CS250: FOUNDATIONS OF COMPUTER SYSTEMS [SOFTWARE]

Write once, run anywhere?
Compile thy code
To an abstract, virtual machine
The VM code

Next translate the VM code
To the machine language
Of the target platform

Viola a code written once
Running wherever
Without re-compiling

SHRIDEEP PALLICKARA
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Are opcodes still being added?
- Are there ARM-based GPUs?
- NVIDIA and RTX: Step in the right direction?

Topics covered in this lecture

- Software
- Virtual machines
- Stack machine

3

4

The magic in "Hello World" [1/2]

- The first magic that we take for granted is that a sequence characters, say, `printString ("Hello World")`, can cause the computer to actually display something on the screen
- How does the computer figure out *what* to do? And even if the computer knew what to do, *how* will it actually do it?

The magic in "Hello World" [2/2]

- The screen is a **grid of pixels**
  - If we want to display "**H**" on the screen, we have to turn on and off a carefully selected subset of pixels
- Of course, this is just the beginning
  - What about displaying this **H** legibly on screens that have different sizes and resolutions?

And beyond simple programs …

- What about dealing with **while** and **for** loops, arrays, objects, methods, classes?
  - And all the other goodies that high-level programmers are trained to use without ever thinking about how they work?
- Indeed, the beauty of high-level programming languages is that they permit using them in a state of blissful ignorance
  - This is true of well-designed abstractions in general

Application programmers and high-level languages

- Application programmers are encouraged to view the language as a **black box abstraction**
  - Without paying any attention to how it is actually implemented
- All you need is a good tutorial, a few code examples, and off you go

Clearly though, at one point or another, someone must implement this language abstraction

- Someone must develop, **once and for all**, the ability …
  - To efficiently compute square roots when the application programmer blissfully says `sqrt(1764)`
  - To elicit a number from the user
  - To find and carve out an available memory block when the programmer nonchalantly creates an object using **`new`**
  - And to perform transparently all the other abstract services that programmers expect to get without ever thinking about them

So, who turns high-level programming into an advanced technology indistinguishable from magic?

- Those who develop compilers, virtual machines, and operating systems

11

The journey from high-level code to machine language

- A high-level program is a **symbolic abstraction** that means nothing to the underlying hardware
- Before executing a program, the high-level code must be **translated** into machine language
- This translation process is called **compilation**, and the program that carries it out is called a **compiler**

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | SOFTWARE | L15.12

12

Writing high-level programs that can execute on any one of many host platforms is a daunting challenge

- One way to streamline this distributed, multi-vendor ecosystem (from a compilation perspective)?
  - Base it on some overarching, agreed-upon **virtual machine (VM)** architecture
- Acting as a common, intermediate run-time environment
  - The VM approach allows developers to write high-level programs that *run almost as* is on many different hardware platforms
    - Each equipped with its own VM implementation

Some languages, for example, Java and C#, employ an elegant **two-tier compilation** model [1/2]

- First, the source program is translated into **an interim, abstract VM** code
  - Called **bytecode** in Java and Python and **Intermediate Language** in C#/.NET
- Next, using a *completely separate* and *independent* process, the VM code can be translated further into the machine language of **any target hardware platform**
- This modularity is at least one reason why Java became such a dominant programming language

Some languages, for example, Java and C#, employ an elegant **two-tier compilation** model [2/2]

- Taking a historical perspective
- Java can be viewed as a powerful object-oriented language whose two-tier compilation model was the right thing in the right time
  - Just as computers were evolving from a few predictable processor/OS platforms into a bewildering hodgepodge of networked PCs, mobile devices ..

Before a high-level program can run on a target computer

- It must be translated into the computer's machine language
- Traditionally, a separate compiler was developed specifically for **any given pair** of high-level language and low-level machine language
- Over the years, the reality of many high-level languages, on the one hand, and **many processors and instruction sets**, on the other led to
  - A proliferation of many different compilers, each depending on every detail of both its source and target languages

One way to decouple this dependency

- Break the overall compilation process into **two** *nearly separate* **stages**
- In the first stage:
  - The high-level code is parsed and translated into **intermediate and abstract processing steps**—steps that are neither high nor low
- In the second stage:
  - The intermediate steps are translated further into the **low-level machine language of the target hardware**

This decomposition is very appealing from a software engineering perspective

- The first translation stage depends only on the specifics of the source high-level language
- The second stage only on the specifics of the target low-level machine language

19

Of course, the interface between the two translation stages needs careful thought

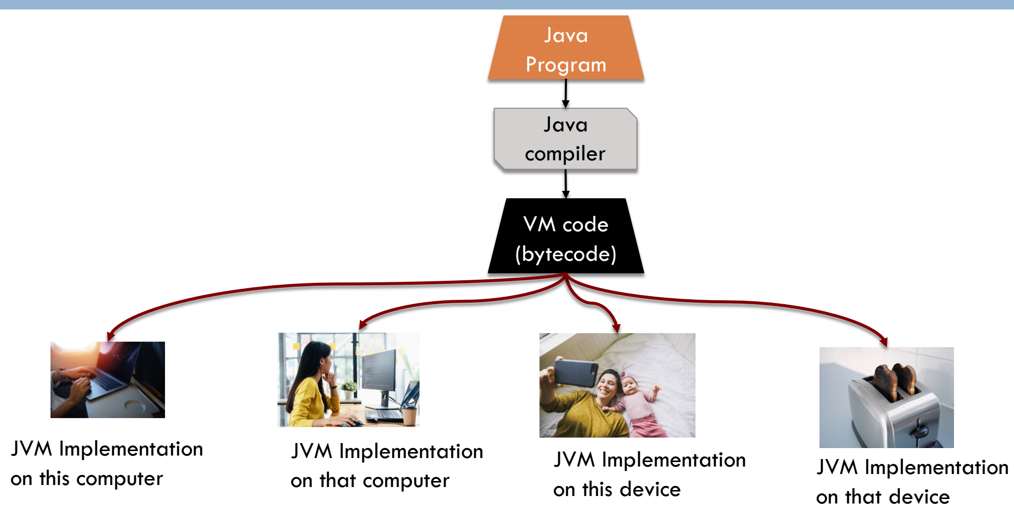
- The exact definition of the intermediate processing steps
  - Must be carefully designed and optimized
- At some point in the evolution of program translation solutions
  - Compiler developers concluded that this **intermediate interface** is sufficiently important to merit its own definition
    - As a standalone language designed to run on an abstract machine
  - Specifically, one can describe a **virtual machine** whose commands realize the *intermediate processing steps* into which high-level commands are translated

20

The compiler that was formerly a single monolithic program is now split

- **Two** *separate* and *much simpler* programs
- The first program:
  - Still termed **compiler**, translates the high-level code into intermediate VM commands
- The second program, called **VM translator**
  - Translates the VM commands further into the machine instructions of the target hardware platform

Virtual Machine framework using Java as an example



The virtual machine framework entails many practical benefits

- When a vendor introduces a new digital device to the market
  - Say, a cell phone
  - The vendor can develop for it a JVM implementation, known as JRE (Java Runtime Environment), with relative ease
- This client-side **enabling infrastructure** immediately endows the device with a huge base of available Java software

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | SOFTWARE | L15.23

23

And, in a world like .NET

- Where **several high-level languages** are made to compile into the same intermediate VM language
  - Compilers for different languages can share the same VM back-end
  - Allowing usage of common software libraries and language interoperability
  - E.g., C#, F#, VisualBasic

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | SOFTWARE | L15.24

24

The price paid for the elegance and power of the VM framework is reduced efficiency

- Naturally, a two-tier translation process results, ultimately, in generating machine code that is **more verbose** and **cumbersome**
  - Than the code produced by direct compilation
- However, as processors become faster and VM implementations more optimized
  - The degraded efficiency is hardly noticeable in most applications

What about HPC applications?

- There will always be high-performance applications and embedded systems
- These systems will continue to demand the efficient code generated by **single-tier compilers** of language like C and C++

The design of an effective VM language seeks to strike a convenient balance

- Between high-level programming languages, on the one hand
- And a great variety of low-level machine languages, on the other

The desired VM language should satisfy requirements coming both from **above** and **below** [1/2]

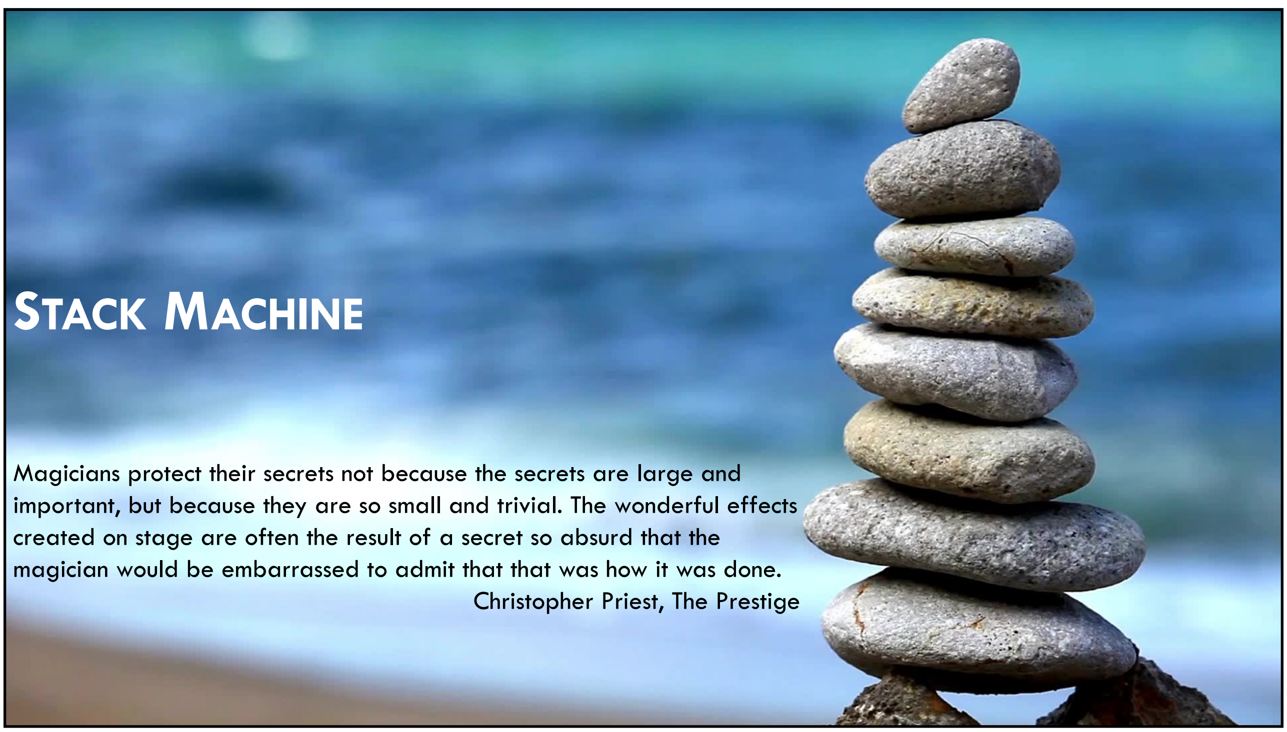
- First, the language should have a reasonable **expressive power**
  - VM languages feature arithmetic-logical commands, push/pop commands, branching commands, and function commands
  - These VM commands should be sufficiently "high" so that the VM code generated by the compiler will be reasonably elegant and well structured

The desired VM language should satisfy requirements coming both from **above** and **below** [2/2]

- At the same time, the VM commands should be sufficiently "low"
  - So that the machine code generated from them by VM translators will be **tight and efficient**
- The **translation gaps** between the high-level and the VM level & the VM level and the machine level should not be wide
  - One way to satisfy these somewhat conflicting requirements is to base the interim VM language on an abstract architecture called a **stack machine**

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | SOFTWARE | L15.29

29



30

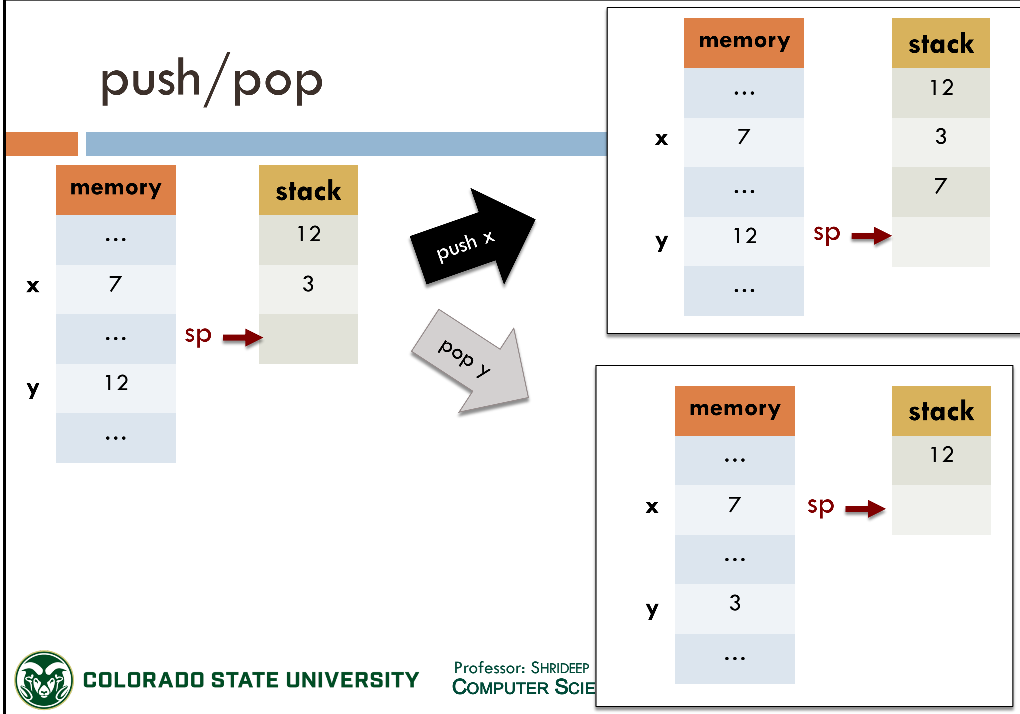
Stack machine

- The centerpiece of the stack machine model is an abstract data structure called a **stack**
- A stack is a **sequential storage space** that *grows* and *shrinks* as needed
- The stack supports various operations, the two key ones being:
  - **push**
  - **pop**

Push and Pop

- The **push** operation *adds a value* to the **top** of the stack
  - Like adding a plate to the top of a stack of plates
- The **pop** operation *removes the stack's* **top** value
  - The value that was just before it becomes the top stack element

push/pop



33

Some more about the push/pop logic

- The push/pop logic results in a **last-in-first-out (LIFO)** access logic:
  - the popped value is always the last one that was pushed onto the stack
- As it turns out, this access logic lends itself perfectly to program translation and execution purposes

34

Observe that stack access is different from conventional memory access [1/2]

- First, the **stack is accessible only from its top**
  - Whereas regular memory allows direct and indexed access to any value in the memory
- Second, reading a value from the stack is a **lossy operation**:
  - Only the top value can be read, and the only way to access it entails removing it from the stack
    - Although some stack models also provide a peek operation (reading without removing)
  - In contrast, the act of reading a value from a regular memory leaves no impact on the memory's state

Observe that stack access is different from conventional memory access [2/2]

- Lastly, writing to the stack entails **adding a value** onto the stack's top **without changing the other values** in the stack
  - In contrast, writing an item into a regular memory location is a lossy operation, since it *overrides* the location's previous value

STACK ARITHMETIC

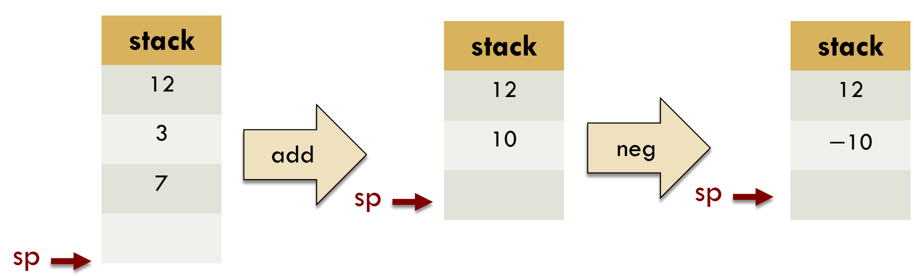
Stack arithmetic

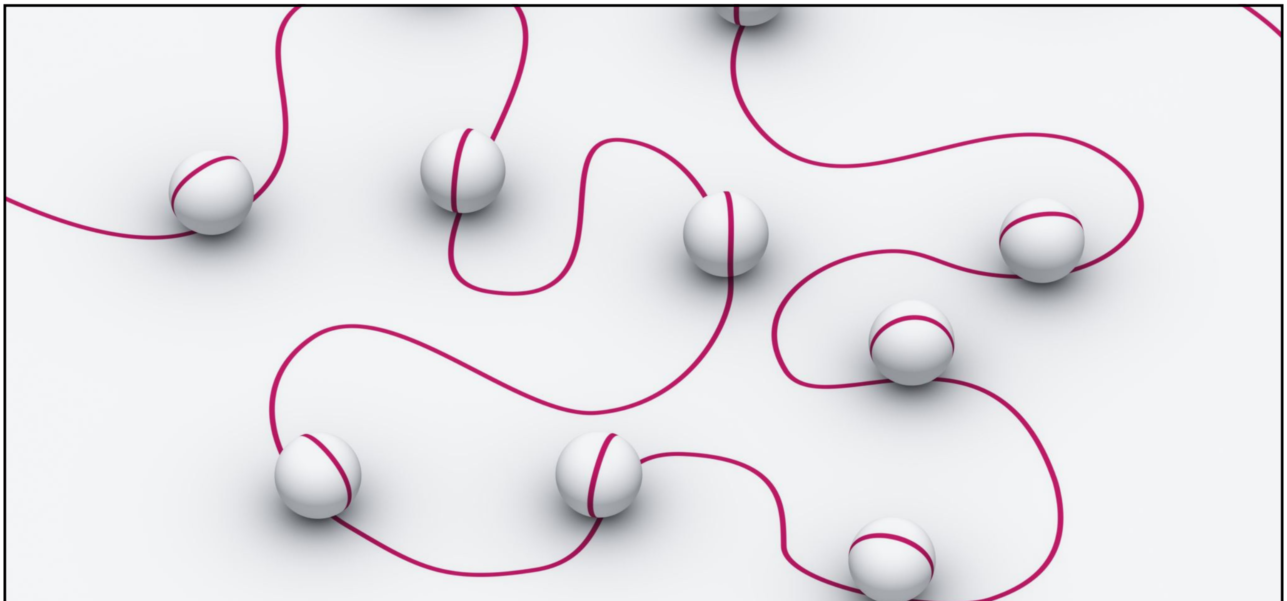
- Consider the generic operation `x op y`, where the operator `op` is applied to the operands `x` and `y`, for example, and so on
- In a stack machine, each `x op y` operation is carried out as follows:
  - the operands `x` and `y` are popped off the top of the stack
  - the value of `x op y` is computed
  - finally, the computed value is pushed onto the top of the stack

Likewise, the unary operation op x

- The unary operation `op x` is realized by
  - Popping `x` off the top of the stack
  - Computing the value of `op x`, and
  - Finally pushing this value onto the top of the stack

Simple stack arithmetic





LET'S TRY SOMETHING A LITTLE MORE COMPLICATED

41

Consider the expression d = $(2 - x) + (y + 9)$ taken from some high-level program [1/3]

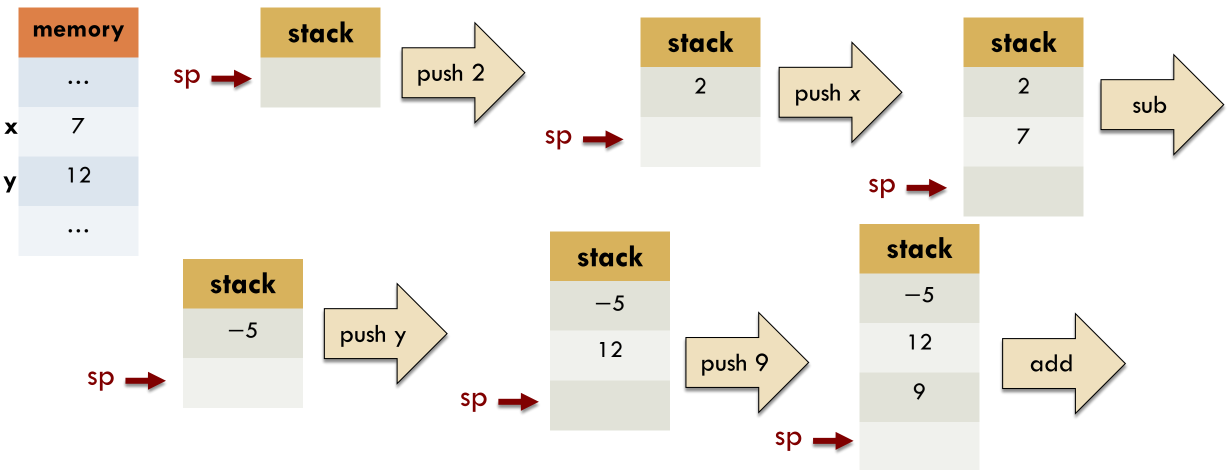
// d = $(2 - x) + (y + 9)$

- push 2
- push x
- sub
- push y
- push 9
- add
- add
- pop d

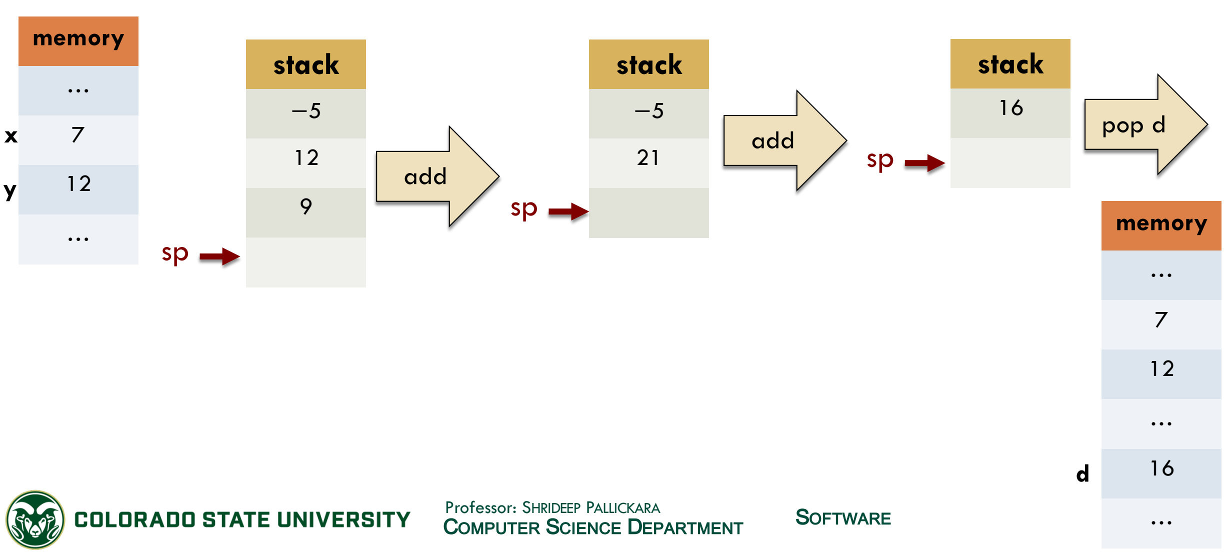
COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT SOFTWARE L15.42

42

Consider the expression d $= (2 - x) + (y + 9)$ taken from some high-level program [2/3]



Consider the expression d $= (2 - x) + (y + 9)$ taken from some high-level program [3/3]



From the stack's perspective

- Each arithmetic or logical operation has the **net impact of replacing the operation's operands** with the operation's **result**
  - Without affecting the rest of the stack
- This is similar to how humans perform mental arithmetic, using our short-term memory

For example, how do we compute $3 \text{ x } (11 + 7) - 6$

- We start by mentally popping 11 and 7 off the expression and calculating $11 + 7$
- We then plug the resulting value back into the expression, yielding $3 \text{ x } (18) - 6$
- The net effect is that $(11 + 7)$ has been replaced by 18, and **the rest of the expression remains the same as before**
  - We can now proceed to perform similar pop-compute-and-push mental operations until the expression is reduced to a single value

These examples illustrate an important virtue of stack machines

- *Any* arithmetic and logical expression— <u>no matter how complex</u>
  - Can be **systematically converted** into, and evaluated by, a sequence of simple operations on a stack
- Therefore, one can write a compiler that **translates high-level arithmetic and logical expressions** into sequences of stack commands
- Once the high-level expressions have been reduced into stack commands?
  - We can proceed to **evaluate them** using a stack machine implementation

47

48

Every computer system must specify a **run-time** model [1/2]

- This model answers essential questions without which programs cannot run:
  - How to **start** a program's execution
  - What the computer should do when a **program terminates**
  - How to **pass arguments** from one function to another
  - How to **allocate memory** resources to running functions
  - How to **free memory** resources when they are no longer needed, and so on

Every computer system must specify a **run-time** model [2/2]

- In particular, the VM translator will not only translate the VM commands (push, pop, add, and so on) into assembly instructions
- The translator will also generate assembly code that **realizes an envelope** in which the program runs

51

High-level languages allow writing programs in high-level terms

- For example, an expression like $x = -b + \sqrt{b^2 - 4ac}$
  - Can be written as $x = -b + sqrt(power(b, 2) - 4 * a * c\ )$
  - This is almost as descriptive as the real thing
- Note the difference between primitive operations like + and − and functions like `sqrt` and `power`
  - The former are **built into the basic syntax** of the high-level language
  - The latter are **extensions** of the basic language

52

Another feature of high-level languages

- The *unlimited capacity* to **extend** the language at will
- Of course, at some point, someone must implement functions; for e.g., `sqrt` and `power`

The story of **implementing** these abstractions is **completely separate** from the story of **using** them

- Application programmers can assume that each one of these functions will get executed — somehow— and …
- Following its execution, control will return — somehow —to the next operation in one's code

FUNCTIONS

Any problem in computer science can be solved with another level of indirection. Except for the problem of too many layers of indirection.
David Wheeler

55

Functions

- Every programming language is characterized by a fixed set of built-in operations
- In addition, high-level and some low-level languages offer the great freedom of **extending this fixed repertoire**
  - With an open-ended collection of **programmer-defined operations**
- Depending on the language, these canned operations are typically called subroutines, procedures, methods, or **functions**
  - We will collectively call these functions

56

Functions: the bread and butter of modular programming

- Functions are **standalone** programming units that are allowed to *call each other* for their effect
  - For example, `solve` can call `sqrt`
    - And `sqrt`, in turn, can call `power`
- This calling sequence can be as **deep** as we please, as well as **recursive**

Typically, the calling function (the caller) **passes arguments** to the called function (the callee)

- The **caller** *suspends* its execution until the callee completes its execution
- The **callee uses the passed arguments** to execute or compute something and then returns a value (which may be void) to the caller
- The caller then snaps back into action, **resuming** its execution

When one function (the caller) calls a function (the callee), someone must take care of the following:

- **Save the return address**, which is the address within the caller's code to which execution must return *after* the callee completes its execution
- **Save the memory resources** of the caller
- **Allocate** the memory resources required by the callee
- Make the **arguments** passed by the caller available to the callee's code
- **Start executing** the callee's code

When the callee terminates and returns a value, someone must take care of the following overhead:

- Make the callee's **return value** available to the caller's code
- **Recycle** the memory resources used by the callee
- **Reinstate** the previously saved memory resources of the caller
- **Retrieve** the previously saved return address
- **Resume executing** the caller's code, from the return address onward

Blissfully,

- High-level programmers don't have to ever think about all these nitty-gritty chores
- The assembly code generated by the **compiler** handles them
  - Stealthily and efficiently

In well-designed languages, built-in commands & programmer-defined functions have the same look and feel [1/2]

- For example, to compute $x + y$ using a stack machine, we push x, push y, and add
- In doing so, we expect the add implementation to pop the two top values off the stack, add them up, and push the result onto the stack

In well-designed languages, built-in commands & programmer-defined functions have the same look and feel [2/2]

- Suppose now that either we, or someone else, has written a power function designed to compute x^y
  - To use this function, we follow exactly the same routine: we push x, push y, and `call power`
- This **consistent calling protocol** allows composing primitive commands and function calls seamlessly

This consistent calling protocol allows composing primitive commands and function calls seamlessly

- For example, expressions like $(x + y)^3$ can be evaluated using
  - push x, push y, add, push 3, `call power`
- The only difference between applying a primitive operation and invoking a function is the keyword `call` preceding the latter
- Everything else is exactly the same

Applying a primitive operation and invoking a function is exactly the same …

- Both operations require the caller to **set the stage** by pushing arguments onto the stack
- Both operations are expected to **consume their arguments**, and
- Both operations are expected to **push return values** onto the stack

Computing the hypotenuse

- $\sqrt{a^2 + b^2}$

```
Function main()
    Push 3
    Push 4
    call hypot
    return

Function hypot(x,y)
    Push x
    Push x
    Call mult
    Push y
    Push y
    Call mult
    add
    Call sqrt
    Return
```

```
Function mult(x,y)
    ....
    Return

Function sqrt(num)
    ...
    Return
```

The contents of this slide-set are based on the following references

- Noam Nisan and Shimon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. 2nd Edition. ISBN-10/ ISBN-13: 0262539802 / 978-0262539807. MIT Press. [Part II, Chapter 7-8]

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT SOFTWARE L15.67

67