MITOCW | MIT6_004S17_11-02-01_300k

Today we're going to talk about how to translate high-level languages into code that computers can execute.

So far we've seen the Beta ISA, which includes instructions that control the datapath operations performed on 32-bit data stored in the registers.

There are also instructions for accessing main memory and changing the program counter.

The instructions are formatted as opcode, source, and destination fields that form 32-bit values in main memory.

To make our lives easier, we developed assembly language as a way of specifying sequences of instructions.

Each assembly language statement corresponds to a single instruction.

As assembly language programmers, we're responsible for managing which values are in registers and which are in main memory, and we need to figure out how to break down complicated operations, e.g., accessing an element of an array, into the right sequence of Beta operations.

We can go one step further and use high-level languages to describe the computations we want to perform.

These languages use variables and other data structures to abstract away the details of storage allocation and the movement of data to and from main memory.

We can just refer to a data object by name and let the language processor handle the details.

Similarly, we'll write expressions and other operators such as assignment (=) to efficiently describe what would require many statements in assembly language.

Today we're going to dive into how to translate high-level language programs into code that will run on the Beta.

Here we see Euclid's algorithm for determining the greatest common divisor of two numbers, in this case the algorithm is written in the C programming language.

We'll be using a simple subset of C as our example high-level language.

Please see the brief overview of C in the Handouts section if you'd like an introduction to C syntax and semantics.

C was developed by Dennis Ritchie at AT&T Bell Labs in the late 60's and early 70's to use when implementing the Unix operating system.

Since that time many new high-level languages have been introduced providing modern abstractions like object-oriented programming along with useful new data and control structures.

Using C allows us describe a computation without referring to any of the details of the Beta ISA like registers, specific Beta instructions, and so on.

The absence of such details means there is less work required to create the program and makes it easier for others to read and understand the algorithm implemented by the program.

There are many advantages to using a high-level language.

They enable programmers to be very productive since the programs are concise and readable.

These attributes also make it easy to maintain the code.

Often it is harder to make certain types of mistakes since the language allows us to check for silly errors like storing a string value into a numeric variable.

And more complicated tasks like dynamically allocating and deallocating storage can be completely automated.

The result is that it can take much less time to create a correct program in a high-level language than it would it when writing in assembly language.

Since the high-level language has abstracted away the details of a particular ISA, the programs are portable in the sense that we can expect to run the same code on different ISAs without having to rewrite the code.

What do we lose by using a high-level language?

Should we worry that we'll pay a price in terms of the efficiency and performance we might get by crafting each instruction by hand?

The answer depends on how we choose to run high-level language programs.

The two basic execution strategies are "interpretation" and "compilation".

To interpret a high-level language program, we'll write a special program called an "interpreter" that runs on the actual computer, M1.

The interpreter mimics the behavior of some abstract easy-to-program machine M2 and for each M2 operation executes sequences of M1 instructions to achieve the desired result.

We can think of the interpreter along with M1 as an implementation of M2, i.e., given a program written for M2, the interpreter will, step-by-step, emulate the effect of M2 instructions.

We often use several layers of interpretation when tackling computation tasks.

For example, an engineer may use her laptop with an Intel CPU to run the Python interpreter.

In Python, she loads the SciPy toolkit, which provides a calculator-like interface for numerical analysis for matrices and data.

For each SciPy command, e.g., "find the maximum value of a dataset", the SciPy tool kit executes many Python statements, e.g., to loop over each element of the array, remembering the largest value.

For each Python statement, the Python interpreter executes many x86 instructions, e.g., to increment the loop index and check for loop termination.

Executing a single SciPy command may require executing of tens of Python statements, which in turn each may require executing hundreds of x86 instructions.

The engineer is very happy she didn't have to write each of those instructions herself!

Interpretation is an effective implementation strategy when performing a computation once, or when exploring which computational approach is most effective before making a more substantial investment in creating a more efficient implementation.

We'll use a compilation implementation strategy when we have computational tasks that we need to execute repeatedly and hence we are willing to invest more time up-front for more efficiency in the long-term.

In compilation, we also start with our actual computer M1.

Then we'll take our high-level language program P2 and translate it statement-by-statement into a program for M1.

Note that we're not actually running the P2 program.

Instead we're using it as a template to create an equivalent P1 program that can execute directly on M1.

The translation process is called "compilation" and the program that does the translation is called a "compiler".

We compile the P2 program once to get the translation P1, and then we'll run P1 on M1 whenever we want to execute P2.

Running P1 avoids the overhead of having to process the P2 source and the costs of executing any intervening

layers of interpretation.

Instead of dynamically figuring out the necessary machine instructions for each P2 statement as it's encountered, in effect we've arranged to capture that stream of machine instructions and save them as a P1 program for later execution.

If we're willing to pay the up-front costs of compilation, we'll get more efficient execution.

And, with different compilers, we can arrange to run P2 on many different machines - M2, M3, etc. - without having rewrite P2.

So we now have two ways to execute a high-level language program: interpretation and compilation.

Both allow us to change the original source program.

Both allow us to abstract away the details of the actual computer we'll use to run the program.

And both strategies are widely used in modern computer systems!

Let's summarize the differences between interpretation and compilation.

Suppose the statement "x+2" appears in the high-level program.

When the interpreter processes this statement it immediately fetches the value of the variable x and adds 2 to it.

On the other hand, the compiler would generate Beta instructions that would LD the variable x into a register and then ADD 2 to that value.

The interpreter is executing each statement as it's processed and, in fact, may process and execute the same statement many times if, e.g., it was in a loop.

The compiler is just generating instructions to be executed at some later time.

Interpreters have the overhead of processing the high-level source code during execution and that overhead may be incurred many times in loops.

Compilers incur the processing overhead once, making the eventual execution more efficient.

But during development, the programmer may have to compile and run the program many times, often incurring the cost of compilation for only a single execution of the program.

So the compile-run-debug loop can take more time.

The interpreter is making decisions about the data type of x and the type of operations necessary at run time, i.e., while the program is running.

The compiler is making those decisions during the compilation process.

Which is the better approach?

In general, executing compiled code is much faster than running the code interpretively.

But since the interpreter is making decisions at run time, it can change its behavior depending, say, on the type of the data in the variable X, offering considerable flexibility in handling different types of data with the same algorithm.

Compilers take away that flexibility in exchange for fast execution.