

Forum:[Hack Forums](#)

Thread: [Implement a simple VM \[Challenge\]](#)

yuumi - February 17, 2022, 11:17 am

I see a lot of posts regarding what people should do after learning the basics of C or asking questions regarding pointers and structures and I figured this would be a fun challenge to try.

Implement a virtual machine which supports five registers: r0,r1,r2,r3,rax
An instruction pointer register: rip

Six operational instructions:

move constant to a register

move register to register

add constant to a register

add register to register

subtract constant from a register

subtract register from a register

and the halt instruction which stops the vm execution.

the semantics of the instructions should be:

Instruction destination source

Eg:

Move rax 20

or

Move rax, r0

Where destination is always a register source can be constant or register .

and each instruction is 3 bytes.

When you've completed it, you can push this bytecode through your VM to test

```
Code uint8_t bytecode[] = {  
    kMovConst, kR0, 10,  
    kAddConst, kR0, 5,  
    kMovConst, kR1, 3,  
    kSubReg, kR0, kR1,  
    kMovConst, kR1, 4,  
    kAddReg, kR0, kR1,  
    kMovReg, kR1, kR0,  
    kSubConst, kR1, 7,  
    kMovReg, kRax, kR1,  
    kHalt  
};
```

You can use these enums

```
Code enum vm_op : uint8_t{  
    kR0,  
    kR1,  
    kR2,
```

```
kR3,  
kRax,  
kHalt,  
kMovReg,  
kMovConst,  
kAddReg,  
kAddConst,  
kSubReg,  
kSubConst,  
};
```

movss - February 17, 2022, 1:53 pm

Implement a virtual machine that doesn't use the instruction cycle, and instead uses threaded coding to execute handlers pertaining to the machine's ISA.

An example would include:

```
mov eax, ecx  
>  
VM  
>
```

tons of x86_64 opcodes that are responsible for handling your VMs opcodes.

This would be utterly confusing to the reader, and in doing so would have to reverse each "handler" and figure out what they are doing.

With dead code obfuscation, control flow obfuscation, opaque predicates this process becomes non trivial.

But easily defeated with LLVM + MCSEMA

yuumi - February 17, 2022, 2:19 pm

(Feb 17, 2022 02:53 PM)movss Wrote: [\[CITE\]](#) Implement a virtual machine that doesn't use the instruction cycle, and instead uses threaded coding to execute handlers pertaining to the machine's ISA.

An example would include:

```
mov eax, ecx  
>  
VM  
>
```

tons of x86_64 opcodes that are responsible for handling your VMs opcodes.

This would be utterly confusing to the reader, and in doing so would have to reverse each "handler" and figure out what they are doing.

With dead code obfuscation, control flow obfuscation, opaque predicates this process becomes non trivial.

But easily defeated with LLVM + MCSEMA

It isn't meant to be an obfuscation challenge. It has nothing to do with manipulating the .text section at runtime and running the executable code through a separate handler which maps VM operations to actual instruction. It's a simple C based VM like a bytecode emulator.

I'm not asking a beginner in C to write an entire obfuscation virtual machine.

movss - February 17, 2022, 2:23 pm

(Feb 17, 2022 03:19 PM)yuumi Wrote: [\[CITE\]](#) It isn't meant to be an obfuscation challenge. It has nothing to do with manipulating the .text section at runtime and running the executable code through a separate handler which maps VM operations to actual instruction. It's a simple C based VM like a bytecode emulator.

I'm not asking a beginner in C to write an entire obfuscation virtual machine.

doesn't pertain to obfuscation

obfuscation is an additive obscurity measure for the aforementioned VM design

using handlers that are threaded is far more a better learning experience than a trivial instruction cycle

yuumi - February 17, 2022, 2:38 pm

(Feb 17, 2022 03:23 PM)movss Wrote: [\[CITE\]](#) doesn't pertain to obfuscation

obfuscation is an additive obscurity measure for the aforementioned VM design

using handlers that are threaded is far more a better learning experience than a trivial instruction cycle

The point is it's meant to be a trivial instruction cycle. If people want to do it they can.

If they'd rather automate an entire function produce bytecode then compile byte code to be called and translate each instruction through a handler while the .text section of that function looks complete gibberish sure they can do that too. It wouldn't make sense to do that with what I asked, since you'd need to map almost the entire instruction set to a virtual instruction set and you'd probably need some disassembler engine to automate the process.

EmacsSucksButILikeIt - February 17, 2022, 9:04 pm

I'm a bit out of touch with C and C++ but isn't the below C++ and this is a challenge for C.

```
Code enum vm_op : uint8_t{
    kR0,
    kR1,
    kRax,
    kRdx,
    kHalt,
    kMovReg,
    kMovConst,
    kAddReg,
    kAddConst,
    kSubReg,
    kSubConst,
};
```

movss - February 17, 2022, 9:22 pm

(Feb 17, 2022 10:04 PM)EmacsSucksButILikeIt Wrote: [\[CITE\]](#) I'm a bit out of touch with C and C++ but isn't the below C++ and this is a challenge for C.

```
Code enum vm_op : uint8_t{
    kR0,
    kR1,
    kRax,
```

```
kRdx,  
kHalt,  
kMovReg,  
kMovConst,  
kAddReg,  
kAddConst,  
kSubReg,  
kSubConst,  
};  
semantics retard  
get the fuck out of this thread
```

EmacsSucksButILikeIt - February 17, 2022, 9:28 pm

(Feb 17, 2022 10:22 PM)movss Wrote: [\[CITE\]](#) semantics retard
get the fuck out of this thread

You understand this is for people that have a basic understanding of C and probably programming.... You shouldn't be posting confusing and conflicting information in your challenge.

Nice challenge BTW.

yuumi - February 17, 2022, 9:53 pm

(Feb 17, 2022 10:04 PM)EmacsSucksButILikeIt Wrote: [\[CITE\]](#) I'm a bit out of touch with C and C++ but isn't the below C++ and this is a challenge for C.

```
Code enum vm_op : uint8_t{  
    kR0,  
    kR1,  
    kRax,  
    kRdx,  
    kHalt,  
    kMovReg,  
    kMovConst,  
    kAddReg,  
    kAddConst,  
    kSubReg,  
    kSubConst,  
};
```

It's a C challenge. I guess you can use C++ but yes it's just semantics. You don't have to use those specific opcodes or even use enums.

movss - February 17, 2022, 10:20 pm

(Feb 17, 2022 10:28 PM)EmacsSucksButILikeIt Wrote: [\[CITE\]](#) You understand this is for people that have a basic understanding of C and probably programming.... You shouldn't be posting confusing and conflicting information in your challenge.

Nice challenge BTW.
bluh bluh bluh

?

EmacsSucksButILikeIt - February 18, 2022, 2:36 am

Might try this latter today.

Quote:The return value register should be rax

Does this mean you expect to support function calls?

Also... Which direction are you moving your registers. Are you using AT&T or Intel style moves.

yuumi - February 18, 2022, 6:14 am

(Feb 18, 2022 03:36 AM)EmacsSucksButILikeIt Wrote: [\[CITE\]](#) Might try this latter today.

Does this mean you expect to support function calls?

Also... Which direction are you moving your registers. Are you using AT&T or Intel style moves.

It's intel syntax

Semantic:

Instruction destination source

No it doesn't have to support function calls the result in rax portion was just something I threw in the bytecode example. You can consider that irrelevant.

Maybe in the next challenge I can make it slightly more complicated by having a virtual stack, as well as a stack register rsp and an actual calling convention.

I made some changes to the original design of the vm: have a look.

EmacsSucksButILikeIt - February 18, 2022, 7:01 am

(Feb 18, 2022 07:14 AM)yuumi Wrote: [\[CITE\]](#) It's intel syntax

Semantic:

Instruction destination source

No it doesn't have to support function calls the result in rax portion was just something I threw in the bytecode example. You can consider that irrelevant.

I'll use it(Rax) as an exit code for the byte code.

This is so easy to implement in a functional language(OCaml)... In C its a bit of a mess.

Code type instr_reg = R0 | R1 | Rax | Rdx

type op_codes =

```
  MovConstReg of (int * instr_reg)
| MovRegReg of (instr_reg * instr_reg)
| AddConstReg of (int * instr_reg)
| AddRegReg of (instr_reg * instr_reg)
| SubConstReg of (int * instr_reg)
| SubRegReg of (instr_reg * instr_reg)
| Halt
```

```
let byte_code = [
  MovConstReg(10, R0);
  AddConstReg(5, R0);
  MovConstReg(3, Rax);
  SubRegReg(R0, Rax);
  MovConstReg(4, Rax);
```

```

AddRegReg(R0, Rax);
MovRegReg(Rax, R0);
SubConstReg(7, Rax);
Halt;
]

```

```

let process_byte_code bc =
  let rax = ref 0 in
  let rdx = ref 0 in
  let r0 = ref 0 in
  let r1 = ref 0 in
  let rec get_reg r =
    match r with
    | Rax -> rax
    | Rdx -> rdx
    | R0 -> r0
    | R1 -> r1 in
  let rec process_byte_code_aux bc =
    match bc with
    | [] -> failwith "Invalid byte code sequence"
    | hd::tl ->
      match hd with
      | Halt ->
        (
          Printf.printf "rax %d, rdx %d, r0 %d, r1 %d\n" !rax !rdx !r0 !r1;
          !rax
        )
      | MovConstReg (c, r) ->
        (
          (get_reg r) := c;
          process_byte_code_aux tl
        )
      | MovRegReg (r1, r2) ->
        (
          (get_reg r1) := !(get_reg r2);
          process_byte_code_aux tl
        )
      | AddConstReg (c, r) ->
        (
          (get_reg r) := !(get_reg r) + c;
          process_byte_code_aux tl
        )
      | SubRegReg (r1, r2) ->
        (
          (get_reg r1) := !(get_reg r1) - !(get_reg r2);
          process_byte_code_aux tl
        )
      | AddRegReg(r1, r2) ->
        (
          (get_reg r1) := !(get_reg r1) + !(get_reg r2);
          process_byte_code_aux tl
        )
      | SubConstReg(c, r) ->

```

```

    (
      (get_reg r) := !(get_reg r) - c;
      process_byte_code_aux tl
    ) in
  process_byte_code_aux bc

```

```

let () =
  print_endline(string_of_int (process_byte_code byte_code))

```

The final register values. I think I got all the moves right.

Quote: rax 9, rdx 0, r0 16, r1 0

I'll see if I can get a working C version soon.

yuumi - February 18, 2022, 8:25 am

(Feb 18, 2022 08:01 AM) EmacsSucksButILikeIt Wrote: [\[CITE\]](#) I'll use it(Rax) as an exit code for the byte code.

This is so easy to implement in a functional language(OCaml)... In C its a bit of a mess.

```

Code type instr_reg = R0 | R1 | Rax | Rdx

```

```

type op_codes =
  MovConstReg of (int * instr_reg)
| MovRegReg of (instr_reg * instr_reg)
| AddConstReg of (int * instr_reg)
| AddRegReg of (instr_reg * instr_reg)
| SubConstReg of (int * instr_reg)
| SubRegReg of (instr_reg * instr_reg)
| Halt

```

```

let byte_code = [
  MovConstReg(10, R0);
  AddConstReg(5, R0);
  MovConstReg(3, Rax);
  SubRegReg(R0, Rax);
  MovConstReg(4, Rax);
  AddRegReg(R0, Rax);
  MovRegReg(Rax, R0);
  SubConstReg(7, Rax);
  Halt;
]

```

```

let process_byte_code bc =
  let rax = ref 0 in
  let rdx = ref 0 in
  let r0 = ref 0 in
  let r1 = ref 0 in
  let rec get_reg r =
    match r with
    | Rax -> rax
    | Rdx -> rdx
    | R0 -> r0
    | R1 -> r1 in

```

```

let rec process_byte_code_aux bc =
  match bc with
  | [] -> failwith "Invalid byte code sequence"
  | hd::tl ->
    match hd with
    | Halt ->
      (
        Printf.printf "rax %d, rdx %d, r0 %d, r1 %d\n" !rax !rdx !r0 !r1;
        !rax
      )
    | MovConstReg (c, r) ->
      (
        (get_reg r) := c;
        process_byte_code_aux tl
      )
    | MovRegReg (r1, r2) ->
      (
        (get_reg r1) := !(get_reg r2);
        process_byte_code_aux tl
      )
    | AddConstReg (c, r) ->
      (
        (get_reg r) := !(get_reg r) + c;
        process_byte_code_aux tl
      )
    | SubRegReg (r1, r2) ->
      (
        (get_reg r1) := !(get_reg r1) - !(get_reg r2);
        process_byte_code_aux tl
      )
    | AddRegReg(r1, r2) ->
      (
        (get_reg r1) := !(get_reg r1)+ !(get_reg r2);
        process_byte_code_aux tl
      )
    | SubConstReg(c, r) ->
      (
        (get_reg r) := !(get_reg r) - c;
        process_byte_code_aux tl
      ) in
    process_byte_code_aux bc

```

```

let () =
  print_endline(string_of_int (process_byte_code byte_code))

```

The final register values. I think I got all the moves right.

I'll see if I can get a working C version soon.

I did it in C which was relatively easy but that could just be because of my experience with C. Just to make it compatible to future changes , you can get rid of the rdx, and rcx register. Instead we'll just use registers r0,r1,r2,r3 and rax

0x9F - February 18, 2022, 11:43 am

Did this in around 40 minutes (dont mind the dirt)

enum/struct defs:

```
Code enum vm_reg : unsigned char {
    eR0,
    eR1,
    eRAX,
    eRDX,
};
```

```
enum vm_op : unsigned char {
    eHALT,
    eMOVER,
    eMOVEC,
    eADDR,
    eADDC,
    eSUBR,
    eSUBC,
};
```

```
typedef struct reg_container_t {
    DWORD R0;
    DWORD R1;
    DWORD RAX;
    DWORD RDX;
    DWORD RIP;
}reg_container;
```

mini-vm.c including execute/handler:

```
Code #define MAX_OPCODE_STACK_SIZE 64
```

```
struct reg_container_t* vm_registers = NULL;
```

```
void handle(vm_op operation, BYTE reg, BYTE v);
```

```
DWORD execute(const BYTE* data)
{
    BYTE* base = (BYTE*)data;
    vm_registers->RIP = 0;
```

```
    do
    {
        vm_registers->RIP++;
```

```
        BYTE op = *data++;
        BYTE reg = *data++;
        BYTE v = *data++;
```

```
        handle((vm_op)op, reg, v);
```

```
        BYTE next = *(base + (3 * vm_registers->RIP));
```

```
        if (next == eHALT)
            break;
```

```

} while (vm_registers->RIP < MAX_OPCODE_STACK_SIZE);

return vm_registers->RAX;
}

void handle(vm_op operation, BYTE reg, BYTE v)
{
    DWORD* dest = (DWORD*)((DWORD)vm_registers + (sizeof(DWORD) * reg));

    switch (operation)
    {
        case eMOVEC:
            *dest = v;
            break;

        case eMOVE:
            *dest = *(DWORD*)((DWORD)vm_registers + (sizeof(DWORD) * v));
            break;

        case eADDC:
            *dest += v;
            break;

        case eADDR:
            *dest += *(DWORD*)((DWORD)vm_registers + (sizeof(DWORD) * v));
            break;

        case eSUBC:
            *dest -= v;
            break;

        case eSUBR:
            *dest -= *(DWORD*)((DWORD)vm_registers + (sizeof(DWORD) * v));
            break;

        default:
            printf("Unknown instruction: 0x%x\r\n", operation);
            break;
    }
}

void initalize_vm()
{
    dispose_vm();
    vm_registers = (reg_container_t*)malloc(sizeof(reg_container_t));
    ZeroMemory(vm_registers, sizeof(reg_container_t));
}

void dispose_vm()
{
    if (vm_registers != NULL)
    {

```

```

free(vm_registers);
vm_registers = NULL;
}
}

```

example usage:

```

Code int main()
{
    BYTE bytecode[] = {
        eMOVEC, eR0, 10,
        eADDC, eR0, 5,
        eMOVEC, eRAX, 3,
        eSUBR, eR0, eRAX,
        eMOVEC, eRAX, 4,
        eADDR, eR0, eRAX,
        eMOVER, eRAX, eR0,
        eSUBC, eRAX, 7,
        eHALT
    };

```

```

    initalize_vm();

```

```

    DWORD result = execute(bytecode);

```

```

    printf("rax: %d (0x%x)\r\n", result, result);

```

```

    dispose_vm();

```

```

    return 0;
}

```

Haven't tested this extensively and probably forgot small things but you get the point

yuumi - February 18, 2022, 12:41 pm

(Feb 18, 2022 12:43 PM)0x9F Wrote: [\[CITE\]](#) Haven't tested this extensively and probably forgot small things but you get the point

i tested it but the results print 0 for me. It should print 9 judging from the bytecode provided. Also just to make it more portable it would be better to use fixed types such as uint8_t, uint32_t, uint64_t or uintptr_t .

Is Code DWORD* dest = (DWORD*)((DWORD)vm_registers + (sizeof(DWORD) * reg)); meant to be the register that's being targeted?

And this line Code } while (vm_registers->RIP < MAX_OPCODE_STACK_SIZE); Is that meant to be maximum number of instructions/addressable code? I think it would be better to just let it run until halt is executed. But I'm unsure why it's called STACK_SIZE

Overall it just needs to be a little cleaner but I can see the code working in concept. It's similar to how I went about doing it.

github.com/yuumikazoomi/vm

I did however make slight changes to the vm by getting rid of rcx, rdx and making the registers r0,r1,r2,r3 and rax. In the future I wanted to make it more challenging by adding a call instruction including a virtual stack with a calling convention hence the rax.

0x9F - February 18, 2022, 1:02 pm

(Feb 18, 2022 01:41 PM)yuumi Wrote: [\[CITE\]](#) i tested it but the results print 0 for me. It should print 9 judging from the bytecode provided. Also just to make it more portable it would be better to use fixed types such as uint8_t, uint32_t, uint64_t or uintptr_t .

Is

Code `DWORD* dest = (DWORD*)((DWORD)vm_registers + (sizeof(DWORD) * reg));` meant to be the register that's being targeted?

And this line

Code `} while (vm_registers->RIP < MAX_OPCODE_STACK_SIZE);` Is that meant to be maximum number of instructions/addressable code? I think it would be better to just let it run until halt is executed. But I'm unsure why it's called `STACK_SIZE`

Overall it just needs to be a little cleaner but I can see the code working in concept. It's similar to how I went about doing it.

github.com/yuumikazoomi/vm

I did however make slight changes to the vm by getting rid of rcx, rdx and making the registers r0,r1,r2,r3 and rax. In the future I wanted to make it more challenging by adding a call instruction including a virtual stack with a calling convention hence the rax.

output is working fine for me:

Code `DWORD* dest = (DWORD*)((DWORD)vm_registers + (sizeof(DWORD) * reg));` I've decided to align storage registers in the same order and as the enumeration, this allows me to access the desired storage register directly (enum acting as an offset).

I'm not sure why it would not give you the right output, here the ready-to-paste snippet.

mini-vm.h:

Code `#pragma once`

`#include <Windows.h>`

`#include <stdio.h>`

`#include <stdint.h>`

`enum vm_reg : uint8_t {`

`eR0,`

`eR1,`

`eRAX,`

`eRDX,`

`};`

`enum vm_op : uint8_t {`

`eHALT,`

`eMOVER,`

`eMOVEC,`

`eADDR,`

`eADDC,`

`eSUBR,`

```
eSUBC,  
};
```

```
typedef struct reg_container_t {  
    uint32_t R0;  
    uint32_t R1;  
    uint32_t RAX;  
    uint32_t RDX;  
    uint32_t RIP;  
}reg_container;
```

```
extern uint32_t execute(const uint8_t* data);  
extern void initialize_vm();  
extern void dispose_vm();  
mini-vm.c:  
Code #include "mini-vm.h"
```

```
struct reg_container_t* vm_registers = NULL;
```

```
void handle(vm_op operation, uint8_t reg, uint8_t v);
```

```
uint32_t execute(const uint8_t* data)  
{  
    uint8_t* base = (uint8_t*)data;  
    vm_registers->RIP = 0;
```

```
    do  
    {  
        vm_registers->RIP++;
```

```
        uint8_t op = *data++;  
        uint8_t reg = *data++;  
        uint8_t v = *data++;
```

```
        handle((vm_op)op, reg, v);
```

```
        uint8_t next = *(base + (3 * vm_registers->RIP));
```

```
        if (next == eHALT)  
            break;
```

```
    } while (true);
```

```
    return vm_registers->RAX;  
}
```

```
void handle(vm_op operation, uint8_t reg, uint8_t v)  
{  
    uint32_t* dest = (uint32_t*)((uint32_t)vm_registers + (sizeof(uint32_t) * reg));
```

```
    switch (operation)  
    {  
        case eMOVEC:
```

```

*dest = v;
break;

case eMOVE:
*dest = *(uint32_t*)((uint32_t)vm_registers + (sizeof(uint32_t) * v));
break;

case eADDC:
*dest += v;
break;

case eADDR:
*dest += *(uint32_t*)((uint32_t)vm_registers + (sizeof(uint32_t) * v));
break;

case eSUBC:
*dest -= v;
break;

case eSUBR:
*dest -= *(uint32_t*)((uint32_t)vm_registers + (sizeof(uint32_t) * v));
break;

default:
printf("Unknown instruction: 0x%x\r\n", operation);
break;
}
}

void initialize_vm()
{
dispose_vm();
vm_registers = (reg_container_t*)malloc(sizeof(reg_container_t));
ZeroMemory(vm_registers, sizeof(reg_container_t));
}

void dispose_vm()
{
if (vm_registers != NULL)
{
free(vm_registers);
vm_registers = NULL;
}
}

```

yuumi - February 18, 2022, 1:16 pm

Yes I'm not sure either. But it's nice you decided to make the registers easier to address by aligning them. In my example I just went through a switch case of the register provided.

I also like how you just did this Code `uint8_t op = *data++;`
`uint8_t reg = *data++;`
`uint8_t v = *data++;`Where as I just checked indices, 0, 1, 2 then incremented rip at the end of the instruction.

If you enjoyed this one I think you'd enjoy the next portion of the challenge too.

0x9F - February 18, 2022, 1:20 pm

Your way of approaching this challenge was also pretty neat, I definitely enjoyed this challenge and I'm sure I will like your next one. Thanks for this ??

EmacsSucksButILikeIt - February 18, 2022, 8:16 pm

Here's a very fragile solution...

Code #include <stdio.h>

```
enum vm_op {
```

```
    kR0,
```

```
    kR1,
```

```
    kR2,
```

```
    kR3,
```

```
    kRax,
```

```
    kHalt,
```

```
    kMovReg,
```

```
    kMovConst,
```

```
    kAddReg,
```

```
    kAddConst,
```

```
    kSubReg,
```

```
    kSubConst,
```

```
};
```

```
unsigned char r0, r1, r2, r3, rax;
```

```
unsigned char* get_reg(unsigned char reg) {
```

```
    switch (reg) {
```

```
        case kRax:
```

```
            return &rax;
```

```
        case kR0:
```

```
            return &r0;
```

```
        case kR1:
```

```
            return &r1;
```

```
        case kR2:
```

```
            return &r2;
```

```
        case kR3:
```

```
            return &r3;
```

```
        default:
```

```
            return NULL;
```

```
    }
```

```
}
```

```
unsigned char process_byte_code(unsigned char * rip) {
```

```
    unsigned char frst, secnd;
```

```
    unsigned char * reg1 = NULL;
```

```
    unsigned char * reg2 = NULL;
```

```
    while (rip) {
```

```
        unsigned char instr = *rip++;
```

```
        switch (instr) {
```

```

case kMovConst:
    frst = *rip++;
    secnd = *rip++;
    *get_reg(frst) = secnd;
    break;
case kAddConst:
    frst = *rip++;
    secnd = *rip++;
    reg1 = get_reg(frst);
    *reg1 = *reg1 + secnd;
    break;
case kSubReg:
    frst = *rip++;
    secnd = *rip++;
    reg1 = get_reg(frst);
    reg2 = get_reg(secnd);
    *reg1 = *reg1 - *reg2;
    break;
case kAddReg:
    frst = *rip++;
    secnd = *rip++;
    reg1 = get_reg(frst);
    reg2 = get_reg(secnd);
    *reg1 = *reg1 + *reg2;
    break;
case kMovReg:
    frst = *rip++;
    secnd = *rip++;
    reg1 = get_reg(frst);
    reg2 = get_reg(secnd);
    *reg1 = *reg2;
    break;
case kSubConst:
    frst = *rip++;
    secnd = *rip++;
    reg1 = get_reg(frst);
    *reg1 = *reg1 - secnd;
    break;
case kHalt:
    return rax;
}
}
return rax;
}

int main(int argc, char **argv) {
    unsigned char bytecode[] = {
        kMovConst, kR0, 10,
        kAddConst, kR0, 5,
        kMovConst, kR1, 3,
        kSubReg, kR0, kR1,
        kMovConst, kR1, 4,
        kAddReg, kR0, kR1,

```

```

    kMovReg, kR1, kR0,
    kSubConst, kR1, 7,
    kMovReg, kRax, kR1,
    kHalt
};
process_byte_code(bytecode);
fprintf(stdout, "rax: %d, r0: %d, r1: %d, r2: %d, r3: %d\n", rax, r0, r1, r2, r3);
return 0;
}

```

Here's the output:

Quote:rax: 9, r0: 16, r1: 9, r2: 0, r3: 0

yuumi - February 19, 2022, 6:03 am

(Feb 18, 2022 09:16 PM)EmacsSucksButILikeIt Wrote: [\[CITE\]](#) Here's a very fragile solution...

Here's the output:

That's very similar to how I went about doing it. One advice I would give is rather than using global variables, use a structure to store your variables in. This could call be called a context structure and pass the structure pointer around between your vm related functions. It becomes easier to manage your code's scalability. Like if you were to run two vm's simultaneously or in loops, it would rely on the most recent variable values.

This is how mine were setup:

```

Code enum vm_op : uint8_t{
    kR0,
    kR1,
    kR2,
    kR3,
    kRax,
    kHalt,
    kMovReg,
    kMovConst,
    kAddReg,
    kAddConst,
    kSubReg,
    kSubConst,
};
struct vm_reg{
    int64_t r0;
    int64_t r1;
    int64_t r2;
    int64_t r3;
    int64_t rax;
    int64_t rip;
};
struct vm_context{
    struct vm_reg registers;
    uint8_t* instructions;
};

```

```
struct vm_context* initvm(void);  
void destroyvm(struct vm_context* ctx);  
void vm(struct vm_context* ctx);
```

I'm currently working on a slightly more complex version of this I haven't gone through the details yet.

unazed - June 4, 2022, 10:49 pm

so is it an assembler or a virtual machine? they're both quite different

yuumi - June 5, 2022, 6:30 pm

(Jun 4, 2022 10:49 PM)unazed Wrote: [\[CITE\]](#) so is it an assembler or a virtual machine? they're both quite different

The second version I had was actually an assembler to virtual machine byte code. But this specifically was writing the bytes out yourself.

In the case that you need encoding or bitshifting storing multiple bits of information in one variable such as a uint16 or uint32, then yes assembler would be really good.

For this challenge specifically since it was all done on single bytes and no bit shifting or encoding was involved, it didn't really need an assembler.