Chapter 2

Understanding VMs

2.1 Prerequisites

Before diving into this chapter, ensure you have access to a standard computer—whether a Mac, PC, or Linux machine—along with a text editor, a C compiler, and a Python interpreter. This chapter assumes a strong proficiency in both C and Python, as we will not cover their fundamentals. We will be working with the source code for VM1, VM2, VM3, VM4, and REGVM, which are available on GitHub. Make sure you have access to these repositories, explore them, and use them as references alongside this chapter when applicable.

2.2 Simple VMs

A *programming language virtual machine* (VM) is an abstract model that simulates a computer for executing programs. This book focuses on this specific model, as other concepts like *cloud computing* (also referred to as 'VM') do not illustrate the same principles and fall outside our scope. Henceforth, *VM* will refer to the programming language virtual machine.

The programming language virtual machines can be implemented in various forms, such as stack-based or register-based architectures. Stack machines use a simple memory structure called Last In, First Out (LIFO), which pushes and pops data during execution. This approach simplifies the execution of operations, using a method from a notation called reverse Polish notation (RPN), where operators come after the operands, making it easier to evaluate expressions without needing parentheses for operator precedence.

The virtual machine operates through an interpreter, which follows a continuous cycle of fetching instructions, identifying them, executing the corresponding

operations, and then moving to the next instruction. This fetch-execute cycle forms the backbone of how virtual machines process commands in a sequence, making them highly adaptable for different types of programs.

Note that the code provided here serves as illustrations of specific concepts and is not intended for further development. It reflects the particular perspectives being emphasised while deliberately omitting others. To start with, we will take a look at a simple example of a stack-based VM is VM1, which performs basic arithmetic operations like addition. In this machine, instructions are pushed onto a stack, executed, and then the results are either printed or the program is halted. In contrast, REGVM, a more complex register-based machine, operates directly on registers, providing more flexibility and efficiency for tasks such as loops and conditional statements. By directly accessing registers, REGVM can perform operations faster and with more control over program flow.

Virtual machines are highly portable, providing an abstraction layer that makes programs independent of the underlying hardware. This allows code to run consistently across different platforms and environments.

2.2.1 The stack

The stack works as a Last In, First Out (LIFO) memory. Below is an illustration of how the stack evolves during execution of simple operations like addition. First we have put on the stack some numbers, where they are stacked from the bottom up. First we push number ↑ '1' onto the stack:

```
1
```

Then we put number ↑ '2', pushed on to the stack:

```
1
2
```

Let's say we have pushed also ↑ '5' and after that ↑ '9' on the stack:

```
1
2
5
9
```

If we use an operator called 'addition' which takes (pops) two integers and adds them, and after the addition push ↑ the resulting number back on the stack:

```
 1
 2
14
```

The machine works with what is called reverse Polish notation (RPN), where operations are written after the operands, e.g., “5 9 +”. However in this case the addition operator isn’t included on the stack, which it sometimes can be. The RPN simplifies expression evaluation and eliminates the need for parentheses, for capturing operator precedence. See Figure 2.1.

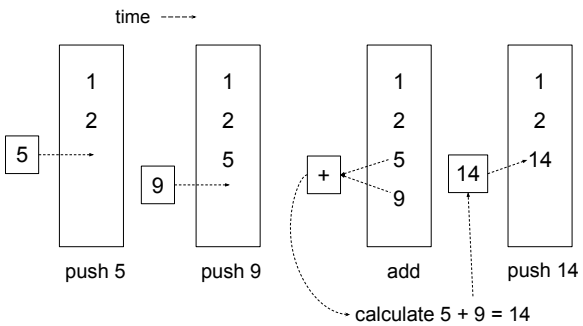


Figure 2.1: Stack.

2.2.2 Interpreter technique

A virtual machine interprets the instructions as follows:

i. Fetch the next instruction.

ii. Look up the instruction type.

iii. Execute the corresponding handler.

iv. Return to the interpreter for the next instruction (go to i).

Thus in this simple stack machine case, the interpreter fetches instructions that use the stack or operates on the stack in a sequence. See Figure 2.2.

2.2.3 VM1 implementation

So a simple stack machine should emulate a program that operates some arithmetic on numbers on a stack, e.g., A sample program in our first machine VM1, data written in a C array with support of constants for instructions:

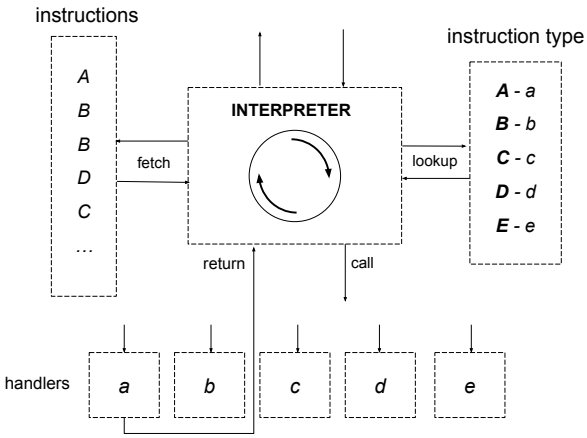


Figure 2.2: Interpreter.

```
int program[] = {
  SET, 33,
  SET, 44,
  ADD,
  PRINT,
  HALT
};
```

The instructions in this program push two numbers onto the stack (33 and 44) with instructions called 'SET', add them 'ADD', and then print the result (77) with 'PRINT'. Then it stops 'HALT'.

The corresponding operations in an "assembly" notation:

```
SET 33    ; Push 33 on stack
SET 44    ; Push 44 on stack
ADD       ; Add the top two numbers (44 + 33)
PRINT     ; Print result (77)
HALT      ; Halt program
```

To run the program, compile and execute both the virtual machine 'vm1.c' and the included hardcoded program, where ('>' marks the command line prompt):

```
> gcc vm1.c -o vm1
> ./vm1
```

```
77
> _
```

Details

The VM supports a small set of instructions, each represented by (or corresponds to) an integer opcode, the 'machine code':

- `HALT`: Stops execution.
- `SET`: Pushes a value onto the stack.
- `ADD`: Pops two values from the stack, adds them, and pushes the result back.
- `SUB`: Pops two values, subtracts them, and pushes the result back.
- `MUL`: Pops two values, multiplies them, and pushes the result back.
- `PRINT`: Pops a value from the stack and prints it.

The complete C-program can be studied on-line, see Section 2.5.

2.2.4 REGVM implementation

Let us for a moment have a look at another more complex implementation. You might be more familiar with register-based machines, if you have experience with machine code or assembly language on a typical PC. The assembly code for register machines and stack machines will naturally differ. In practice, many machines combine elements of both architectures, allowing programs to leverage the strengths of each approach. The A register can in this case be looked upon as an integer variable, from above.

Instructions pointer and program counters

In the programs 'regvm.c' and 'regvm.py' you will see how simple virtual register machines work, see Section 2.5. You are encouraged to compare the register machine and the stack machine, particularly in how they manage execution order. In this register machine, instructions are processed sequentially, with the instruction pointer advancing step by step. The pointer might change when jumps, returns etc. occurs, but it is still sequential. The fetch cycle retrieves the instruction from the current pointer location, along with any optional arguments, *and then increments the pointer* to the next instruction.

In contrast, the stack machine uses a program counter, which does not distinguish between instructions and arguments. Each instruction may be followed

by one (or maybe more arguments), which are fetched during the execution of the instruction. The program counter is first incremented, *and then the code at that location is fetched*, regardless of its type.

The differences between these machines affect how operations like jumps are handled with addresses. For example, in the register machine REGVM, the instruction pointer can be directly set to a specific address to perform a jump. In contrast, in this stack machine VM1, the program counter manages jumps by adjusting itself based on the next instruction, without distinguishing between addresses and arguments. This distinction leads to slight variations in how jumps and other control flow operations are executed.

Factorial sample

Compiling and running the C-program:

```
> gcc -o regvm regvm.c
> ./regvm
Register A: 120
> _
```

Running the Python-program:

```
> python3 regvm.py
Register A: 120
> _
```

The register A is 120 as the result from calculating the factorial of 5, which can be done in assembly:

```
1   MOV A 1
2   MOV B 5
3   CMP B 0
4   JZ 7
5   MUL A B
6   SUB B 1
7   JMP 2
8   PRINT A
```

1. **MOV A 1** Load register `A` with the value `1`. This register will store the result of the factorial.
2. **MOV B 5** Load register `B` with the value `5`. `B` will hold the current number to multiply.

3. **CMP B 0** Compare the value in `B` with `0`. This checks if the loop is complete.
4. **JZ 7** Jump to instruction 7 (`PRINT A`) if `B` equals `0`, ending the loop.
5. **MUL A B** Multiply the value in `A` by `B`, storing the result in `A`. This performs a step of the factorial calculation.
6. **SUB B 1** Subtract `1` from `B`, preparing for the next loop iteration.
7. **JMP 2** Jump back to instruction 2 (`CMP B 0`) to continue the loop until `B` equals `0`.
8. **PRINT A** Print the value in `A`, which now holds the result of the factorial calculation.

In a register-based machine, the focus shifts from manipulating a stack to direct operations on registers, which function similarly to variables. This change has a significant impact on how the machine handles instructions, particularly with branching and control flow.

A summary of instructions for the register machine REGVM can be seen in Table 2.1.

Registers and direct access

Unlike a stack machine, where data is implicitly stored and retrieved in a last-in-first-out (LIFO) manner, registers provide explicit, direct access to data. Each register has a specific name (such as `A`, `B`, etc.), and operations reference them directly. This allows the machine to:

- Perform operations without needing to push/pop from a stack.
- Store intermediate results in registers for later use.
- Efficiently execute repetitive tasks like loops or conditional operations without the overhead of stack manipulation.

In the factorial example, registers `A` and `B` act as placeholders for the running result and the current multiplier, respectively.

Comparisons and conditional jumps

An important addition to the register machine is the ability to compare register values and make decisions based on the outcome. In a stack machine, control flow may rely on the state of the stack, but in a register machine, comparisons provide more flexibility and transparency:

- CMP B 0 compares the value in register B with 0. This comparison is crucial for controlling the loop.
- JZ 7 checks the result of the comparison. If B is zero, the machine jumps to instruction 7, ending the loop and proceeding to print the result. Otherwise, the loop continues.

This comparison-based approach enables structured control flow, such as:

- Conditional jumps, where execution continues at a different point in the program if a condition is met.
- Loops, where the program can return to an earlier instruction (like JMP 2) to repeat a sequence of operations until a condition changes.

Jump instructions and control flow

In a register machine, jumps are essential for implementing loops and conditional logic. By using jumps, the machine can control which instructions to execute next based on the values in the registers:

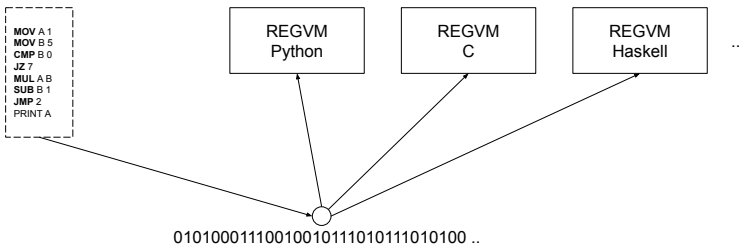
- JMP 2 sends the program counter back to instruction 2, allowing the program to repeat the multiplication and subtraction until B reaches zero.
- This type of jump loop is common in register machines, and the jump mechanism can be used to create both conditional loops (like this one) and more complex control structures (e.g., nested loops or function calls).

Instruction	Flags	Example
`MOV <reg> <val>`	-	`MOV A 1` sets A to 1.
`ADD <reg> <val>`	Z, N updated	`ADD A B` adds B to A.
`SUB <reg> <val>`	Z, N updated	`SUB A 1` subtracts 1 from A.
`MUL <reg> <val>`	Z, N updated	`MUL A B` multiplies A by B.
`CMP <reg> <val>`	Z set if values are equal	`CMP B 0` sets Z if B is 0.
`JMP <addr>`	-	`JMP 4` jumps to address 4.
`JZ <addr>`	-	`JZ 8` jumps to address 8 if Z.
`PRINT <reg>`	-	`PRINT A` prints the value in A.

Table 2.1: REGVM instructions with flags and examples

2.2.5 Portability

In the realm of software development, virtual machines play a crucial role by providing an abstraction layer that emulates a physical computer. This abstraction allows programs to run in a consistent environment that is isolated from the underlying hardware and operating system. The primary advantage of this abstraction is *code portability*—the capability of software to operate across different computing environments without requiring modification.



Here with REGVM a virtual machine abstracts the details of the host hardware by offering a uniform set of registers and an execution environment, but naturally a stack machine also follows the precise same rules as it is also portable. This kind of uniformity ensures that programs written for the VM are not tied to any specific hardware configuration, thus allowing them to be executed consistently across various platforms. For instance, by implementing a virtual machine REGVM in both C and Python, we demonstrate that code designed for the VM can run seamlessly in different virtual environments, reinforcing the concept of portability.

The use of virtual machines offers several practical benefits. It allows developers to create applications that can function across multiple operating systems or hardware platforms, eliminating the need for separate versions of the software for each environment. This capability is particularly valuable for reaching a broad user base with diverse system configurations. Moreover, VMs facilitate the running of legacy software on modern hardware, preserving access to older applications that were designed for discontinued operating systems.

In addition, VMs enhance security by providing an isolation layer between applications and the host system. This isolation helps mitigate the risk of vulnerabilities by preventing direct interactions with the host's hardware and operating system. Furthermore, the controlled environment of a VM simplifies the development and testing processes, as developers can ensure that their applications behave consistently across different systems.

Virtual machines embody an abstract machine model, which standardises the execution environment by defining a consistent set of instructions and registers. This model allows developers to focus on programming for the VM rather than dealing with hardware-specific details. Additionally, VMs often use intermediate representations (IR) of code, such as *bytecode*, which can be compiled or interpreted to run on various hardware platforms, further supporting code portability.

2.2.6 Summary

A virtual machine (VM) is essentially an abstract model of a computer that can run programs. It can be designed in various ways, such as a stack-based machine or a register-based machine. Stack machines are simpler and rely on a Last In, First Out (LIFO) memory model, whereas register machines use a set of registers for data manipulation. This section introduces the concept of virtual machines and focuses initially on stack machines, which will be explained in detail.

The stack. The stack in a virtual machine operates as a LIFO structure, where data is added and removed in a sequential manner. Instructions are executed using reverse Polish notation (RPN), meaning operators come after operands. For example, adding two numbers involves pushing them onto the stack, then popping and adding them, and finally pushing the result back onto the stack. This simple mechanism eliminates the need for managing operator precedence with parentheses.

Interpreter technique. The interpreter in a virtual machine repeatedly follows a simple fetch-execute cycle. It retrieves the next instruction, identifies its type, executes the corresponding operation, and returns to fetch the next instruction. This cycle continues until the program ends, forming the core of how a VM processes commands in sequence.

VM1 implementation. VM1 is a simple stack-based virtual machine that executes basic arithmetic instructions. A sample program pushes two numbers onto the stack, adds them, prints the result, and halts. The VM1 system translates these operations into machine instructions (opcodes), such as ‘SET‘ to push values and ‘ADD‘ to sum two numbers. The VM can be implemented in C and supports basic instructions like addition, subtraction, multiplication, and printing.

REGVM implementation. REGVM is a more complex implementation based on a register machine. Unlike a stack machine, which relies on implicit memory (LIFO stack), a register machine uses explicit registers for data storage and manipulation. Instructions are executed in sequence, and the machine directly modifies the registers. Programs in REGVM are more efficient for operations like loops or conditional statements, as registers allow for direct access to values and more flexible control flow.

Factorial sample. A sample program demonstrates how a register machine calculates the factorial of a number. It uses a loop with conditional jumps and comparisons to multiply numbers stored in registers until the factorial is computed. This showcases how register-based VMs handle iterative processes and manage control flow through direct manipulation of the instruction pointer.

Portability. Virtual machines provide an abstraction layer that enables code portability across different platforms. By standardising the execution environment, VMs like REGVM and stack machines allow programs to run consistently, regardless of the underlying hardware. This abstraction also enhances security, simplifies development, and supports legacy software. The portability of code across different environments is one of the key benefits of using virtual machines in software development.

2.3 Stack-based VM

We will now shift our focus from register-based machines back to stack-based machines. This time, however, we are working with a higher level of abstraction than the typical machine code we associate with traditional CPUs. The design of VM2 is inspired by the programming language Forth[1], which emphasizes stack-based operations. The fundamental operations of VM2 closely resemble the stack-based behavior and low-level machine language constructs found in Forth, making it a useful model for simulating high-level tasks with a stack-oriented architecture.

Fibonacci sample

In VM2 we can implement the Fibonacci series: $1, 1, 2, 3, 5, 8, 13, \ldots$

```
0 + 1 = 1
    1 + 1 = 2
        1 + 2 = 3
            2 + 3 = 5
                3 + 5 = 8
                    5 + 8 = 13
                        8 + 13 = ..
```

This using some simple basic stack-based operations:

[1] See: https://en.wikipedia.org/wiki/Forth_(programming_language). Forth has many interesting properties besides being stack-based. One such is the closeness to the machine, which we use here.

```
TWODUP
ADD
ROT
DROP
```

Running iterations of these instruction, the series can be recognised in the starting position (a), then at (e) and at (i) representing the stack after operations in two interations:

```
(a)     (b)     (c)     (d)     (e)     (f)     (g)     (h)     (i)   ..
 1       1       1       2       2       2       2       3       3
 1       1       1       1       1       1       1       2       2
         1       2       1               2       3       1
         1                               1
```

The TWODUP operation duplicates the top two stack items, ADD adds the top two numbers, ROT rotates the stack, and DROP discards the top, or the most recently pushed, item on the stack.

TWODUP operation duplicates the top two elements of the stack. This is particularly useful in stack-based languages like FORTH because it allows you to preserve values on the stack for further use in calculations. Here's how it is implemented in C:

```
case TWODUP:
    b = pop(vm);
    a = pop(vm);
    push(vm, a);
    push(vm, b);
    push(vm, a);
    push(vm, b);
    break;
```

```
(before)            (after)
  2                   2
  67                  67
  9                   9
                      67
                      9
```

ROT operation shifts the top three elements of the stack in a circular fashion. This operation allows you to reorganize the stack without disturbing the rest of the data. Specifically, it takes the third element from the top and moves it to the top, while pushing the other two elements down one position.

```
case ROT:
  c = pop(vm);
  b = pop(vm);
  a = pop(vm);
  push(vm, b);
  push(vm, c);
  push(vm, a);
  break;
```

```
(before)          (after)
  2                 67
  67                9
  9                 2
```

DROP operation discards the top element of the stack. This is useful when a value is no longer needed, and you want to reduce the stack size. The implementation is just to pop the "top" of the stack, and not doing anything with the value.

```
(before)          (after)
  2                 2
  67                67
  9
```

2.3.1 Comparisons

The comparisons in this VM2 work with the stack. They are as before used to check the relationship between two values (e.g., whether one is smaller than the other, or if two values are equal). These comparisons result in either TRUE (1) or FALSE (0) being pushed onto the stack, which is then used for making decisions, like whether to jump or not.

Less than LT:

```
case LT:
    b = pop(vm);
    a = pop(vm);
    push(vm, (a < b) ? TRUE : FALSE);
    break;
```

LT pops two values, ‘a’ and ‘b’, from the stack. It checks if ‘a’ is less than ‘b’ and pushes TRUE (1), if ‘a < b’, otherwise it pushes FALSE (0).

Equal EQ:

```
case EQ:
    b = pop(vm);
    a = pop(vm);
    push(vm, (a == b) ? TRUE : FALSE);
    break;
```

EQ pops two values, ‘a’ and ‘b’, from the stack. It checks if ‘a’ is equal to ‘b’. It pushes TRUE if ‘a == b’, otherwise it pushes FALSE.

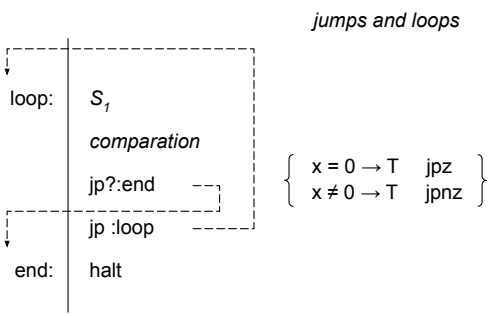
Equal to zero EQZ:

```
case EQZ:
    a = pop(vm);
    push(vm, (a == 0) ? TRUE : FALSE);
    break;
```

EQZ pops one value, ‘a’, from the stack and checks if ‘a’ is equal to zero. It pushes TRUE if ‘a == 0’, otherwise FALSE.

2.3.2 Iterations

The comparison results (TRUE or FALSE) are usually used with conditional jumps (JPZ, JPNZ). For example: If the result is TRUE, the program might jump to a different part of the code. If it’s FALSE, it will continue executing the next instructions. In this way, the VM can make decisions based on comparisons, such as repeating loops or branching to different parts of the program.



Jumps are used to change the flow of the program, allowing it to go to a different instruction rather than just moving to the next one.

Unconditional jump: JP makes the program jump directly to a specified instruction.

```
case JP:
    vm->pc = nextcode(vm); // set the program counter (pc) to a new
         address
    break;
```

This moves the program counter (pc) to a different part of the code, skipping everything in between.

Conditional jump: JPZ or JPNZ jumps only if a condition is met (e.g., if a value is zero or non-zero).

```
case JPZ:
    addr = nextcode(vm); // get the address to jump to
    v = pop(vm); // pop a value from the stack
    if (v == 0) { // jump only if the value is zero (or FALSE)
        vm->pc = addr;
    }
    break;
```

This checks a value, and if it meets the condition, the program jumps to the given address. If not, it just continues as normal.

Loops allow the program to repeat a block of code. In this VM, loops are often created by combining conditional jumps with a block of code. For example, you might pop a value, check if it's zero, and if not, jump back to repeat the block.

2.3.3 Error handling

So far, we have kept error handling to a minimum to avoid the additional code it often generates, which can be distracting when reading through the main logic. Therefore, we include this note here to address error handling considerations.

When writing software, it's *essential* to expect and plan for errors. Errors are a natural part of programming as a craft—they can happen for many reasons, such as incorrect inputs, running out of memory, or trying to access invalid memory locations. By handling errors gracefully, a program can prevent crashes and give helpful feedback instead of failing unexpectedly. Bugs can also be detected earlier in the process.

Memory allocation: If the VM fails to allocate memory (e.g., during a 'malloc()' call), it should detect this and handle the error by either exiting the program or displaying a clear error message. Example: After calling 'malloc()' to allocate memory for the stack, you should check if the returned pointer is 'NULL'. If it is, print an error and stop the program safely:

In program VM2:

```
    vm->stack = (int*) malloc(sizeof(int) * STACK_SIZE);
    if (vm->stack == NULL) {
        return NULL;
    }
```

Can be swapped for a new one:

```
    vm->stack = (int*) malloc(sizeof(int) * STACK_SIZE);
    if (vm->stack == NULL) {
        fprintf(stderr, "Error: Unable to allocate memory for the
            stack.\n");
        exit(1);
    }
```

Stack overflow or underflow: When pushing or popping from the stack, you need to ensure that you don't exceed the stack's capacity (overflow) or pop an item from an empty stack (underflow). Error handling can check the stack's boundaries and issue a warning or stop the program if these limits are violated.

```
int pop(VM* vm) {
    int sp = (vm->sp)--;
    return vm->stack[sp];
}

void push(VM* vm, int v) {
    int sp = ++(vm->sp);
    vm->stack[sp] = v;
}
```

Can be replaced by the more informative:

```
int pop(VM* vm) {
    if (vm->sp < 0) {
        fprintf(stderr, "Error: Stack underflow\n");
        exit(EXIT_FAILURE); // terminate program (or optionally
            return code)
    }
    return vm->stack[(vm->sp)--];
}

void push(VM* vm, int v) {
    if (vm->sp >= STACK_SIZE - 1) {
        fprintf(stderr, "Error: Stack overflow\n");
        exit(EXIT_FAILURE); // terminate program (or optionally
            return code)
```

```
        }
        vm->stack[++(vm->sp)] = v;
    }
```

Opcode errors: In a VM, you might run into invalid instructions or unsupported opcodes. Adding a basic check can ensure that the VM either ignores the unknown instruction or stops with an informative error message. Example: In the ‘switch‘ statement that processes opcodes, handle the default case by reporting an invalid opcode:

```
        default:
            break;
```

Replace with:

```
        default:
            fprintf(stderr, "Error: Unknown instruction at PC %d\n", vm
                ->pc);
            exit(1);
```

Common error-handling techniques

- *Return codes.* Functions can return special values (e.g., ‘-1’, ‘NULL’) to indicate that something went wrong (or had success). The calling code should check these values and respond accordingly.

- *Error messages.* When an error is detected, print a helpful message so the user or developer knows what went wrong and where to start troubleshooting.

- *Exiting gracefully.* If the program encounters a critical error that prevents further execution, use a function like ‘exit()’ to stop the program gracefully, providing a meaningful exit code.

In C and similar languages, these basic techniques serve as the foundation for error management. In languages such as C++ or Python they often rely on exception handling, and other more advanced mechanisms.

We will later on discuss error handling in another more hardware related context, see Section 4.9.

2.3.4 Summary

VM2, inspired by the stack-based language FORTH, focuses on stack operations to simulate high-level tasks. It provides a higher abstraction than traditional CPU machine code, making it effective for tasks suited to stack-oriented architectures.

Fibonacci sample. The implementation of the Fibonacci series in VM2 uses basic stack-based operations such as TWODUP, ADD, ROT, and DROP. These operations manipulate the stack to produce the Fibonacci sequence by repeatedly adding the two topmost numbers and adjusting the stack with rotation and duplication. The process is iterative, with the stack reflecting the series after each set of operations. The TWO UP duplicates the top two elements, ADD sums the top two numbers, ROT rotates the stack, and DROP removes unnecessary values.

Comparisons. VM2 implements comparisons like "less than" (LT), "equal" (EQ), and "equal to zero" (EQZ) using the stack. These operations pop values off the stack, perform the comparison, and push either TRUE (1) or FALSE (0) back onto the stack. For example, LT checks if one value is less than another, while EQ checks for equality, and EQZ checks if a value equals zero. These comparison results can be used with conditional jumps (JPZ for zero, JPNZ for non-zero), enabling control flow such as loops or decision-making in the program.

Iterations. Jumps control program flow in a virtual machine (VM). Unconditional jumps (JP) redirect the program counter (pc) to a specified address, skipping intermediate instructions. Conditional jumps (JPZ or JPNZ) occur only if a condition is met, such as jumping if a value is zero (JPZ) or non-zero (JPNZ). Loops are created by combining conditional jumps with code blocks, allowing the program to repeat instructions based on evaluated conditions (comparisons).

Error handling. Error handling is important in programming, though here minimised for simplicity. Errors like memory allocation failures (e.g., during 'malloc()') can be handled by checking if memory allocation returned 'NULL', then printing an error and exiting the program. Stack overflow or underflow is addressed by ensuring that pushes and pops stay within the stack's capacity, and providing clear error messages when limits are violated. Additionally, opcode errors are managed by checking for invalid instructions and printing informative error messages before halting execution. Common error-handling techniques include returning error codes, printing descriptive messages, and exiting gracefully to ensure a robust and fault-tolerant program.

2.4 Memory and functions

VM3, below, introduces more advanced features compared to VM1 and VM2, including jump instructions, comparison operators, and memory storage, which allow for loops, conditionals, and function calls. The machine uses *activation*

records to handle function calls and returns, storing arguments, local variables, and return addresses on the stack.

Assembly, machine code and bytecode

The role of an assembler is to translate this assembly code into machine code, which consists of binary instructions that the hardware can directly execute. This process maps the mnemonics to their corresponding binary representations, handles labels, and manages data addresses. The 'mnemonics' is here the assembly language.

However, VM3 operates differently. Instead of letting the assembly code directly be represented as some machine code (integers), the assembler converts text into what is known as *bytecode*. Bytecode is an intermediate code format used by the virtual machine. Unlike machine code, which is specific to hardware, bytecode is designed to be executed by the virtual machine, which provides a layer of abstraction between the code and the underlying hardware.

This abstraction is significant because it allows programmers to write assembly code without needing to worry about the specifics of the machine code for different hardware platforms. The assembler simplifies this process by converting the assembly language into bytecode, which the virtual machine can then interpret and execute. This approach offers greater portability and flexibility in managing code execution across various environments.

Example: Prime numbers

```
1   # prime numbers generator
2
3   INIT:
4      SETZ
5      ST 0 # c = 0
6      SET 1
7      ST 1 # j = 1
8
9   LOOP:
10     LDARG 1. # i
11     LD 1 # j
12     MOD # i % j
13     EQZ # = 0 ?
14
15     JPZ :NOINC # jump if = 0 false
16
17     LD 0  # c ->
```

```
18      INC  # c++
19      ST 0  # -> c
20
21    NOINC:
22      LD 1  # j ->
23      INC   # j++
24      ST 1  # -> j
25
26      LD 1  # j
27      LDARG 1  # i
28      LT  # j < i ?
29      JPNZ :LOOP # true, loop
30
31      LD 1  # j
32      LDARG 1  # i
33      EQ  # j = i ?
34      JPNZ :LOOP # true, loop
35
36      LD 0  # c
37      SET 2
38      EQ  # c = 2 ?
39      JPZ :EXIT # no? exit call
40
41      LDARG 1  # prime arg
42      PRINT  # print
43
44    EXIT:
45      DROP  # drop excess on stack
46      RET
47
48    START:
49      SET 99
50      STORE 0 # n = 99
51
52      SET 1
53      STORE 1 # i = 1
54
55    NEXT:
56      LOAD 1  # i
57      INC  # i++
58      DUP  # i i
59      STORE 1 # i global reg
60
61      STARG 1  # i arg for call
62      CALL :INIT  # call with 1 argument on stack
```

```
63
64      LOAD 1  # i
65      LOAD 0  # n
66      LT   # i < n ?
67      JPNZ :NEXT # true, next
68
69      LOAD 0
70      LOAD 1
71      EQ   # i = n ?
72      JPNZ :NEXT # true, next
73
74      HALT
```

The program's goal is to find and print prime numbers in the range from 1 to 'n' (99 in this case), by checking each number to see if it's prime. A number is prime if it has exactly two divisors: 1 and itself. This is accomplished by counting the number of divisors (variable 'c'), and checking if it's exactly 2 for each number.

This algorithm could be described in pseudo-code.

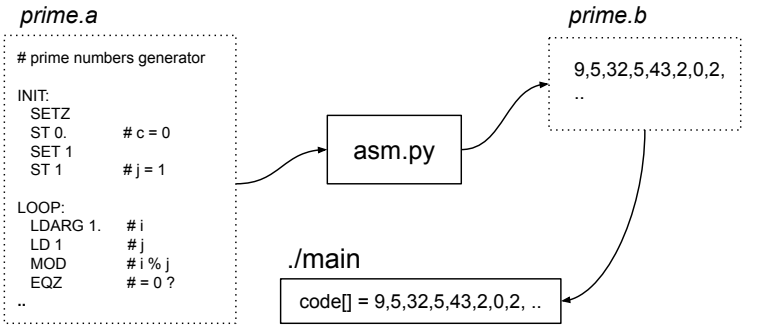
```
for i from 2 to n:
    count = 0 # init divisor count
    for j from 1 to i:
        if i % j == 0:
            count += 1 # Increment count, if j divides i
    if count == 2: # only two divisors (1 and itself)
        print(i) # i is prime, so print
```

Key sections in implementation

- INIT line 3 to 7
  - Initialises 'c ' (divisor count) and 'j' (current divisor candidate) to zero and one, respectively, before starting the loop that checks for divisibility.
- LOOP line 9 to 19
  - This section checks if 'i % j = 0'. If true, 'c' is incremented.
  - 'j' is then incremented, and the program checks whether 'j < i' to continue finding divisors.
  - When 'j = i', the loop ends, and the program checks if 'c = 2'; if so, 'i' is printed as a prime number.

- NEXT line 55 to 74
  - Increments ‘i’, checks if ‘i < n’, and repeats the process if true.

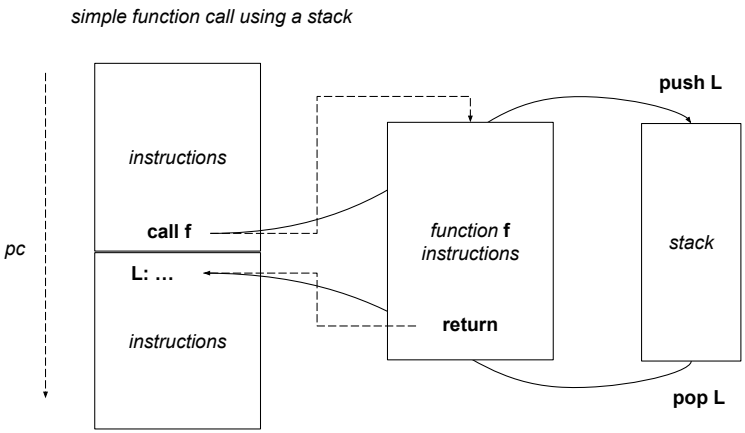
This is a straightforward algorithm for checking prime numbers, but it can be optimised in several ways for better performance. One such is skipping even numbers after 2, another is that storing and loading numbers can be reduced. We will return to optimisation later on, see Section 3.4.



Compilation and execution

The assembly of ‘prime.a’ to ‘prime.b’ and then running, works through a make file to ease development. The ‘vm3.c’ is also compiled in these steps.

```
> make FILE=prime
> python3 ./asm.py -i prime.a -o prime.b
> ./main prime.b
> loading ..
> code length = 84
> running ..
> - - - - - - - - - - - -
> 2
> 3
...
> 97
> - - - - - - - - - - - -
> duration 0.000494 seconds
> done running.
> _
```



2.4.1 Frame pointer

VM3 adds complexity to the virtual machine by supporting function calls, but also the previous loops, and conditionals. In this virtual machine implementation, the *frame pointer* (`fp`) plays a crucial role in managing function calls, particularly for local variables and arguments. Let's break down how the frame pointer is used and how it interacts with the call stack.

Calling a function: When a function is called, the VM saves the current frame pointer and program counter on the stack, establishes a new frame pointer (pointing to the current top of the stack), and then jumps to the function's code.

Returning from a function: The VM restores the previous frame pointer and program counter (return address) from the stack, effectively discarding the current stack frame and resuming execution from where the function was called.

The frame pointer is used to keep track of the current "stack frame," which contains local variables, arguments, and return addresses for a particular function call. Each function call creates a new stack frame, and the frame pointer ensures that the virtual machine can access the correct local variables and arguments relative to that frame.

Frame pointer ('fp') in the code:

```
vm->fp = 0;
```

The frame pointer is initialised to '0' when the virtual machine is created. This means the initial stack frame starts at the beginning of the stack.

Function **CALL**

```
case CALL:
    addr = nextcode(vm);
    push(vm, vm->fp); // save current frame pointer
    push(vm, vm->pc); // save return address (program counter)
    vm->fp = vm->sp; // set new frame pointer to current stack
        pointer
    vm->pc = addr; // jump to function address
    break;
```

Save current frame pointer (fp): Before calling the function, the current frame pointer is pushed onto the stack. This allows the VM to return to the previous frame after the function completes.

Save return address (pc): The return address (program counter) is also saved on the stack so the VM knows where to resume after the function returns.

Set new frame pointer (fp): The new frame pointer is set to the current stack pointer (sp). This establishes a new "stack frame" for the function, which starts where the current top of the stack is.

Jump to function: The program counter is set to the function's address, which effectively jumps to the function's instructions.

RET from function

```
case RET:
    rval = pop(vm); // get return value
    vm->sp = vm->fp; // restore stack pointer to frame pointer
    vm->pc = pop(vm); // restore the program counter (return
        address)
    vm->fp = pop(vm); // restore previous frame pointer
    push(vm, rval); // push return value onto stack
    break;
```

Restore stack pointer: The stack pointer (sp) is restored to the current frame pointer (fp), effectively removing the current stack frame.

Restore program counter: The return address (stored earlier) is popped from the stack, allowing the VM to resume execution at the point where the function was called.

Restore frame pointer: The previous frame pointer is restored, returning the VM to the calling function's stack frame.

Push return value: The function's return value is pushed onto the stack so that the calling function can use it. But first in RET, the return value is at the top of the stack, so it is has to be temporarily saved.

Frame pointers ..

- establishes a new stack frame for each function call,
- tracks the base of the current stack frame, allowing local variables and arguments to be accessed relative to it,
- allows the VM to return to the correct point in the program and restore the previous stack frame when a function completes, and
- simplifies managing function calls and returns by keeping track of the stack frame, ensuring that local variables and return addresses are correctly managed.

2.4.2 Local storage

Local storage in local variables in use by LD and ST:

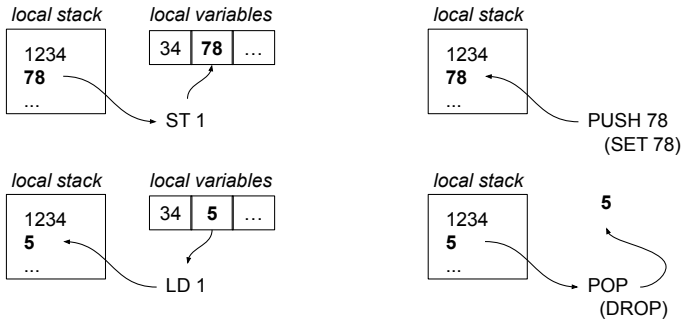
```
case LD:
   offset = nextcode(vm);
   v = vm->locals[vm->fp + (offset * OFF + OFF)];
   push(vm, v);
   break;

case ST:
   v = pop(vm);
   offset = nextcode(vm);
   vm->locals[vm->fp + (offset * OFF + OFF)] = v;
   break;
```

Local variables for the current function are accessed relative to the frame pointer. As there is a common area for the local variables, they are separated by an offset relative to `fp`. A constant OFF marks how many local variables there can be of local variables in a frame. The frame pointer marks the beginning of the stack frame for the current function. By adding an offset to the frame pointer (OFF), the VM accesses the local variable.

Push and pop operations simulate stack-based behavior typical in function calls or computations. The function 'push' pushes a value onto the frame's local stack by incrementing the stack pointer and placing the value in the stack. And

'pop' pops a value from the stack by returning the value at the top of the stack and then decrementing the stack pointer.



2.4.3 Memory management

Let's briefly discuss memory usage. Currently, we allocate fixed-size arrays for variables, local storage, and function arguments, all of which use the same size in VM3. This approach is simple to work with, not ideal for practical use. The stack can grow arbitrarily, and code may occupy redundant memory space. Given that memory is often limited, it's crucial for the virtual machine to manage memory efficiently and adaptively.

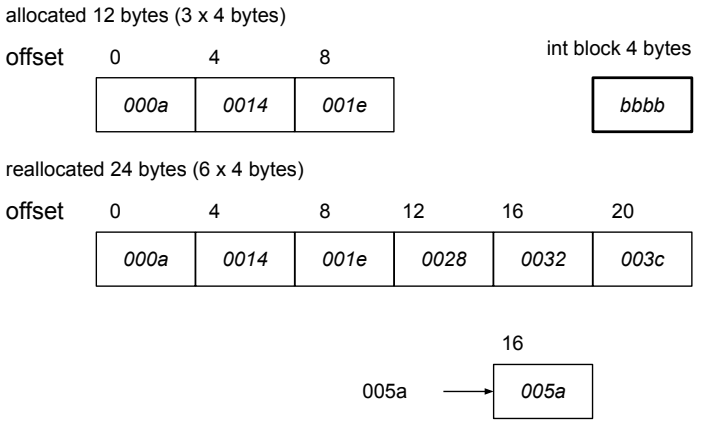
Physical machines typically have fixed, limited memory resources. However, within these constraints, the virtual machine should manage resources more efficiently. This can involve allocating, reallocating, and deallocating memory in smaller chunks. For instance, dividing memory into blocks is one way to address these constraints effectively. Here we enter the difference between *the stack* and *the heap*.

Stack Memory. This is what we have already learned. This is often where local variables, function parameters, and control flow (e.g., return addresses) are stored. The key characteristic is its Last-In-First-Out (LIFO) structure. Every time a function is called, a new "stack frame" is created. When the function returns, the frame is destroyed. An example from the VM, every time we push or pop values onto the stack, we're using this part of memory. But the stack is limited in size. Stack overflows can occur if too many functions are called recursively or too much data is pushed onto the stack without popping.

Heap Memory. This is where dynamically allocated memory is stored. It's used for variables that need to persist longer than a single function call or for data whose size isn't known in advance. Its key characteristic is that the heap is more flexible, but also more complex to manage. You must manually allocate and free memory when using the heap. An example of the VM, when you use

‘malloc()‘ to allocate arrays for the stack, arguments, or variables, you’re working with heap memory.

Imagine a memory allocation of blocks:



1. Memory allocation: allocate a block large enough to hold 3 integers. Each integer takes 4 bytes. This acts as our starting “dynamic array.”
2. Storing values: store integers at different offsets within the block, simulating array access: 10 (0x0a) at 0 offset, 20 (0x14) at 4 offset, 30 (0x1e) at 8 offset.
3. Reallocation (or expanding): when more space is needed (to store more integers), we reallocate the block to a larger size, 3 more integers: 40 (ox28), 50 (0x32) and 60 (0x3c). This is similar to how a dynamic array might double its size when full.
4. Accessing values: retrieve values from the block using the offset. We can replace at a certain offset e.g., replace 50 with 90 (0x5a) at offset 16.

An example of this approach is demonstrated in the implementation found in MEM, as detailed in the Workbook, see Section 2.5. At present, the virtual machine implementation does not include any advanced memory management features to keep the code simple.

2.4.4 Frame stack

The code illustration VM4 operates on a stack, much like the previous machines we’ve worked with. However, this VM has two distinct types of stacks. The first is the familiar data stack, which holds integers and allows operations like

addition, multiplication, and possibly other arithmetic functions. The second stack, introduced here, is dedicated to managing frames. While we've previously allocated and deallocated memory blocks for data in a more arbitrary manner, this stack helps us structure the process more effectively, specifically by dealing with frames.

Frames in this case are more complex structures (as in C 'struct') that consist of several key components: a smaller stack, a stack pointer, an array for storing the local variables' data, a slot for the return address, and a slot for the return value.

Virtual machines can often be designed with specific programming languages in mind, and in this case, the focus is on supporting function calls. Functions in programming languages can be understood as mechanisms that closely resemble the allocation and deallocation of memory on a stack.

- 'Frame' represents a single stack frame, which holds its own data stack, local variables, a return address and a return value.
- 'FrameStack' manages multiple frames, representing the call stack. Here the frame pointer tracks the top of the stack.

```
typedef struct {
    int stack[STACK_SIZE]; // data stack for frame
    int locals[LOCALS_SIZE]; // local variables
    int sp; // stack pointer
    int returnValue; // special: return value for frame
    int returnAddress; // special: return address
} Frame;

typedef struct FrameStack {
    Frame* frames[STACK_SIZE]; // stack of frames
    int fp; // frame pointer
} FrameStack;
```

Push and pop frames

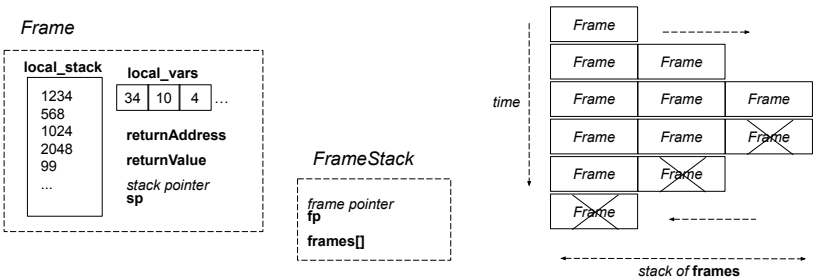
The allocation/deallocation is managed through pushing and popping frames to and from the frame stack or 'call stack'. Some bounds checking ensures that neither the stack nor the frame stack overflows or underflows.

- `pushFrame` allocates a new stack frame and pushes it onto the frame stack. It sets the internal stack pointer to -1, the return value to 0, and also the return address to 0. I at last return which frame pointer it is represented by.

- `popFrame` removes the top frame from the frame stack. Before it frees memory, the program counter is set to the return address (from an assumed function call filled in). Then it decreases the stack pointer for the frames, and returns the frame pointer number of the now deleted frame.

```
int pushFrame(VM* vm) {
    if (vm->fstack.fp >= STACK_SIZE - 1) {
        error(vm, "Frame stack overflow");
    }
    Frame* frame = (Frame*) malloc(sizeof(Frame));
    frame->sp = -1;
    frame->returnValue = 0;
    frame->returnAddress = 0;
    vm->fstack.frames[++(vm->fstack.fp)] = frame;
    return vm->fstack.fp;
}

int popFrame(VM* vm) {
    if (vm->fstack.fp < 0) {
        error(vm, "Frame stack underflow");
    }
    Frame* currentFrame = vm->fstack.frames[vm->fstack.fp];
    vm->pc = currentFrame->returnAddress;
    free(currentFrame);
    vm->fstack.fp--;
    return vm->fstack.fp + 1;
}
```



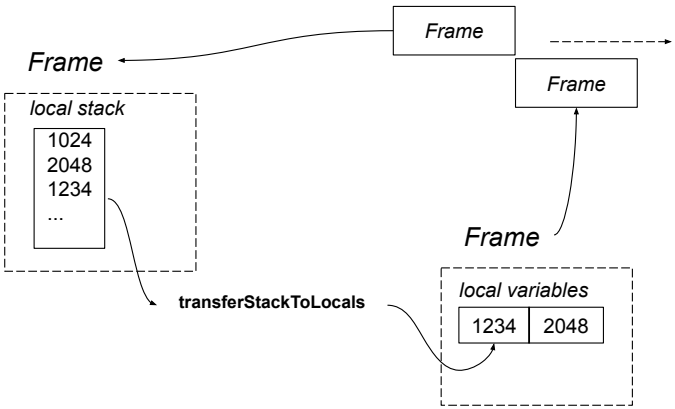
Call with or without arguments

The opcodes CALL and RET abstract *argument passing and value returning*, functioning similarly to how higher-level languages manage function calls.

The *CALL* opcode sets up a new function call. It pushes a new frame onto the frame stack, saves the current program counter as the returnAddress in the new frame, and transfers any specified arguments from the calling frame's stack to the new frame's local variables.

After this setup, CALL updates the program counter to the target address of the called function, effectively "jumping" to the start of the function.

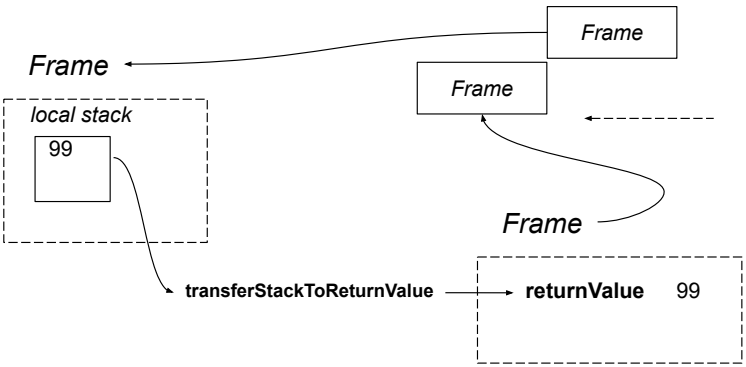
```
case CALL:
    num = next(vm);
    addr = next(vm);
    int frm = pushFrame(vm);
    fr = getFrame(vm, frm);
    fr->returnAddress = vm->pc;
    if (num > 0) {
        transferStackToLocals(vm, num);
    }
    vm->pc = addr;
    break;
```



The *CRET* opcode retrieves the return value of a function call. It pushes the returnValue stored in the current frame onto the stack. This allows the main program or a calling function to use the result of the last RET operation after a function finishes execution.

The *RET* opcode completes a function call. It optionally transfers a computed result to the calling frame's returnValue, pops the current frame from the frame stack, and restores the program counter to the stored returnAddress, allowing the program to resume from where the function was called.

```
case RET:
    if (vm->fstack.fp > 0) {
        transferStackToReturnValue(vm);
    }
    fr = vm->fstack.frames[vm->fstack.fp];
    vm->pc = fr->returnAddress;
    popFrame(vm);
    break;
```



These opcodes support recursive functions by maintaining separate frames for each function call, allowing return addresses and local data to be preserved independently.

Example: Function simulation

The factorial of a number n can be defined recursively as:

$$\text{factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times \text{factorial}(n-1), & \text{if } n > 0 \end{cases}$$

The function then can be represented in our familiar VM.

```
// Main Program
PUSH, 5, // Push the initial value n = 5
CALL, 1, 8, // Call factorial(5)
CRET, // Retrieve the result of factorial(5)
PRINT, // Print the result
HALT, // End of program
```

```
// Function factorial (Address 8)
LD, 0, // Load n (the argument)
PUSH, 1, // Push constant 1
SUB, // Subtract to check if n == 1
JZ, 28, // Jump to base case if n == 1

// Recursive Case (if n > 1)
LD, 0, // Load n again (the argument)
LD, 0, // Load n again (for factorial(n - 1))
PUSH, 1, // Push constant 1
SUB, // Calculate n - 1
CALL, 1, 8, // Call factorial(n - 1)
CRET, // Get factorial(n - 1)
MUL, // Calculate n * factorial(n - 1)
RET, // Return the result of n * factorial(n - 1)

// Base Case (Address 28)
PUSH, 1, // Push 1 (factorial of 1 is 1)
RET // Return 1
```

2.4.5 Summary

Memory and functions. VM3 introduces advanced capabilities compared to previous virtual machines (VM1 and VM2), including jump instructions, comparison operators, and memory storage that allow for loops, conditionals, and function calls. VM3 manages function calls using "activation records," which store function arguments, local variables, and return addresses on the stack.

Assembly, machine code, and bytecode. In VM3, the assembler converts assembly code into "bytecode," an intermediate format that is executed by the virtual machine. The bytecode is represented as a series of integers. Unlike machine code, bytecode adds a layer of abstraction that increases portability across hardware platforms. This process simplifies execution and makes the VM more versatile.

Prime numbers example. A sample program demonstrates how VM3 generates prime numbers from 1 to 99. The algorithm counts divisors for each number and prints the number if it has exactly two divisors (i.e., it's a prime). Although this approach works, there are potential Optimisations such as skipping even numbers after 2 or reducing redundant storing and loading operations.

Compilation and execution. The program's assembly code is compiled into bytecode using a makefile process and then executed by the VM3 engine. The prime number program runs efficiently, displaying the prime numbers within the specified range.

Frame pointer. The frame pointer is essential in VM3 for managing function calls. Each function call creates a new stack frame, and the frame pointer keeps track of local variables and arguments relative to the current frame. During a function call, the frame pointer is saved and a new frame is created. When returning from a function, the frame pointer is restored to manage the calling function's stack. The VM3 operates within the bounds of already allocated memory, so in all there is no allocation and deallocation of memory in total.

Memory management. VM3 uses fixed-size arrays for variables, local storage, and function arguments. While simple, this method is inefficient for practical use because the stack can grow arbitrarily and lead to redundant memory usage. An improved memory management strategy would involve more dynamic allocation techniques, such as reallocating memory in smaller, adaptable blocks, as demonstrated in a sample implementation in MEM.

Allocation and deallocation of frames. MEM uses dynamic memory allocation (‘malloc‘) to allocate memory for local variables and stacks at runtime. This reduces memory usage compared to static allocation at compile time. ‘allocFrame‘ and ‘deallocFrame‘ functions manage memory by allocating space for local variables and freeing it when no longer needed.

Local storage. The code handles local variables in a stack-based virtual machine using LD (load) and ST (store) instructions. Local variables are accessed relative to the frame pointer (fp), and an offset (OFF) is used to separate them within the frame. The push operation places values onto the stack, while pop retrieves them, adjusting the stack pointer accordingly. The “local” concept is logical, meaning the order of variables is managed by offsets, rather than their physical location in memory.

Frame Stack. The VM4 operates with two stacks: a data stack for computations and a frame stack for managing function calls. The frame stack handles more complex structures like local variables, a local stack, return addresses, and return values. Frames are pushed onto or popped from this frame stack as functions are called or completed.

Call with or without arguments. The CALL and RET opcodes in the virtual machine facilitate function calls by managing stack frames and program counters, supporting recursive functionality. CALL sets up a new frame, transfers arguments, and jumps to the function's address, while RET restores the return address, optionally transfers results, and deallocates the frame to resume execution.

2.5 Practice

Workbook: Chapter 2.
https://github.com/Feyerabend/bb/tree/main/workbook/ch02
Scan the QR code to access exercises, code, resources and updates.

In this chapter the focus is on exploring different implementations and techniques related to virtual machines (VMs). The chapter starts by introducing some simple VMs and provides a breakdown of their core components, including the stack and the interpreter technique. It goes on to describe the VM1 implementation and how it works, followed by an explanation of the REGVM implementation, with attention to aspects like portability. It also examines how registers work compared to stack operations in a VM.

The chapter then delves into the VM2 implementation, highlighting topics such as performance comparisons and error handling. Moving further, it introduces VM3 implementation, which incorporates more advanced concepts like the frame pointer, local storage, memory management, and the frame stack. The VM4 goes even further with especially function calls and returns.

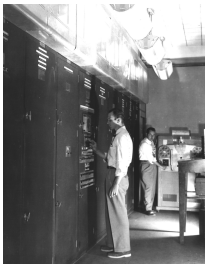
Example: Exploration exercises

1. *What is a virtual machine (VM)?* Describe the concept of a virtual machine and its purpose. Explain how it provides an abstraction layer between the software and hardware, allowing code to run in different environments.

2. *What is the history of VMs, and what has changed?* Discuss e.g., the origins of virtual machines in the 1960s. Describe the evolution of VMs from resource management on mainframes to their current applications in cloud computing and development environments.

3. *What is the difference between stack-based and register-based VMs?* Compare and contrast the architectures of stack-based and register-based virtual machines, highlighting the advantages and disadvantages of each.

Example: Project to build a von Neumann machine



The EDVAC[2](Electronic Discrete Variable Automatic Computer), completed in the late 1940s, was one of the earliest stored-program computers. It was designed by a team that included John von Neumann, who contributed a groundbreaking concept known as the von Neumann architecture. This architecture proposed that a computer's memory could store both data and instructions, allowing the computer to read and execute instructions directly from memory—a major departure from previous designs where programming was done by manually rewiring hardware.

Build you own VM

Build a simple virtual machine (VM) in your preferred programming language, whether it's Haskell, Rust, Lua, Go, or something else.

If you have a lot of experience on programming in some language other than C or Python but so far have not explored virtual machines, consider making a plan for an implementation of a VM of your own. Overall, this exercise builds a small foundation in systems programming, compilers, and computer architecture.

1. Instruction set:
    - Start by defining a minimal set of instructions (e.g., arithmetic operations, memory manipulation, jumps). Keep the instruction set simple to avoid overwhelming complexity.
    - Decide whether your VM will support stack-based or register-based instructions. A stack-based VM is easier to implement initially, while a register-based VM can be more efficient.
2. Memory model:
    - Define how your VM will handle memory. Will you implement a stack for function calls, local variables, and intermediate values? Will you use registers or memory addresses?
    - Consider how to handle global variables, heap memory, and memory bounds checking to avoid crashes.

[2]By Photo located at: http://ftp.arl.mil/ftp/historic-computers/,
Public Domain, https://commons.wikimedia.org/w/index.php?curid=1652631.

3. Errors:
    - Handle errors gracefully, such as illegal instructions, memory access violations, and stack overflows. Implement clear error messages that help debug your VM's behaviour.
    - Include handling for invalid program counter jumps, infinite loops, and incorrect use of instructions (e.g., division by zero).
4. Instruction fetch and execute cycle:
    - Ensure that your VM has a well-structured instruction fetch-decode-execute loop. This should increment the program counter after fetching each instruction and execute the corresponding operation.
    - Plan how you'll decode the instructions and their arguments (e.g., instruction format and operand handling).
5. I/O:
    - Plan for how your VM will interact with the outside world. Will it take input from files or allow console-based interaction?
    - Think about adding instructions for basic I/O operations like printing or reading from standard input.
6. File format:
    - Decide how your programs will be represented in files. You could use a simple text-based format with instructions in plain text, or a binary format that your VM can directly interpret.
    - If you're working with text-based assembly, you'll also need to write an assembler to translate human-readable code into machine code for your VM.
7. Testing and debugging:
    - Start with simple programs to test your VM, like basic arithmetic or looping. Debugging VMs can be tricky, so implement logging or a simple debugger to print the current state (e.g., the value of the program counter, stack, registers).
    - Write test cases for all instructions to catch bugs early and ensure that your VM behaves as expected.
8. Extensibility:

- Keep your design modular. As you progress, you might want to add new features like function calls, more complex data types, or additional instructions, so it's important to keep your initial design flexible enough to accommodate growth.

9. Performance:

   - In the beginning, focus on correctness rather than performance. Later, you can explore optimisations like instruction caching or efficient memory access patterns.