performance. Even so, hardly anyone looks at machine language, since ASM captures the ISA details in a readable form, tells you all you need to know, and is easy to translate. When we do look at machine language (e.g., in a debugger), we first *disassemble* it back to ASM.

At the other end, modern programming languages are designed for expressive power and safety, e.g.. an ability to detect errors at translation time, before the faulty program ever launches. High-level languages have massively enhanced the productivity of programmers.

Compiler. In particular, compilers are a keystone technology in computer science. Their power rests on rigorous and interesting mathematics (e.g., formal grammars) to define languages and generate compiler modules for it. Compiled languages provide portability: we can run an old program P on a new machine without changing the source code. We only have to change the compiler back-end module to output code in the ASM of the new target machine. Linkers operate on the assembler outputs: software parts of a program P might be written in different high-level languages, and the linker can combine them to form P, if the parts are built to the same executable ("binary") format.

Compilers and other translators are examples of programs that operate on other programs. In the world of systems it is common for programs to take other programs as input or produce programs as output. They can implement languages, optimize code, check for errors, inject instrumentation for dynamic monitoring, and many other useful functions. The classic text *Structure and Interpretation of Computer Programs* (Abelson and Sussman) insists "it is no exaggeration to regard this as the most fundamental idea in programming."

Emulator. Figure 4 depicts an emulator (e.g., facileCPU in `machine.c`)—software that "pretends to be a machine" and executes P's machine language. We tend to think of a "machine" as physical hardware in the real world, like a bicycle or a CPU chip. But in computer science, a machine might be software—e.g., a program that accepts a program P as input and interprets it, emulating the "real" machine. In particular, P "does not care" if the machine is implemented in hardware or software: it runs the same.

It is possible to emulate the machine in software because it can only manipulate its own machine state and produce output. Its state elements are nothing more than storage for data of specified kinds and sizes, so it is easy for an emulator to use data structures in its program memory with the same kinds and sizes. The machine cannot do anything in the "real" world. For that we would need another specialized machine to act on the program output, like a robot or an Automated Teller Machine (ATM, a cash machine).

Interpreter. Emulators suggest another alternative to implement high-level languages: define an *intermediate* ISA suited to the needs of the language, compile program P down to the intermediate ISA, and then implement an *interpreter* on any target platform of interest. The interpreter emulates the intermediate ISA. That idea allowed languages like Java and Python to gain widespread adoption rapidly. It removes the need to recompile P to run on different machines: the binary can run on any platform with an interpreter. That property was once called *transportability*, and it is important for web programming languages: "build once, run anywhere."

These portability properties illustrate the power of *abstraction*. In a software systems context, it means defining a function or task in a simplified form that allows multiple implementations to coexist, e.g., for different target platforms. Abstraction frees the user of a function from having to reason about the details: programs written to use them are interoperable with any implementation. If it is done right. And doing abstraction right is a fundamental focus of computer science and computer systems.

Abstract machines

Returning to Figure 4, we still face the challenge of how to ensure that each representation of P has the same meaning and behavior, and that the translators and emulators preserve it. How do we know what a program means or how it behaves? We can build it and run it. But computer scientists who create languages, translators, and machines need much more than that. How can we be sure that semantics and behavior are preserved for all possible programs, and for all possible inputs to those programs?

Specification. The answer lies in rigorous specification of the behavior of each element of the language, individually and in combination. We define the behavior of a language by specifying the behavior of an *abstract machine* that runs programs written in the language. We call the machine *abstract* because its behavior is independent of its implementation: its specification says what it can do and how it behaves.

How to write a specification? How to check that an implementation conforms to a specification? Some formal methods exist; it is an active research area. A common approach, which we take in Facile, is to document the abstract machine (§3) and also provide a *reference implementation*—facileCPU. To check another candidate implementation, we can compare the candidate's behavior to that of the reference on any program P. To be sure, certain errors in P may result in behavior that is specified as *undefined*. In that case, an implementation is free to produce any behavior that is convenient: P has no meaning, even if it passes translation.

Power. A related question is: how to be sure that our representations have matching power, so that a P written in a new language can run on any machine? Can we prove it? Yes! If we can write an emulator program for a first abstract machine M_1 and run the emulator on a second machine M_2, we show that M_2 is at least as powerful as M_1. More precisely, any P that runs on M_1 can also run on M_2: just feed P as input to the emulator. Now, one kind of abstract machine called a Universal Turing Machine has been shown to be as powerful as any programming formalism that can ever exist. Could we implement a UTM in facileASM? I think so—if we add a few more instructions. But that is a topic for another course.