Impact of using stack-based and register-based bytecode on execution speed of process virtual machines

Research Question: To what extent register-based bytecode is more suitable for high-performance process virtual machines than stack-based bytecode?

A Computer Science Extended essay

Word count: 3999

Contents

1 Introduction

A virtual machine (VM) is a piece of software designed to emulate physical or abstract machine behavior. There are two types of virtual machines: system virtual machines and process virtual machines. System virtual machines emulate physical systems. Process virtual machines provide a platform-independent environment for programs to run. In this study, I will only consider process virtual machines. Examples of such virtual machines include .NET core, Java Runtime Environment, and WebAssembly runtimes for browsers (e.g. Chrome, Firefox).

Usually, applications that have low latency and high throughput requirements are written in compiled languages such as C, C++, Go, or Rust. The resulting executables are fast as they consist of a machine code optimized for a particular platform. However, the compilers for these languages usually produce binaries that are only capable of running only on the system they were compiled for, as machine code in those binaries specific to one ISA and/or uses conventions that are specific to a particular platform. For example, if the program was compiled for the x86-64 Linux target, it won't run on x86-64 Microsoft Windows or aarch64 Linux. As such, if a program should run on multiply platforms without any modifications, binaries generated by compilers (and linkers) can't be used. In this case, process virtual machines are required for cross-platform execution.

The main problem with process virtual machines is that of achieving high performance without sacrificing portability. The common solution to this problem is Just In Time compilation (JIT), meaning that platform-independent code is compiled to machine code just before this code is executed. This approach is frequently used for creating high-performant runtime environments. However, implementing JIT for every single platform is a complex undertaking, and time/money spent on supporting JIT may often outweigh speed gains. Besides, JIT compilation may be impossible on some platforms. For example, if the computer system is based on Harvard architecture (instead of Von Neuman architecture), new machine code can't be loaded at runtime, which is a requirement for JIT. On some platforms, the program needs to run with special privileges to use JIT compilation[4]. As such, unprivileged users may not be able to use the software that relies on JIT compilation.

If those concerns are of high significance, process VM should not use JIT compilation. Consequently, the question of how to increase virtual machine performance becomes even more challenging as one of the most widely used techniques for optimization can no longer be used. In such conditions, *any single implementation detail can significantly impact execution speed.*

The goal of this study is to *determine the impact of program representation on process virtual machine execution speed.* In particular, register-based bytecode and stack-based bytecode forms are tested on their impact on VM performance using a set of test programs. This is done by implementing two virtual machines that use different approaches to program representation (one will use stack-based form, another one will use register-based form) and measuring the time required for them to run test suite.

1.1 Potential applications

This study is directly connected to the performance of process virtual machines that are not using JIT compilation. That case is widely applicable to microcontroller-based systems that have to execute arbitrary code without being reprogrammed. For example, deep space satellites may make use of this technique so that navigation programs they execute can be uploaded from the space center while the core of the system that includes virtual machine and communication systems resides in more reliable ROM memory.

Another application is in the game development field. Some games allow players to change gameplay with modifications. In most cases, modifications should be cross-platform so they are available to users on different platforms. These modifications can't use machine code binaries as they are essentially unportable. Instead, the game can make use of a process virtual machine to run scripts that are used in modifications.

2 Program representation in virtual machines

In this study, only register and stack-based forms of representing programs are considered. The reason for this is that register-based and stack-based bytecode forms are the most widely used (that will be shown by a few examples later). In both of these forms, programs are sequences of *instructions*. In practice, program representation can be much more complex (For instance, in Java *.class* files, the code for each method is stored separately and additional metadata about the code is stored alongside), but in order to simplify the matter, I won't consider such cases.

Each instruction has *opcode* that specifies operation instruction should perform. Optionally, instruction will have *operands* that specify some inputs and outputs for the operation.

2.1 Requirements for program representation format

Modern virtual machines used for runtime environments include a lot of advanced features such as thread synchronization primitives and garbage collection. However, I have decided to not consider those features in this study. Instead, to simplify the implementation of VMs, I focus on the bare essentials needed to write simple procedural programs, avoiding the need for non-essential features. This is the list of operations, that virtual machines I have written for the experiment were capable of doing.

1. Basic operations on numbers and booleans. Virtual machines should be capable of adding, subtracting, multiplying, and dividing integers.

2. Storage. Both virtual machines should provide some way of accessing permanent memory.

3. Conditional jump instruction. There should be a way to express conditional and loop logic. This type of jumps is used for if statements or while loops as they need to execute different code depending on whether the condition is satisfied.

4. Unconditional jump instruction. Instruction of this kind is needed for implementing loops. When the code block with the loop body ends its execution, unconditional jump instruction is used to return back to the part in which the loop condition is evaluated and execution is dispatched with a conditional jump instruction.

5. Comparisons. Programs should be able to compare numbers and use comparison results in conditional statements.

Hence, the program representation format should be capable of representing programs with arithmetic operations, conditional jumps, unconditional jumps, and memory manipulations.

In the following two sections, I illustrate register-based and stack-based bytecode approaches to representing programs by providing examples of register-based/stack-based bytecode programs for virtual machines (VM) I have implemented and used in the experiment. The instruction set that I describe in the following two parts is hence only applicable to VMs that I have written for this investigation. However, the big picture behind stack-based/register-based bytecode is preserved across all VMs that use the same type of bytecode, so that conclusions from the analysis of program representation can be (at least to some extent) meaningful in the context of other virtual machine implementations.

3 Stack-based program representation

Stack-based instructions are used by many virtual machines such as Java Runtime Environment or .NET. In stack-based bytecode, instructions use *stack* to take inputs and to store outputs

A stack is a LIFO structure, meaning that elements that are pushed earlier will be removed from the stack later than ones pushed recently. This gives the property of locality: results of the recent operations are accessible from the top of the stack and can be used in the next operations.

For example, a stack-based program for calculating 2 * 4 - 3 may look like this.

```
PUSH 2
PUSH 4
MUL
3
SUB
```

Figure 1. $2 \times 4 - 3$ in pseudo stack-based VM bytecode

While all instructions have inputs and outputs, only PUSH instructions make use of operands in this example. PUSH instruction takes only one operand and simply pushes its value on the stack

```
; Stack: [...]
PUSH a
; Stack: [..., a]
```

Figure 2. Stack VM PUSH instruction

Arithmetic instructions pop two top values from the stack, perform some operation on them, and push the result back on the top of the stack. For MUL instruction, this operation will be a multiplication, and for SUB it will be subtraction.

```
; Stack: [..., a, b]
MUL
; Stack: [..., a * b]
```

Figure 3. Stack VM MUL instruction

Comparisons work similarly: two values are taken from the stack and the result of the comparison is pushed on the stack. For example, LT instruction pushes 1 on the stack if the top element on the stack is greater than the element underneath it and 0 otherwise.

3.1 Storage and memory

I separate storage used in stack-based bytecode programs into two main categories:

1. Variable storage. This type of storage is used to store actual variables used throughout the code
2. Indexable memory. This area of storage is used to store actual data program operates on. For example, arrays are stored in this portion of the virtual machine memory.

In my stack-based virtual machine, variable storage can be accessed with VLOAD and VSTORE instructions.

```
; Stack: [...]
PUSH 5
; Stack: [..., 5]
VSTORE 1
; Stack: [...]
VLOAD 1
VLOAD 1
; Stack: [..., 5, 5]
MUL
; Stack: [..., 25]
```

Figure 4. VLOAD/VSTORE instructions

This program stores 5 in the first variable with VSTORE 1 instruction. This instruction pops a value from the stack and saves it to the variable at index 1.

VLOAD 1 instruction performs an opposite operation: it pushes value of the variable at index 1 on the stack. In the example above, the value of this variable is loaded two times to compute square with MUL instruction.

Indexable memory can be implemented differently in virtual machines. For instance, in Java Virtual Machine (JVM), there is no indexable memory on the bytecode level. Instead, there are references to objects and arrays which JVM manages without any help from a programmer or a compiler. This ensures that valid Java bytecode will always be memory-safe. WebAssembly, on the other hand, provides a single memory array accessible from the program.

It is important to note here that usually virtual machines only offer typed memory accesses. For example, local variables in Java have to be of a certain type and array dereferences can only return the value of a type that is stored in the array itself[1]. In fact, JVM bytecode instructions for loading/storing locals or array elements have a type specified in the opcode of the instruction (JVM only has only a handful of types so it is possible to have load/store instructions for each type). In WebAssembly typing rules are not checked when general memory accesses happen (e.g. it is possible to store 32-bit float at some location and then load 32-bit integer from the same area), but all the instructions for loading from or storing to memory are also typed[2].

In virtual machines that I had implemented for this experiment, signed 32-bit integer is the only type that programs can work with. As such, I have decided to model memory as an array of 32 bit signed integers that can be accessed with MLOAD and MSTORE instructions. Those are similar to VLOAD and VSTORE, but they don't have any operands. Instead, they take memory addresses from the stack.

```
; Stack: [...]
PUSH 0x1000
; Stack: [..., 0x1000]
MLOAD
; Stack: [..., Value from the address 0x1000]
PUSH 0x2000
; Stack: [..., Value from the address 0x1000, 0x2000]
MSTORE
; Stack: [...]
```

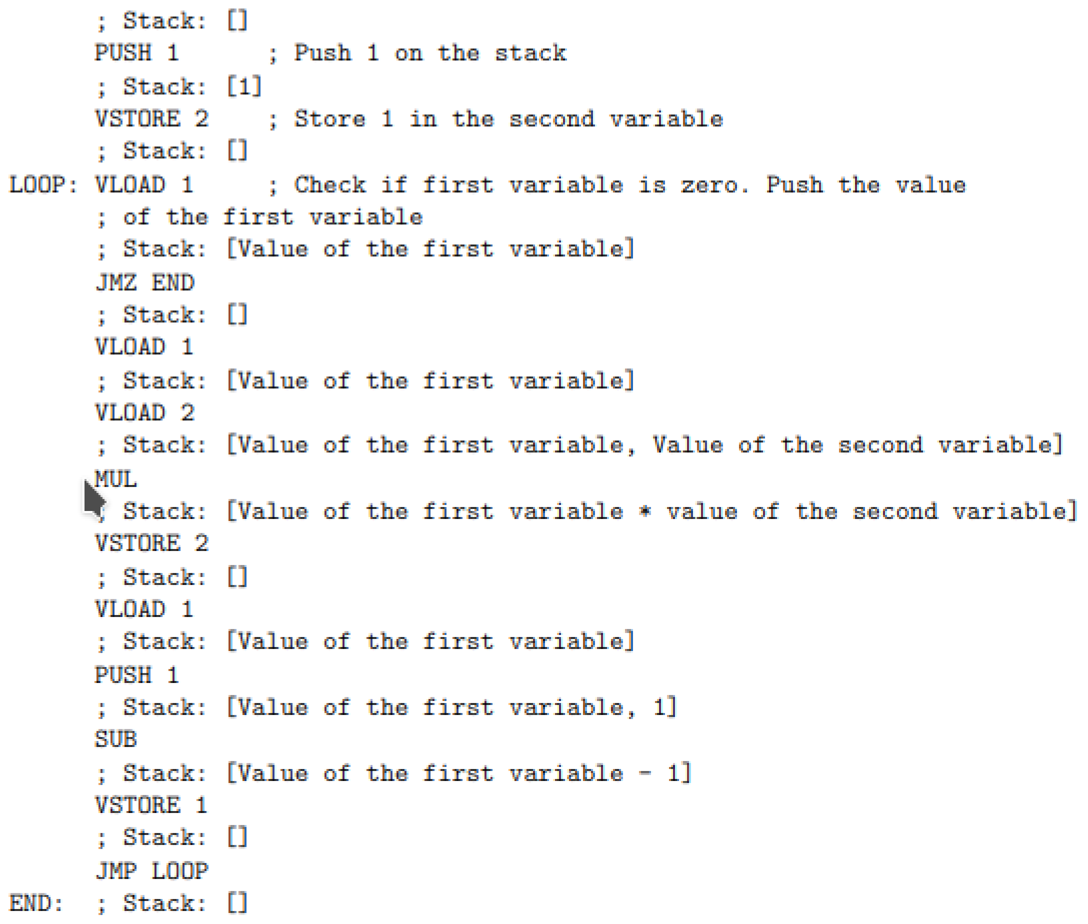
Figure 5. MLOAD/MSTORE in stack VM

This program loads the value at address 0x1000 and stores it at address 0x2000. MLOAD takes memory address, loads value at this offset, and pushes the result on the stack. MSTORE takes value and memory address and copies the value to this address.

3.2 Jumps and branching

As noted in section 2.1, there are two types of jumps: unconditional jumps and conditional jumps. In virtual machines I have implemented, there are only two instructions for jumping. JMP is an unconditional jump instruction that takes a single operand that specifies the target of the jump. JMP instruction execution is very simple - VM just sets the value of the instruction pointer to the value of the operand. JMZ instruction is quite similar, however, it first pops the value from the stack, compares it to zero, and then jumps to the target location if and only if the value compared was equal to zero, otherwise, the execution flow continues after the JMZ instruction. This type of instruction is used to implement conditional jumps.

For an example of these instructions usage, here is the factorial program, which computes the factorial of a number in the first variable and puts the result in the second variable. Note: this program computes factorial backwards ($N! = N \times (N-1) \times (N-2) \times (N-3) \times ... \times 1$)



```
      ; Stack: []
      PUSH 1      ; Push 1 on the stack
      ; Stack: [1]
      VSTORE 2    ; Store 1 in the second variable
      ; Stack: []
LOOP: VLOAD 1     ; Check if first variable is zero. Push the value
      ; of the first variable
      ; Stack: [Value of the first variable]
      JMZ END
      ; Stack: []
      VLOAD 1
      ; Stack: [Value of the first variable]
      VLOAD 2
      ; Stack: [Value of the first variable, Value of the second variable]
      MUL
      ; Stack: [Value of the first variable * value of the second variable]
      VSTORE 2
      ; Stack: []
      VLOAD 1
      ; Stack: [Value of the first variable]
      PUSH 1
      ; Stack: [Value of the first variable, 1]
      SUB
      ; Stack: [Value of the first variable - 1]
      VSTORE 1
      ; Stack: []
      JMP LOOP
END:  ; Stack: []
```

Figure 6. Factorial on Stack VM

For the sake of illustration, offset operand for JMP and JMZ instructions are written as labels. For instance, JMP LOOP instruction jumps to VLOAD 1 instruction pointed by the ”LOOP” label. JMZ END conditional jump instruction jumps to the END label if the value of the first variable is equal to zero. These two instructions are used to implement a while loop that executes its body until the factorial is computed.

4 Register-based program representation

Representing programs as a sequence of register-based instructions is less common, however, there are some notable examples of virtual machines that use register-based bytecode. For example, the reference implementation of Lua programming language uses register-based bytecode VM at its core.

In register-based instruction format, instructions inputs and outputs are always specified as operands. For example, the program for computing 2 * 4 - 3 may look like this

```
LOAD R0, 2        ; Set register R0 to 2
LOAD R1, 4        ; Set register R1 to 4
LOAD R2, 3        ; Set register R2 to 3
MUL R0, R0, R1    ; Multiply R0 by R1 and put the result in R0
SUB R0, R0, R2    ; Substract R2 from R0 and put the result in R0
```

Figure 7. 2 * 4 - 3 on register-based VM

It can be easily observed that the number of operands used increased significantly. The reason is that all inputs and outputs for instructions in register-based bytecode should be specified explicitly. In particular, these instructions work on the array of registers R0, R1, R2, ... The number of registers available can vary. In Lua, only three registers are available[5]. For the register-based VM used in the benchmark, the program can make use of up to 256 registers. The reasoning behind this register limit will be provided later on.

In the above program, arithmetic instruction has three operands. The first operand is the output operand, and the two others are input operands. SUB instruction subtracts value in the register passed as the third operand from the value in the register passed as a second operand and puts the result in the register passed as a first operand. MUL, DIV, MOD and ADD operations are only different from SUB instruction in the arithmetic operation they perform.

LOAD instruction is used to initialize some register with a given constant value. For instance, LOAD R0, 2 sets register R0 to 2.

4.1 Memory accesses

Previously, I have mentioned that there is no upper bound on the count of registers that VM can offer (the VM I had written allows to access up to 256 of them, but this limitation wasn't a problem for any of the test programs). This means that registers themselves can be used for storing variables. For instance, the computing square of R1 in the R2 register can be done like this

```
MUL R2, R1, R1  ; Multiply value in R1 by itself to
                ; compute square and put result in P2
```

Figure 8. Squaring the number

This instruction multiplies the value of R1 by itself and puts the result to R2.

Indexable memory access instructions are not very different from those in stack-based bytecode: instructions MLOAD and MSTORE are still present, but they use register operands instead of the stack for inputs/outputs.

```
MLOAD RO, R1  ; Load register RO with value in memory at offset stored in R1
MSTORE RO, R1 ; Load memory at offset stored in RO with value of R1
```

Figure 9. Memory accesses on register VM

4.2 Jumps and branches

Instructions JMP and JMZ are present in register-based bytecode as well and operate similarly. Both instructions take one parameter that is the target of the jump, and JMZ instruction has an additional operand from which value, which will be later compared to zero, is taken.

As an example, here is the listing for the factorial program written in register-based bytecode.

```
      ; RO register contains the number factorial of which should be computed
      LOAD R2, 1       ; Set R2 register to 1. (Will be used to load value 1)
      LOAD R1, 1       ; Set R1 register to 1. (Will be used to store factorial)
LOOP: JMZ END, RO      ; Jump to the END label if RO is zero
      MUL R1, R1, RO   ; Multiply R1 register by RO
      SUB RO, RO, R2   ; Substract 1 (R2 stores 1 for the whole duration) from RO
      JMP LOOP         ; LOOP again
END:  ; Factorial of the initial value of RO is computed in R1
```

Figure 10. Factorial on register VM

5 Practical investigation

In this section, I am evaluating the impact of using register-based bytecode in place of stack-based bytecode on the execution speed using an experiment. This section describes the experimental procedure and presents the results.

5.1 Experimental procedure

To evaluate register-based and stack-based forms, I have implemented two virtual machines (VMs) in the C programming language. Both VMs have a similar instruction set that was partially described in sections 3 and 4, but one VM uses a stack-based form for program representation (section 3), and another VM uses a register-based form (section 4).

The code for both virtual machines was compiled using the GNU C compiler (version 10.1.0) with the -O2 flag. The resulting 32-bit binary tests both virtual machines with benchmarks and outputs benchmarking summary.

The testing procedure is as follows:

1. Two paired programs that perform the identical operation (e.g. calculate factorial of 10'000'000) are written for two virtual machines. One program from the pair is in register-based bytecode, another one is in stack-based bytecode, but the algorithm implemented should stay the same.

2. The benchmark runs a few times to obtain accurate results and running time is averaged.

3. Running times for the benchmarks are written to standard output to be used for the analysis.

The tasks, for which pairs of programs (one for register-based VM, another for stack-based VM) were written, are listed below.

1. Calculating factorial ($n! = 1 \times 2 \times 3 \times .. \times n$) of 100000000 5 times in a row

2. Calculating 100000000th Fibonacci number 5 times in a row

3. Constructing Eratosthenes sieve for first 5000000 integers

5.2 Results

This diagram compares the performance of virtual machines on the benchmarks. Code for all benchmarks and virtual machines themselves is included in the appendix.

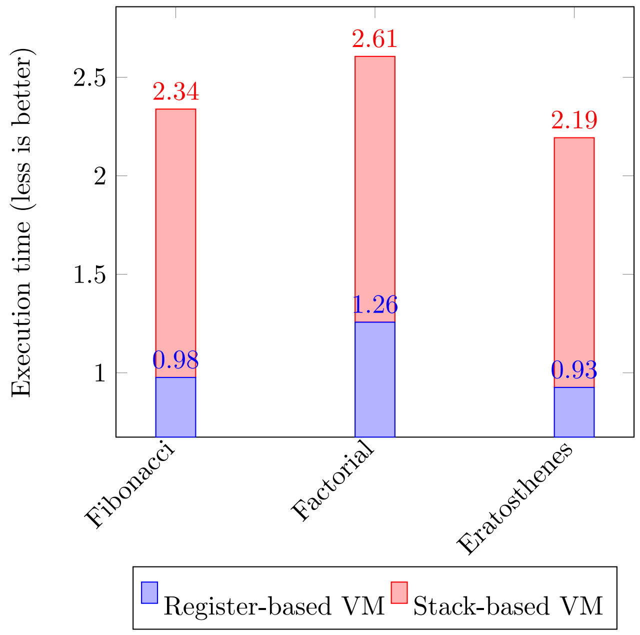


Figure 11. Benchmark results

It is apparent, that on all test programs, register-based VMs perform significantly faster. As such, an analysis of stack-based/register-based bytecode is needed to understand the reason behind the performance gap.

6 Performance analysis of stack-based and register-based machines

In this section, I research the performance implications of using stack-based/register-based code representation, by decomposing the process of executing each instruction on both virtual machines. The goal is to find an explanation for the results of the experiment.

6.1 Bytecode size

The discussion of memory size needed to store programs of these two formats is relevant to the discussion of execution speed as

1. It takes more time to fetch greater amounts of memory.
2. Programs that operate on a smaller amount of memory tend to execute faster. L1 cache on modern CPUs is somewhere near 64 KB[6], which means that if there is more code or data, less of it (in relative terms) will fit into the cache, and cache misses will be more frequent.

The size of the memory needed will depend on how instructions are stored and how many instructions are there in the given program. Stack-based bytecode instructions are smaller in size as they usually have no operands (1 operand at most). For example, ADD instruction in my implementation can consist of a simple label pointer (4 bytes on 32-bit x86), but ADD instruction in register-based bytecode needs to also encode information about registers that will be used for the operation. In my register-based virtual machine, I have limited the number of registers accessible to the program to 256 so that the index of the register can be conveniently encoded in a single byte. That makes ADD instruction in register-based bytecode 8 bytes long (4-byte label pointer + 3-byte register indices + 1-byte padding). A factor of 2 increase in size may have a significant impact on cache utilization and hence on performance.

On the other hand, programs written for register-based virtual machines usually consist of fewer instructions, as they can access variables directly. For simple expressions, stack-based VM programs need exactly as many instructions as register-based do (one for each constant used and one for each operation). However, it can be empirically stated that most programs do not have complex math operations. In most cases, the program will load and store values from variables at least as frequently as it will perform arithmetic operations. Register-based bytecode is better in this aspect as it allows to use of variables directly without VLOAD/VSTORE instructions.

6.2 Fetching and decoding costs

The first two stages of the execution cycle on CPU are the ”fetch” and ”decode” stages. This also applies to virtual machines: firstly instruction needs to be loaded from memory, then opcode and operands are decoded, and only after these steps instruction can be executed. It was already stated that stack-based instructions are largely easier to fetch and decode as they are smaller and usually have a smaller number of operands.

6.3 Dispatch costs

The important part of the instruction execution cycle is dispatch. Dispatch refers to the part of the execution cycle that is responsible for directing execution handling to the appropriate part of the VM code.

There are two main approaches to dispatch in virtual machines: switch dispatch and threaded dispatch.

In switch dispatch, instruction is loaded from program memory and the opcode of the instruction is used in the switch case statement as a parameter to direct control flow where needed.

Threaded dispatch is a faster dispatch technique[3]. It works as follows: a program is represented as an array of pointers to the parts of the code that execute those instructions. On each instruction, the element is taken from the array, and control flow just jumps to the function/label pointed by this array element.

As I am trying to make the fairest comparison, I have decided to make use of threaded dispatch for both process virtual machines. The speedup from using threaded dispatch is higher for stack-based bytecode, as it tends to have more instructions (see Bytecode size), hence dispatches are more frequent.

6.4 Execution phase duration

Likely there is no significant difference between stack-based and register-based virtual machines in terms of how long the execute phase of instructions lasts. For stack-based VM, values that instructions operate on are mostly fetched from the stack at stack pointer location. In register-based VM, instruction will fetch values from registers. The costs of fetching these values and performing identical operations are likely to be the same in both virtual machines.

7 Evaluation

The experiment showed that the performance of register-based bytecode programs surpasses that of stack-based bytecode ones significantly. In particular, register-based programs are approximately 2 -

3 times faster.

While there are numerous performance factors in favor of stack-based virtual machines, such as smaller size and lower fetching/decoding costs, those advantages are outweighed by the fact that register-based bytecode programs use fewer instructions to implement the same functionality. As test programs are quite small, running out of cache space is not an issue. The reduction in a number of instructions in test programs has hence proved to be a significant factor in favor of using register-based bytecode as a way to represent programs in process virtual machines.

That being said, there are some limitations to this study. Some of them are listed below.

1. Test programs are not really representative real-world programs, as they are small in size. This means that conclusions of the study may not be easily applicable to real-world cases of process virtual machines use.

2. Some additional instructions for stack-based bytecode like DUP (duplicate top item on the stack) or SWAP (swap two top items on the stack) may increase the performance of the stack-based virtual machines.

The first is likely to have a higher impact, as programs written for the experiment may not be a good representative of average code for big projects. For example, code that I have written might do variable load/store instructions more frequently than programs usually do on average. Additionally, bigger programs might take up more space, allowing more compact stack-based bytecode programs to win because they use less space for the code. However, for relatively small programs, the conclusion still holds.

The second point will have a smaller impact, as adding newer operations that are specific to stack-based bytecode won't affect the majority of programs. For example, DUP instruction can help a few programs where the variable is duplicated on the stack, but that is not the case with any of the test programs, soa there would not be any benefit from adding such an instruction.

8 Conclusion

The study shows that register-based program representation is much more applicable for the case of small progams if high execution speed is required. This conclusion has significance for small game plugins that run inside process virtual machines, as size of those modifications is nowhere near size of large-scale applications. On the other hand, the results of the study may not be applicable for larger programs, as the patterns in the operations bigger programs perform may be very different from the ones for programs in the test. Hence, it would not make sense to extrapolate the conclusion for larger programs.

References

[1] Chapter 6. The Java Virtual Machine Instruction Set. Accessed November 13, 2020. https://docs.oracle.com/javase/specs/jvms/se15/html/jvms-6.html.

[2] Sunfishcode. "Sunfishcode/Wasm-Reference-Manual." GitHub, May 12, 2020. https://github.com/sunfishcode/wasm-reference-manual/blob/master/WebAssembly.md.

[3] Zaleski, Mathew. "3 Dispatch Techniques ." 3 Dispatch Techniques. Accessed November 13, 2020. http://www.cs.toronto.edu/~matz/dissertation/matzDissertation-latex2html/node6.html.

[4] Apple Developer Documentation. Accessed December 14, 2020. https://developer.apple.com/documentation/bundleresources/entitlements/com_apple_security_cs_allow-jit.

[5] "Lua 5.3 Bytecode Reference." Lua 5.3 Bytecode Reference - Ravi Programming Language 0.1 documentation. Accessed December 14, 2020. https://the-ravi-programming-language.readthedocs.io/en/latest/lua_bytecode_reference.html#instruction-notation

[6] "Intel® Core™ i7-8550U Processor (8M Cache, up to 4.00 GHz) Product Specifications." (8M Cache, up to 4.00 GHz) Product Specifications. Accessed December 14, 2020. https://ark.intel.com/content/www/us/en/ark/products/123767/intel-core-i7-7820x-x-series-processor-11m-cache-up-to-4-30-ghz.html.

9 Appendix

9.1 C code for virtual machines and benchmarks

```c
#include <assert.h>  // For verifying correctness
#include <stdbool.h> // For booleans
#include <stdint.h>  // For fixed size data types
#include <stdio.h>   // For IO routines
#include <stdlib.h>  // For memory allocation/deallocation routines
#include <string.h>  // For memset
#include <time.h>    // For benchmarking

/*******************************************
 * VIRTUAL MACHINES INSTRUCTION LISTINGS *
 *******************************************/

// Represent instruction opcode in register based virtual machine
enum regs_ins_id {
        REGS_LOAD = 0U,    // LOAD R0, VAL   <=> R0 = VAL;
        REGS_ADD = 1U,     // ADD R0, R1, R2 <=> R0 = R1 + R2
        REGS_SUB = 2U,     // SUB R0, R1, R2 <=> R0 = R1 - R2
        REGS_MUL = 3U,     // MUL R0, R1, R2 <=> R0 = R1 * R2
        REGS_DIV = 4U,     // DIV R0, R1, R2 <=> R0 = R1 / R2
        REGS_REM = 5U,     // REM R0, R1, R2 <=> R0 = R1 % R2
        REGS_LT = 6U,      // LT R0, R1, R2  <=> R0 = R1 < R2
        REGS_GT = 7U,      // GT R0, R1, R2  <=> R0 = R1 > R2
        REGS_LE = 8U,      // LE R0, R1, R2  <=> R0 = R1 <= R2
        REGS_GE = 9U,      // GE R0, R1, R2  <=> R0 = R1 >= R2
        REGS_NE = 10U,     // NE R0, R1, R2  <=> R0 = R1 != R2
        REGS_EQ = 11U,     // EQ R0, R1, R2  <=> R0 = R1 == R2
        REGS_MLOAD = 12U,  // MLOAD R0, R1   <=> R0 = memory[R1]
        REGS_MSTORE = 13U, // MSTORE R0, R1  <=> memory[R0] = R1
        REGS_AND = 14U,    // AND R0, R1, R2 <=> R0 = R1 & R2
        REGS_OR = 15U,     // OR R0, R1, R2  <=> R0 = R1 | R2
        REGS_NOT = 16U,    // NOT R0, R1     <=> R0 = ~R1
        REGS_JMP = 17U,    // JMP LABEL      <=> goto LABEL
        REGS_JMZ = 18U,    // JMZ LABEL, R0  <=> if (R0 == 0) goto LABEL
        REGS_MOV = 19U,    // MOV R0, R1     <=> R0 = R1
        REGS_HALT = 20U,   // HALT Halts the programs
};

// Represent instruction opcode in stack based virtual machine
enum stack_ins_id {
        STACK_PUSH = 0U,    // [...] => [..., imm]
        STACK_ADD = 1U,     // [..., a, b] => [..., a + b]
        STACK_SUB = 2U,     // [..., a, b] => [..., a - b]
        STACK_MUL = 3U,     // [..., a, b] => [..., a * b]
        STACK_DIV = 4U,     // [..., a, b] => [..., a / b]
        STACK_REM = 5U,     // [..., a, b] => [..., a % b]
        STACK_LT = 6U,      // [..., a, b] => [..., a < b]
        STACK_GT = 7U,      // [..., a, b] => [..., a > b]
        STACK_LE = 8U,      // [..., a, b] => [..., a <= b]
        STACK_GE = 9U,      // [..., a, b] => [..., a >= b]
        STACK_NE = 10U,     // [..., a, b] => [..., a != b]
        STACK_EQ = 11U,     // [..., a, b] => [..., a == b]
        STACK_MLOAD = 12U,  // [..., a] => [..., memory[a]]
        STACK_MSTORE = 13U, // [..., a, b] => [...] & memory[a] = b
        STACK_VLOAD = 14U,  // [...] => [..., vars[imm]]
        STACK_VSTORE = 15U, // [..., a] => [...] & vars[imm] = a
        STACK_AND = 16U,    // [..., a, b] => [..., a & b]
        STACK_OR = 17,      // [..., a, b] => [..., a | b]
        STACK_NOT = 18U,    // [..., a] => [..., ~a]
        STACK_JMP = 19U,    // goto LABEL
        STACK_JMZ = 20U,    // [..., a] => [...] & if (a == 0) goto LABEL
        STACK_HALT = 21U,   // HALT Halts the program
};

/****************************************
 * VIRTUAL MACHINES CODE REPRESENTATION *
 ****************************************/

// Represents element in threaded code array for register based VM
// Can be instruction opcode or instruction data
union regs_threaded_code_slot {
        void *instruction; // Pointer to the instruction handler
        uint32_t id;       // Instruction id
        int32_t imm;       // 32 bit signed immediate
        struct {
                uint16_t target; // target of the jump
                uint8_t cond_id; // id of the register that is compared to zero. Unused
                                 // for JMP
        } jump_info; // Info about jump instruction. Used for JMP/JMZ instructions
        uint8_t reg_ids[4]; // Identifiers of registers used for the operation
};
// ADD R0, R1, R2 will be encoded in two slots as
// [instruction = &&ins_add] [reg_ids = {0, 1, 2}] last id is not used

// Represents element in threaded code array for stack based VM
// Can be instruction opcode or instruction data
union stack_threaded_code_slot {
        void *instruction;    // Pointer to the instruction handler
        uint32_t id;          // Instruction id
        int32_t imm;          // 32 bit signed immediate
        uint16_t jump_target; // Jump target
        uint8_t var_id;       // Variable identifier
};

/**********************
 * SIZE LOOKUP TABLES *
 **********************/

// Size lookup table for register based bytecode.
```

```
static const uint16_t regs_opcode_to_size[] = {
    [REGS_LOAD] = 3,  [REGS_ADD] = 2,    [REGS_SUB] = 2, [REGS_MUL] = 2,
    [REGS_DIV] = 2,   [REGS_REM] = 2,    [REGS_LT] = 2,  [REGS_GT] = 2,
    [REGS_LE] = 2,    [REGS_GE] = 2,     [REGS_NE] = 2,  [REGS_EQ] = 2,
    [REGS_MLOAD] = 2, [REGS_MSTORE] = 2, [REGS_AND] = 2, [REGS_OR] = 2,
    [REGS_NOT] = 2,   [REGS_JMP] = 2,    [REGS_JMZ] = 2, [REGS_MOV] = 2,
    [REGS_HALT] = 1};

// Size lookup table for stack based bytecode
static const uint16_t stack_opcode_to_size[] = {
    [STACK_PUSH] = 2,   [STACK_ADD] = 1,    [STACK_SUB] = 1,
    [STACK_MUL] = 1,    [STACK_DIV] = 1,    [STACK_REM] = 1,
    [STACK_LT] = 1,     [STACK_GT] = 1,     [STACK_LE] = 1,
    [STACK_GE] = 1,     [STACK_NE] = 1,     [STACK_EQ] = 1,
    [STACK_MLOAD] = 1,  [STACK_MSTORE] = 1, [STACK_VLOAD] = 2,
    [STACK_VSTORE] = 2, [STACK_AND] = 1,    [STACK_OR] = 1,
    [STACK_NOT] = 1,    [STACK_JMP] = 2,    [STACK_JMZ] = 2,
    [STACK_HALT] = 1};

/************************
 * VIRTUAL MACHINE DATA *
 ************************/

// Represents state of register based virtual machine
struct regs_vm {
        union regs_threaded_code_slot *code; // Code to be executed
        uint16_t code_size;                  // Size of the code
        int32_t registers[256];              // Virtual machine registers
        int32_t *memory;    // Pointer to the memory given to the virtual machine
        size_t memory_size; // Size of the memory given to the virtual machine
};

// Represents state of stack based virtual machine
struct stack_vm {
        union stack_threaded_code_slot *code; // Code to be executed
        uint16_t code_size;                   // Size of the code
        int32_t variables[256];               // Variables
        int32_t stack[256];                   // Stack used by the virtual machine
        int32_t *memory;    // Pointer to the memory given to the virtual machine
        size_t memory_size; // Size of the memory given to the virtual machine
};

/******************************
 * BYTECODE EXECUTION METHODS *
 ******************************/

// Execute register based bytecode
__attribute__((noinline)) void regs_vm_execute(struct regs_vm *vm) {
        // Step 1. Convert instruction ids to offsets in execute function
        // Lookup table from opcode to label
        static void *regs_opcode_to_label[] = {
            [REGS_LOAD] = &&ins_load,   [REGS_ADD] = &&ins_add,
            [REGS_SUB] = &&ins_sub,     [REGS_MUL] = &&ins_mul,
            [REGS_DIV] = &&ins_div,     [REGS_REM] = &&ins_rem,
            [REGS_LT] = &&ins_lt,       [REGS_GT] = &&ins_gt,
            [REGS_LE] = &&ins_le,       [REGS_GE] = &&ins_ge,
            [REGS_NE] = &&ins_ne,       [REGS_EQ] = &&ins_eq,
            [REGS_MLOAD] = &&ins_mload, [REGS_MSTORE] = &&ins_mstore,
            [REGS_AND] = &&ins_and,     [REGS_OR] = &&ins_or,
            [REGS_NOT] = &&ins_not,     [REGS_JMP] = &&ins_jmp,
            [REGS_JMZ] = &&ins_jmz,     [REGS_MOV] = &&ins_mov,
            [REGS_HALT] = &&ins_halt,
        };
        uint16_t ip = 0;
        while (ip < vm->code_size) {
                uint32_t opcode = vm->code[ip].id;
                vm->code[ip].instruction = regs_opcode_to_label[opcode];
                ip += regs_opcode_to_size[opcode];
        }
// Predefined macro for handling simple arithmetic operations
// Used to avoid code repetition
#define REGS_ARITH_INS_HANDLER(name, op)                                       \
        ins_##name : {                                                         \
                ++ip;                                                          \
                uint8_t r0, r1, r2;                                            \
                r0 = vm->code[ip].reg_ids[0];                                  \
                r1 = vm->code[ip].reg_ids[1];                                  \
                r2 = vm->code[ip].reg_ids[2];                                  \
                vm->registers[r0] =                                            \
                    (uint32_t)(vm->registers[r1] op vm->registers[r2]);        \
                ++ip;                                                          \
        }                                                                      \
        DISPATCH
// Step 2. Execute instructions
#define DISPATCH goto *(vm->code[ip].instruction);
        // Dispatch to the first instruction
        ip = 0;
        DISPATCH
ins_load:;
        // Load instruction
        uint32_t immediate = vm->code[++ip].imm;
        uint8_t regid = vm->code[++ip].reg_ids[0];
        vm->registers[regid] = immediate;
        ++ip;
        DISPATCH
        // Simple arithmetic instructions
        REGS_ARITH_INS_HANDLER(add, +)
        REGS_ARITH_INS_HANDLER(sub, -)
        REGS_ARITH_INS_HANDLER(mul, *)
        REGS_ARITH_INS_HANDLER(div, /)
        REGS_ARITH_INS_HANDLER(rem, %)
        REGS_ARITH_INS_HANDLER(and, &)
        REGS_ARITH_INS_HANDLER(or, |)
        REGS_ARITH_INS_HANDLER(ne, !=)
        REGS_ARITH_INS_HANDLER(eq, ==)
        REGS_ARITH_INS_HANDLER(gt, >)
        REGS_ARITH_INS_HANDLER(lt, <)
        REGS_ARITH_INS_HANDLER(ge, >=)
        REGS_ARITH_INS_HANDLER(le, <=)
// Bit inverse instruction
ins_not:;
        ++ip;
        {
                uint8_t r0, r1;
                r0 = vm->code[ip].reg_ids[0];
                r1 = vm->code[ip].reg_ids[1];
                vm->registers[r0] = ~(vm->registers[r1]);
        }
        ++ip;
        DISPATCH
// Memory load instruction
ins_mload:;
        ++ip;
        {
                uint8_t r0, r1;
                r0 = vm->code[ip].reg_ids[0];
                r1 = vm->code[ip].reg_ids[1];
                int32_t addr = vm->registers[r1];
                // Check for out of bounds access
                assert((addr >= 0) && (addr < vm->memory_size));
                // Read from memory
                int32_t fetchedValue = vm->memory[addr];
                vm->registers[r0] = fetchedValue;
        }
        ++ip;
        DISPATCH
// Memory store instruction
ins_mstore:;
        ++ip;
        {
                uint8_t r0, r1;
                r0 = vm->code[ip].reg_ids[0];
                r1 = vm->code[ip].reg_ids[1];
                int32_t addr = vm->registers[r0];
                // Check for out of bounds access
                assert((addr >= 0) && (addr < vm->memory_size));
                // Write to memory
                vm->memory[addr] = vm->registers[r1];
        }
        ++ip;
        DISPATCH
// Jump to the given location
ins_jmp:;
```

```
        ++ip;
        {
                uint16_t jump_target = vm->code[ip].jump_info.target;
                ip = jump_target;
        }
        DISPATCH
// Conditional jump to the given location
ins_jmz:;
        ++ip;
        {
                uint16_t jump_target = vm->code[ip].jump_info.target;
                uint8_t cond_id = vm->code[ip].jump_info.cond_id;
                int32_t cond_val = vm->registers[cond_id];
                if (cond_val == 0) {
                        ip = jump_target;
                } else {
                        ++ip;
                }
        }
        DISPATCH
// Mov instruction
ins_mov:;
        ++ip;
        {
                uint8_t r0, r1;
                r0 = vm->code[ip].reg_ids[0];
                r1 = vm->code[ip].reg_ids[1];
                vm->registers[r0] = vm->registers[r1];
        }
        ++ip;
        DISPATCH
ins_halt:;
        return;
// Undefine used macros
#undef REGS_ARITH_INS_HANDLER
#undef DISPATCH
}

// Execute stack based bytecode
__attribute__((noinline)) void stack_vm_execute(struct stack_vm *vm) {
        // Step 1. Convert instruction ids to offsets in execute function
        // Lookup table from opcode to label
        static void *stack_opcode_to_label[] = {
            [STACK_PUSH] = &&ins_push,   [STACK_ADD] = &&ins_add,
            [STACK_SUB] = &&ins_sub,     [STACK_MUL] = &&ins_mul,
            [STACK_DIV] = &&ins_div,     [STACK_REM] = &&ins_rem,
            [STACK_LT] = &&ins_lt,       [STACK_GT] = &&ins_gt,
            [STACK_LE] = &&ins_le,       [STACK_GE] = &&ins_ge,
            [STACK_NE] = &&ins_ne,       [STACK_EQ] = &&ins_eq,
            [STACK_MLOAD] = &&ins_mload, [STACK_MSTORE] = &&ins_mstore,
            [STACK_VLOAD] = &&ins_vload, [STACK_VSTORE] = &&ins_vstore,
            [STACK_AND] = &&ins_and,     [STACK_OR] = &&ins_or,
            [STACK_NOT] = &&ins_not,     [STACK_JMP] = &&ins_jmp,
            [STACK_JMZ] = &&ins_jmz,     [STACK_HALT] = &&ins_halt,
        };
        uint16_t ip = 0, sp = 0;
        while (ip < vm->code_size) {
                uint32_t opcode = vm->code[ip].id;
                vm->code[ip].instruction = stack_opcode_to_label[opcode];
                ip += stack_opcode_to_size[opcode];
        }
// Predefined macro for handling simple arithmetic operations
// Used to avoid code repetition
#define STACK_ARITH_INS_HANDLER(name, op)                                       \
        ins_##name:;                                                             \
        {                                                                        \
                vm->stack[sp - 2] = (int32_t)(vm->stack[sp - 2] op vm->stack[sp - 1]); \
                --sp;                                                                \
                ++ip;                                                                \
        }                                                                        \
        DISPATCH
// Step 2. Execute instruction
#define DISPATCH goto *(vm->code[ip].instruction);
        // Dispatch to the first instruction
        ip = 0;
        DISPATCH
// Push instruction
ins_push:;
        ++ip;
        {
                int32_t imm = vm->code[ip].imm;
                vm->stack[sp] = imm;
                sp++;
        }
        ++ip;
        DISPATCH
        // Arithmetic instructions
        STACK_ARITH_INS_HANDLER(add, +)
        STACK_ARITH_INS_HANDLER(sub, -)
        STACK_ARITH_INS_HANDLER(mul, *)
        STACK_ARITH_INS_HANDLER(div, /)
        STACK_ARITH_INS_HANDLER(rem, %)
        STACK_ARITH_INS_HANDLER(lt, <)
        STACK_ARITH_INS_HANDLER(gt, >)
        STACK_ARITH_INS_HANDLER(le, <=)
        STACK_ARITH_INS_HANDLER(ge, >=)
        STACK_ARITH_INS_HANDLER(ne, !=)
        STACK_ARITH_INS_HANDLER(eq, ==)
        STACK_ARITH_INS_HANDLER(and, &)
        STACK_ARITH_INS_HANDLER(or, |)
// Bit invert instruction
ins_not:;
```

```
        vm->stack[sp - 1] = ~vm->stack[sp - 1];
        ++ip;
        DISPATCH
// Memory load instruction
ins_mload:;
        {
                int32_t address = vm->stack[sp - 1];
                assert((address >= 0) && (address < vm->memory_size));
                vm->stack[sp - 1] = vm->memory[address];
        }
        ++ip;
        DISPATCH
// Memory store instruction
ins_mstore:;
        {
                int32_t val = vm->stack[sp - 1];
                int32_t address = vm->stack[sp - 2];
                sp -= 2;
                assert((address >= 0) && (address < vm->memory_size));
                vm->memory[address] = val;
        }
        ++ip;
        DISPATCH
// Variable load instruction
ins_vload:;
        ++ip;
        vm->stack[sp] = vm->variables[vm->code[ip].var_id];
        ++sp;
        ++ip;
        DISPATCH
// Variable store instruction
ins_vstore:;
        ++ip;
        vm->variables[vm->code[ip].var_id] = vm->stack[sp - 1];
        --sp;
        ++ip;
        DISPATCH
// Jump instruction
ins_jmp:;
        ++ip;
        {
                uint16_t jump_target = vm->code[ip].jump_target;
                ip = jump_target;
        }
        DISPATCH
// Conditional jump instruction
ins_jmz:;
        ++ip;
        {
                uint16_t jump_target = vm->code[ip].jump_target;
                int32_t condition = vm->stack[sp - 1];
                --sp;
                if (condition == 0) {
                        ip = jump_target;
                } else {
                        ++ip;
                }
        }
        DISPATCH
// Halt instruction
ins_halt:;
        return;
#undef STACK_ARITH_INS_HANDLER
#undef DISPATCH
}

/********************************************************
 * MACROS FOR WRITING BYTECODE IN HUMAN READABLE FORMAT *
 ********************************************************/

#define REGS_EMIT_LOAD(reg, val)                                        \
        {.id = REGS_LOAD}, {.imm = val}, {.reg_ids = {reg}},
#define REGS_EMIT_ADD(reg0, reg1, reg2)                                 \
        {.id = REGS_ADD}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_SUB(reg0, reg1, reg2)                                 \
        {.id = REGS_SUB}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_MUL(reg0, reg1, reg2)                                 \
        {.id = REGS_MUL}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_DIV(reg0, reg1, reg2)                                 \
        {.id = REGS_DIV}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_REM(reg0, reg1, reg2)                                 \
        {.id = REGS_REM}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_LT(reg0, reg1, reg2)                                  \
        {.id = REGS_LT}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_GT(reg0, reg1, reg2)                                  \
        {.id = REGS_GT}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_LE(reg0, reg1, reg2)                                  \
        {.id = REGS_LE}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_GE(reg0, reg1, reg2)                                  \
        {.id = REGS_GE}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_NE(reg0, reg1, reg2)                                  \
        {.id = REGS_NE}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_EQ(reg0, reg1, reg2)                                  \
        {.id = REGS_EQ}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_MLOAD(reg0, reg1)                                     \
        {.id = REGS_MLOAD}, {.reg_ids = {reg0, reg1}},
#define REGS_EMIT_MSTORE(reg0, reg1)                                    \
        {.id = REGS_MSTORE}, {.reg_ids = {reg0, reg1}},
#define REGS_EMIT_AND(reg0, reg1, reg2)                                 \
        {.id = REGS_AND}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_OR(reg0, reg1, reg2)                                  \
        {.id = REGS_OR}, {.reg_ids = {reg0, reg1, reg2}},
#define REGS_EMIT_NOT(reg0, reg1) {.id = REGS_NOT}, {.reg_ids = {reg0, reg1}},
#define REGS_EMIT_JMP(loc) {.id = REGS_JMP}, {.jump_info = {.target = loc}},
#define REGS_EMIT_JMZ(loc, reg0)                                              \
        {.id = REGS_JMZ}, {.jump_info = {.target = loc, .cond_id = reg0}},
#define REGS_EMIT_MOV(reg0, reg1) {.id = REGS_MOV}, {.reg_ids = {reg0, reg1}},
#define REGS_EMIT_HALT {.id = REGS_HALT},

#define STACK_EMIT_PUSH(val) {.id = STACK_PUSH}, {.imm = val},
#define STACK_EMIT_ADD {.id = STACK_ADD},
#define STACK_EMIT_SUB {.id = STACK_SUB},
#define STACK_EMIT_MUL {.id = STACK_MUL},
#define STACK_EMIT_DIV {.id = STACK_DIV},
#define STACK_EMIT_REM {.id = STACK_REM},
#define STACK_EMIT_LT {.id = STACK_LT},
#define STACK_EMIT_GT {.id = STACK_GT},
#define STACK_EMIT_LE {.id = STACK_LE},
#define STACK_EMIT_GE {.id = STACK_GE},
#define STACK_EMIT_NE {.id = STACK_NE},
#define STACK_EMIT_EQ {.id = STACK_EQ},
#define STACK_EMIT_MLOAD {.id = STACK_MLOAD},
#define STACK_EMIT_MSTORE {.id = STACK_MSTORE},
#define STACK_EMIT_VLOAD(v_id) {.id = STACK_VLOAD}, {.var_id = v_id},
#define STACK_EMIT_VSTORE(v_id) {.id = STACK_VSTORE}, {.var_id = v_id},
#define STACK_EMIT_AND {.id = STACK_AND},
#define STACK_EMIT_OR {.id = STACK_OR},
#define STACK_EMIT_NOT {.id = STACK_NOT},
#define STACK_EMIT_JMP(target) {.id = STACK_JMP}, {.jump_target = target},
#define STACK_EMIT_JMZ(target) {.id = STACK_JMZ}, {.jump_target = target},
#define STACK_EMIT_HALT {.id = STACK_HALT},

/*******************
 * BENCHMARKS CODE *
 *******************/

// Benchmarking helper
double benchmark_func(int32_t (*func)(int32_t), int32_t input, int attempts) {
        double sum = 0.0;
        for (int i = 0; i < attempts; ++i) {
                clock_t start = clock();
                int32_t result = func(input);
                clock_t time = clock() - start;
                sum += time / ((double)CLOCKS_PER_SEC);
        }
        sum /= (double)attempts;
        return sum;
}

/// Factorial ///

// Factorial on register VM
// Attribute noinline is used to prevent compiler from computing the
// result once
__attribute__((noinline)) int32_t factorial_on_regs_vm(int32_t number) {
        struct regs_vm vm;
        // Pass argument in the 0th register
        vm.registers[0] = number;
        union regs_threaded_code_slot code[] = {
            REGS_EMIT_LOAD(2, 1)    //       LOAD R2, 1
            REGS_EMIT_LOAD(1, 1)    //       LOAD R1, 1
            REGS_EMIT_JMZ(14, 0)    // LOOP: JMZ END, R0    ; LOOP is at offset 6
            REGS_EMIT_MUL(1, 1, 0) //       MUL R1, R1, R0
            REGS_EMIT_SUB(0, 0, 2) //       SUB R0, R0, R2
            REGS_EMIT_JMP(6)        //       JMP LOOP
            REGS_EMIT_HALT          // END:  HALT           ; End is at offset 14
        };
        vm.code = code;
        vm.code_size = sizeof(code) / sizeof(*code);
        regs_vm_execute(&vm);
        // The result is in the 1st register
        return vm.registers[1];
}

// Factorial on stack VM
// Attribute noinline is used to prevent compiler from computing the
// result once
__attribute__((noinline)) int32_t factorial_on_stack_vm(int32_t number) {
        struct stack_vm vm;
        // Pass argument in the first variable
        vm.variables[0] = number;
        // clang-format off
        union stack_threaded_code_slot code[] = {
            STACK_EMIT_PUSH(1)
            STACK_EMIT_VSTORE(1)
            STACK_EMIT_VLOAD(0)
            STACK_EMIT_JMZ(24)
            STACK_EMIT_VLOAD(0)
            STACK_EMIT_VLOAD(1)
            STACK_EMIT_MUL
            STACK_EMIT_VSTORE(1)
            STACK_EMIT_VLOAD(0)
            STACK_EMIT_PUSH(1)
            STACK_EMIT_SUB
            STACK_EMIT_VSTORE(0)
            STACK_EMIT_JMP(4)
            STACK_EMIT_HALT
        };
        // clang-format on
        vm.code = code;
        vm.code_size = sizeof(code) / sizeof(*code);
        stack_vm_execute(&vm);
        return vm.variables[1];
}
```

```
// Compare two VMs on the task of computing factorial
void benchmark_factorial() {
        int32_t param = 100000000;
        int attempts = 5;
        // Register based benchmark
        double time_regs_diff =
            benchmark_func(factorial_on_regs_vm, param, attempts);
        // Stack based benchmark
        double time_stack_diff =
            benchmark_func(factorial_on_stack_vm, param, attempts);
        // Output results
        printf(
            "Factorial: Stack based VM completed in %f seconds. Register based VM "
            "completed in %f seconds.\n",
            time_stack_diff, time_regs_diff);
}

/// Fibonacci series ///

// Function to compute nth fibbonacchi number using register based VM
__attribute__((noinline)) int32_t fib_on_regs_vm(int32_t number) {
        struct regs_vm vm;
        // Pass argument in the 0th register
        vm.registers[0] = number;
        // clang-format off
        union regs_threaded_code_slot code[] = {
            REGS_EMIT_LOAD(1, 1)
        REGS_EMIT_MOV(2, 1)
        REGS_EMIT_MOV(4, 1)
            REGS_EMIT_JMZ(19, 0)
        REGS_EMIT_MOV(3, 2)
        REGS_EMIT_ADD(2, 1, 2)
            REGS_EMIT_MOV(1, 3)
        REGS_EMIT_SUB(0, 0, 4)
        REGS_EMIT_JMP(7)
            REGS_EMIT_HALT
    };
        // clang-format on
        vm.code = code;
        vm.code_size = sizeof(code) / sizeof(*code);
        regs_vm_execute(&vm);
        // The result is in the 1st register
        return vm.registers[1];
}

// Function to compute nth fibbonacchi number using stack based VM
__attribute__((noinline)) int32_t fib_on_stack_vm(int32_t number) {
        struct stack_vm vm;
        vm.variables[0] = number;
        // clang-format off
        union stack_threaded_code_slot code[] = {
            STACK_EMIT_PUSH(1)
        STACK_EMIT_VSTORE(1)
        STACK_EMIT_PUSH(1)
            STACK_EMIT_VSTORE(2)
        STACK_EMIT_VLOAD(0)
        STACK_EMIT_JMZ(32)
            STACK_EMIT_VLOAD(1)
        STACK_EMIT_VLOAD(2)
            STACK_EMIT_ADD
        STACK_EMIT_VLOAD(2)
        STACK_EMIT_VSTORE(1)
            STACK_EMIT_VSTORE(2)
        STACK_EMIT_VLOAD(0)
            STACK_EMIT_PUSH(1)
            STACK_EMIT_SUB
        STACK_EMIT_VSTORE(0)
            STACK_EMIT_JMP(8)
        STACK_EMIT_HALT
    };
        // clang-format on
        vm.code = code;
        vm.code_size = sizeof(code) / sizeof(*code);
        stack_vm_execute(&vm);
        return vm.variables[1];
}

// Function to benchmark VMs using fibbonacchi number calculation task
void benchmark_fib() {
        int32_t param = 100000000;
        int attempts = 5;
        // Register based benchmark
        double time_regs_diff = benchmark_func(fib_on_regs_vm, param, attempts);
        // Stack based benchmark
        double time_stack_diff = benchmark_func(fib_on_stack_vm, param, attempts);
        // Output results
        printf("Fibbonacchi: Stack based VM completed in %f seconds. Register "
               "based VM "
               "completed in %f seconds.\n",
               time_stack_diff, time_regs_diff);
}

/// Eratosthenes Sieve ///

// Function to make Eratosthenes Sieve on register based VM
__attribute__((noinline)) int32_t
eratosthenes_sieve_on_regs_vm(int32_t number) {
        struct regs_vm vm;
        vm.memory_size = number + 1;
        vm.memory = malloc((number + 1) * 4);
        if (vm.memory == NULL) {
                assert(false);
        }
        memset(vm.memory, 0, (number + 1) * 4);
        vm.registers[0] = number;
        // clang-format off
    union regs_threaded_code_slot code[] = {
        REGS_EMIT_LOAD(3, 1)
        REGS_EMIT_LOAD(4, 0)
        REGS_EMIT_MSTORE(4, 3)
        REGS_EMIT_MSTORE(3, 3)
        REGS_EMIT_LOAD(1, 2)
        REGS_EMIT_LT(5, 1, 0)
        REGS_EMIT_JMZ(33, 5)
        REGS_EMIT_ADD(2, 1, 1)
        REGS_EMIT_LT(5, 2, 0)
        REGS_EMIT_JMZ(29, 5)
        REGS_EMIT_MSTORE(2, 3)
        REGS_EMIT_ADD(2, 2, 1)
        REGS_EMIT_JMP(19)
        REGS_EMIT_ADD(1, 1, 3)
        REGS_EMIT_JMP(13)
        REGS_EMIT_HALT
    };
        // clang-format on
        vm.code = code;
        vm.code_size = sizeof(code) / sizeof(*code);
        regs_vm_execute(&vm);
        return 0;
}

// Function to make Eratosthenes Sieve on stack based VM
__attribute__((noinline)) int32_t
eratosthenes_sieve_on_stack_vm(int32_t number) {
        struct stack_vm vm;
        vm.memory_size = number + 1;
        vm.memory = malloc((number + 1) * 4);
        if (vm.memory == NULL) {
                assert(false);
        }
        memset(vm.memory, 0, (number + 1) * 4);
        vm.variables[0] = number + 1;
        // clang-format off
    union stack_threaded_code_slot code[] = {
        STACK_EMIT_PUSH(0)
                STACK_EMIT_PUSH(1)
                STACK_EMIT_MSTORE
                STACK_EMIT_PUSH(1)
                STACK_EMIT_PUSH(1)
                STACK_EMIT_PUSH(2)
                STACK_EMIT_VSTORE(1)
                STACK_EMIT_MSTORE
                STACK_EMIT_VLOAD(1)
                STACK_EMIT_VLOAD(0)
                STACK_EMIT_LT
                STACK_EMIT_JMZ(58)
                STACK_EMIT_VLOAD(1)
                STACK_EMIT_VLOAD(1)
                STACK_EMIT_ADD
                STACK_EMIT_VSTORE(2)
                STACK_EMIT_VLOAD(2)
                STACK_EMIT_VLOAD(0)
                STACK_EMIT_LT
                STACK_EMIT_JMZ(49)
                STACK_EMIT_VLOAD(2)
                STACK_EMIT_PUSH(1)
                STACK_EMIT_MSTORE
                STACK_EMIT_VLOAD(2)
                STACK_EMIT_VLOAD(1)
                STACK_EMIT_ADD
                STACK_EMIT_VSTORE(2)
                STACK_EMIT_JMP(28)
                STACK_EMIT_VLOAD(1)
                STACK_EMIT_PUSH(1)
                STACK_EMIT_ADD
                STACK_EMIT_VSTORE(1)
                STACK_EMIT_JMP(14)
                STACK_EMIT_HALT
    };
        // clang-format on
        vm.code = code;
        vm.code_size = sizeof(code) / sizeof(*code);
        stack_vm_execute(&vm);
        return 0;
}

// Comparing two VMs in terms of execution speed using
// Eratosthenes Sieve benchmark
void benchmark_eratosthenes() {
        int32_t param = 5000000;
        int attempts = 5;
        // Register based benchmark
        double time_regs_diff =
            benchmark_func(eratosthenes_sieve_on_regs_vm, param, attempts);
        // Stack based benchmark
        double time_stack_diff =
            benchmark_func(eratosthenes_sieve_on_stack_vm, param, attempts);
        // Output results
        printf(
            "Eratosthenes Sieve: Stack based VM completed in %f seconds. Register "
            "based VM "
            "completed in %f seconds.\n",
            time_stack_diff, time_regs_diff);
```

```c
}

int main() {
        benchmark_factorial();
        benchmark_fib();
        benchmark_eratosthenes();
}
```