The Go Virtual Machines

Gina K. Nasseri

Abstract

The Go Virtual Machine provides an isolated computing environment for executing programs written for Susan; a simple I/O device which runs on a hypothetical ISA with ten 32-bit CPU registers.

1 Introduction

Process virtual machines provide an isolated computing environment within a host machine in which a process, known as the guest process, can be run independent from the underlying hardware of the host. The process is encapsulated by runtime software (runtime), which provides a layer of abstraction for the guest, handling all memory management and communications to the host operating system (OS). This allows a device which supports one OS to be able to run an application designed to be run on an OS or instruction set architecture (ISA) different than its own. This report provides a demonstrative example of this concept. It begins by defining a simple ISA for a hypothetical device which is given the name Susan. It takes the reader step by step in implementing a virtualized environment capable of running programs designed for Susan.

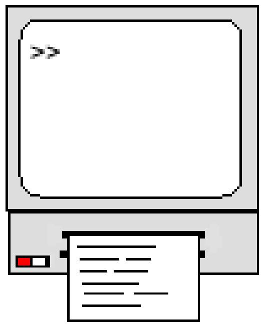


Figure 1: A device which runs on the Susan ISA

2 Susan Specs

I designed Susan to be a little device outfitted with a scanner and a display screen (Figure 1). Suse runs on the instruction set architecture (ISA) defined in the following sections. It executes programs by scanning a list of instructions from a tape and prints the output on its display screen. The scanner head keeps track of the current instruction number (PC) and the program stops when the scanner reaches the end of the tape.

2.1 Defining the ISA

Instruction Set Summary:

Mnemonic	**Description**	**Args**	
STDOUT	Print register value	REG	
LDI	Load Immediate	REG, INT	
JUMP	Jump	INT	
ADD	Add register values	REG, REG	*A
PRINTR	Print all registers		
ADDV	Visual mode add	REG, REG	
DRAW	Draw shape	SHAPE	
BLINK	Blink shape	SHAPE	

list with more detail is provided in Appendix A.

CPU Registers

The ISA includes ten 32 bit registers for read and write operations. Let Ri denote register i and Ri:Rj denote registers i through j:

- R0 - special purpose register for the program counter (PC). Only the scanner has write permissions for R0.
- R1:R9 - general purpose registers which can be used for read or write operations.

2.2 Exception Handling

An exception signal can raised by the scanner as a result of a `JUMP` k instruction if the k (address/instruction number) provided is not reachable. There are two cases which can cause an exception:

1. $k \leq$ PC: the scanner sends a signal that a request to move backward was received.
2. $k \gg$ PC: this refers to the case where k is greater than the number of instructions on the list. In this case, the scanner will reach the end of the paper and sends a signal that the end of the list

has been reached and a request to move forward was received.

In each case, an error is printed to the screen and the program exits with an error message. If k is reachable, the scanner stops after k steps, updates the PC in R0 to PC + k, and instruction execution is resumed with the current instruction under the scanner head.

3 Designing the GVM

The GVM executes Susan programs, referred to as the source program, within the Go runtime in two stages. First, the source code is parsed into representative bytecode instructions; instructions which consist of the opcode and references to any operands with correspond to the current instruction in the source code. Once the parsing is complete, the interpreter takes over.

The interpreter contains a list of routines, one for each opcode, where each routine is designed to emulate a single instruction on the source IS list. A routine emulating an instruction means that the routine produces the same change in state and/or output which would be observed if the instruction were executed on its native device or platform. When invoked, the interpreter looks at each bytecode instruction, decodes it, extracting the opcode and references to any operands, and then dispatches to the routine corresponding to the opcode.

4 First Stage: Parsing Input

4.1 Generating Tokens

The first step to processing input is to perform lexical analysis on the source code, which is to break the input down into tokens. The next step is feeding the tokens to the parser which first checks for any syntax errors and if none are present, then it generates a bytecode instruction from the source code instruction.

To generate tokens for the input, we need to first examine the language in order to define the set of token types to use. Then from those token types, we can separate the types into categories which define how each is processed.

The IS list was shown previously in 2.1, but a typical source program looks something like this:

```
LDI r1, 2
LDI r3, 4
ADD r1, r3
STDOUT r1
DRAW $bird
```

Examining the language, we can see that there are four non-command types: registers, integers, commas, and a $ followed by a shape, in addition to commands. So we need to define routines to process each type of token.

The token values are given type int32 to be consistent with the 32-bit CPU registers we will be emulating. The tokens are defined as:

```
type Token struct {
    TokenType string
    Value     int32
}
```

And the following constants are defined:

```
 const (
    INT     = "INT"
    REG     = "REG"
    COMMA   = "COMMA"
    LDI     = "LDI"
    STDOUT  = "STDOUT"
    JUMP    = "JUMP"
    ADD     = "ADD"
    ADDV    = "ADDV"
    DRAW    = "DRAW"
    BLINK   = "BLINK"
    SHAPE   = "SHAPE"
    PRINTR  = "PRINTR"
    EOF     = "EOF"
    )
```

The lexer is defined as:

```
type Lexer struct {
    Input string
    Position int
    CurrentChar byte
}
```

When a new lexer is created, it is initialized with a line of source code as the Input string, the Position is set to 0, and the CurrentChar is set at the first character in the string. The token generating function GetNextToken() is shown on the next page (with some lines omitted due to length). Suppose we are performing lexical analysis on the input `LDI r1, 2`: When a new lexer 'lex' is initialized, it has the following fields: `("LDI R1, 2", 0, 'L')`. When lex.GetNextToken() is called and the `switch` statement is entered on line 192, `lex.CurrentChar = 'L'` which satisfies the `.isUpper()` case on line 244, in which case the lexer attempts to get a valid command by calling: `command, err := lex.Command()` on line 244 (the code is provided in Appendix B), which builds a string starting from 'L' until a not upper case character is encountered, at which point `"LDI", nil` is returned to 244, with nil indicating that there were no errors. Since no error was returned, GetNextToken() moves into the switch statement on line 251. As case "LDI" is true, a new token `(token.LDI,0)` is created and returned along with a nil error. A value of 0 is assigned for all command and comma tokens which indicates a nil value.

```
func (lex *Lexer) GetNextToken() (*token.Token, error) {

    for lex.CurrentChar != 0 {
        switch {

        // Whitespace
        case unicode.IsSpace(rune(lex.CurrentChar)):
            lex.IgnoreWhiteSpace()

        // Register
        case unicode.ToLower(rune(lex.CurrentChar)) == 'r':
            regIndex, err := lex.RegisterIndex()
            if err != nil {
               return nil, err
            }
            return token.New(token.REG, regIndex), nil

        // Integer
        case unicode.IsDigit(rune(lex.CurrentChar)):
            integer, err := lex.Integer()
            if err != nil {
                return nil, err
            }
            return token.New(token.INT, integer),
        // Command
        case unicode.IsUpper(rune(lex.CurrentChar)):
            command, err := lex.Command()
            if err != nil {
                return nil, err
            }
            // ensure command is valid
            switch command {
            case "LDI":
                return token.New(token.LDI, 0), nil
            case "STDOUT":
                return token.New(token.STDOUT, 0), nil
            case "PRINTR":
                return token.New(token.PRINTR,0), nil
            case "JUMP":
                return token.New(token.JUMP, 0), nil
            case "ADD":
                return token.New(token.ADD, 0), nil
            case "ADDV":
                return token.New(token.ADDV, 0), nil
            case "DRAW":
                return token.New(token.DRAW, 0), nil
            case "BLINK":
                return token.New(token.BLINK, 0), nil
            default:
                return nil, fmt.Errorf("gvm: undefined: '%s'.",command)
            }
        // something else
        default:
            return nil, fmt.Errorf("gvm: unrecognized symbol '%c'",rune(lex.CurrentChar))
        }
```

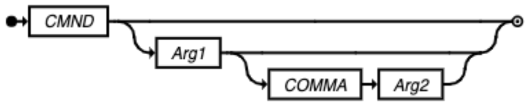
INT and REG are assigned the integer value and register index (respectively), and SHAPE values are 1 if the shape is a heart and 2 if the shape is a bird. (Shape code omitted). The method for generating an integer token is also provided in Appendix B, along with some additional notes about finding lexing errors.

4.2 Generating Bytecode

To create the "bytecode" instructions for my interpreter, I defined an opcode for each instruction in the Susan IS:

```
const (
    OPCODE_STDOUT = 0x00
    OPCODE_LDI    = 0x01
    OPCODE_JUMP   = 0x02
    OPCODE_ADD    = 0x17
    OPCODE_ADDV   = 0x18
    OPCODE_DRAW   = 0x19
    OPCODE_BLINK  = 0x20
    OPCODE_PRINTR = 0x21
)
```

By observing the general syntax structure of the language, we can make a syntax diagram which all instructions must satisfy.



This defines the structure of the language and defines how tokens are parsed. There are three types of instructions; instructions either have 0, 1 or 2 arguments. We can now define an Instruction interface and define three instruction types according to the number of arguments:

```
type Instruction interface {
    GetOpCode() int32
    GetArg1()   int32
    GetArg2()   int32
}

// Binary instructions: 2 arguments
type BinaryInstruction struct {
    OpCode, Arg1, Arg2 int32
}

// Unary Instructions:  single argument
type UnaryInstruction struct {
    OpCode int32
    Arg1   int32
}

// Nullary Instructions: no arguments
type NullaryInstruction struct {
    OpCode int32
}

// Initialization, Arg1, Arg2 methods
// temporarily omitted for space
```

Each instruction type has an opcode field in addition to 0, 1, or 2 arguments. Now that we have our instructions defined and a syntax diagram to follow, we define a parser which has a lexer. And we can initialize a new parser instance as follows:

```
type Parser struct {
     Lex *lexer.Lexer
     CurrentToken *token.Token
}
func New(input string) (*Parser, error) {
    lex := lexer.New(input)
    currToken, err := lex.GetNextToken()
    return &Parser{Lex: lex, CurrentToken: currToken}, er
}
```

A parser is initialized with a new lexer instance, which it passes the input string, which is a line in the source code, and it sets the current token to the first token from the input returned from lex.GetNextToken(). The next page shows the parser method to advance to the next token, Consume(), and a portion of the Instruction generating method Instruction().

The instruction method creates bytecode instructions from a stream of tokens as they are parsed. If the current token type is a specific instruction, then Instruction checks that the instruction syntax is correct and all required parameters have been provided. If the instruction syntax is correct, then a bytecode instruction is returned to the interpreter, in other words, it reads the current type of instruction being parsed and attempts to create a bytecode instruction. The expected syntax for each instruction is provided as a comment following the `case INSTR` line.

As an example, suppose a new parser instance is initialized with the input `"LDI r1, 6"` (load the value 6 to R1), the parser will be initialized with a new lexer instance, then the call to lex.GetNextToken() sets the CurrentToken field to (token.LDI,0). Now suppose the Instruction() method (next page) is invoked on the parser instance: the case for currentToken.TokenType = LDI (line 98) is true. Lines 99 to 119 demonstrate how the parser obtains the LDI arguments. Once the register index and load value have been found. The parser can now create a BinaryInstruction from the LDI opcode along with the register index and integer value. The string representation for the binary instruction is `0x02/1/6`.

The instruction gives the opcode for the LDI instruction: 0x02 along with a reference to the register to do the write to: R1, and the value to load: 6. For the decoding step in instruction emulation, we implement the interface methods in the instruction interface.

```
func (p *Parser) Consume(expectedType string) error {
    if p.CurrentToken.TokenType != expectedType {
        // return unexpected token error message
    }
    var err error
    p.CurrentToken, err = p.Lex.GetNextToken()
// return error if error other wise return nil

----------------------------------------------------------
func (p *Parser) Instruction() (instructions.Instruction, error) {
    currentToken := p.CurrentToken
    switch currentToken.TokenType {

    case token.JUMP:
        // JUMP INT (address)
        if err := p.Consume(token.JUMP); err != nil {
            return instructions.NewError(err), err
        }
        currentToken = p.CurrentToken
        if err := p.Consume(token.INT); err != nil {
            return instructions.NewError(err), err
        }
        jumpTo := currentToken.Value
        return instructions.NewUnaryInstruction(int32(OPCODE_JUMP), jumpTo), nil

    case token.LDI:
        // LDI REG COMMA INT
        if err := p.Consume(token.LDI); err != nil {
            return instructions.NewError(err), err
        }
        // REG (load to)
        currentToken = p.CurrentToken
        if err := p.Consume(token.REG); err != nil {
            return instructions.NewError(err),err
        }
        regIndex := currentToken.Value

        // COMMA
        if err := p.Consume(token.COMMA); err != nil {
            return instructions.NewError(err), err
        }
        // INT (load value)
        currentToken = p.CurrentToken
        if err := p.Consume(token.INT); err != nil {
            return instructions.NewError(err), err
        }
        loadValue := currentToken.Value

        // return LDI instruction bytecode
        return instructions.NewBinaryInstruction(int32(OPCODE_LDI),regIndex,loadValue),     nil

    case token.ADD:
        // ADD REG COMMA REG
        if err := p.Consume(token.ADD); err != nil {
            return instructions.NewError(err), err
```

For example, for the BinaryInstructions, we include the following:

```
func (bi *BinaryInstruction) GetOpCode() int32 {
    return int32(bi.OpCode)
}

func (bi *BinaryInstruction) GetArg1() int32 {
    return int32(bi.Arg1)
}

func (bi *BinaryInstruction) GetArg2() int32 {
    return int32(bi.Arg2)
}
}
```

And we write the same functions for Unary and Nullary instructions, only with GetArg2() returning nothing and GetArg1() and GetArg2() returning nothing for the Unary and Nullary types, respectively.

An Aside: Go Slices

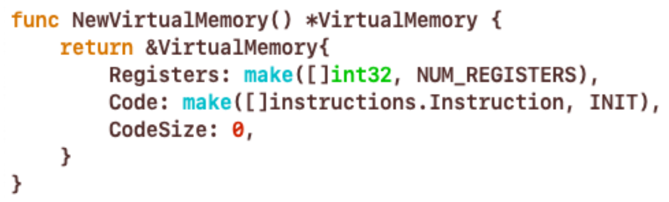
A Go slice is an array with an abstraction layer which provides a "dynamically-sized, flexible view into the elements of an array"[1]. When a slice is initialized by `make([]type, SIZE)`, memory is allocated for the underlying array, and the slice is initialized with a pointer to the array, the length of the slice, and the initial capacity. If the length of the slice reaches capacity, the resizing is automatically handled by the abstraction layer. Similarly, unused memory is automatically identified and freed. Go slice types, which are also reference types, then provide an abstracted and protected emory space which can be shared among the structures in the virtual machine.

4.3 Defining the Guest Memory Image

Now that we can parse the source code instructions into our opcode instruction types, we need to designate a dynamic and protected memory space to write them too. We also need to emulate the 32 bit CPU registers. So we can define a VirtualMemory struct as follows:

```
         type VirtualMemory struct {
        Registers []int32
        Code []instructions.Instruction
         CodeSize int32
     }
```

Now, knowing what we know about how Go slices work, we can allocate a region of memory for the guest memory image when we initialize a new instance of VirtualMemory:



```
func NewVirtualMemory() *VirtualMemory {
    return &VirtualMemory{
        Registers: make([]int32, NUM_REGISTERS),
        Code: make([]instructions.Instruction, INIT),
        CodeSize: 0,
    }
}
```

When a new VirtualMemory instance is created, memory is allocated for the underlying register and code block arrays and the CodeSize is set to 0 indicating that no instructions have been parsed yet.

4.4 Defining the VM

We define our virtual machine with a pointer to the VitualMemory struct and an Interpreter

```
type VirtualMachine struct {
    VMem *VirtualMemory
    Interpreter *interpreter.Interpreter
}
```

When we initialize a new VirtualMachine instance, we initialize a new VirtualMemory instance as described above, and we pass it by reference to the interpreter. The interpreter now has access to the guest memory image, which includes the emulated CPU registers and the code block.

```
func NewVirtualMachine() *VirtualMachine {
    vMem := NewVirtualMemory()
    return &VirtualMachine{
      VMem: vMem,
      Interpreter: interpreter.New(vMem.Registers,
           vMem.Code),
    }
}
```

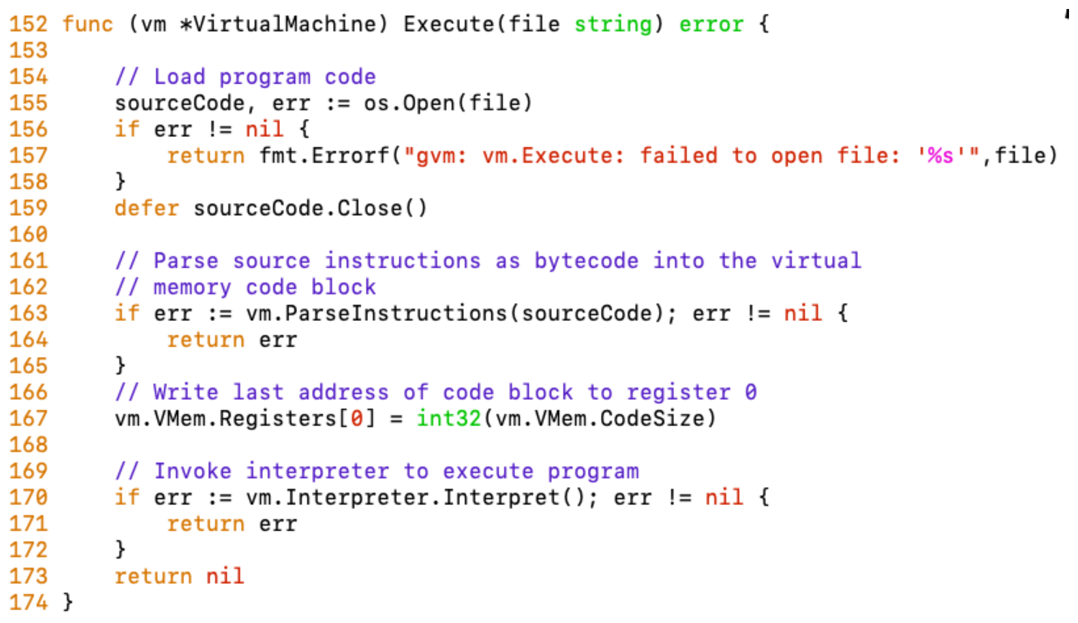
4.5 Parse the Code to VirtualMemory

So now, we have our virtual memory set up and we can parse a file of source code into the guest memory image. The following code in figures 2 and 3 on the next page are added into the VM's package

Suppose a new VM instance has been initialized and 'file' is a file which contains the instuction "LDI r1, 6". After the lines 155-163 in 2 and then in 3, line 118,

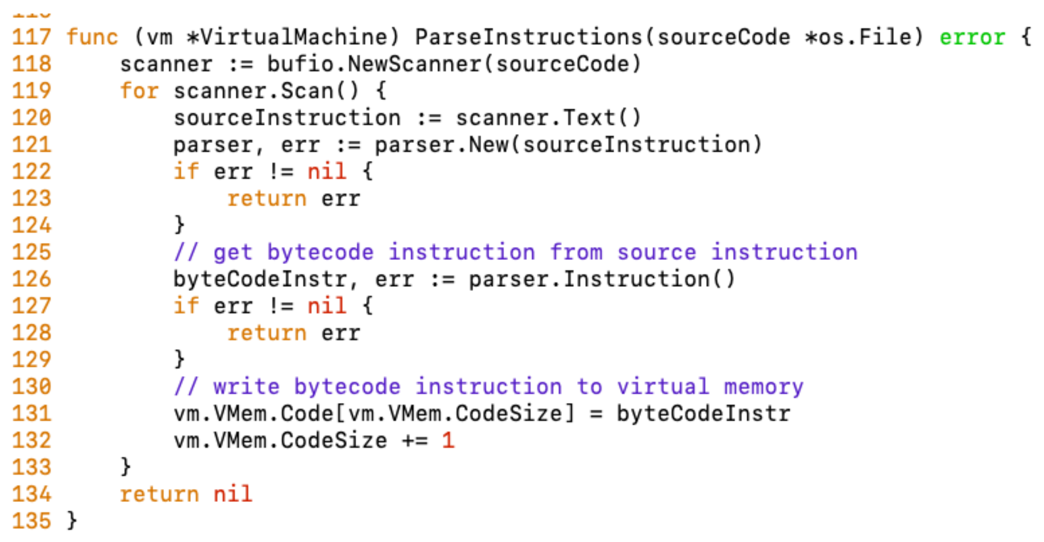
1. (line 118) a new parser is generated with "LDI r1, 6" as input
2. (line 126) byteCodeInstr is assigned `0x02/1/6`
3. (line 131) the parsed bytecode instruction is written into VMem

Once all the instructions are written, the size of the code block is written to R0, (which the interpreter has access to, since its initialized with this memory) the interpreter uses this to emulate the exception where a JUMP instruction would push the scanner off the edge of the tape.



```
152 func (vm *VirtualMachine) Execute(file string) error {
153
154     // Load program code
155     sourceCode, err := os.Open(file)
156     if err != nil {
157         return fmt.Errorf("gvm: vm.Execute: failed to open file: '%s'",file)
158     }
159     defer sourceCode.Close()
160
161     // Parse source instructions as bytecode into the virtual
162     // memory code block
163     if err := vm.ParseInstructions(sourceCode); err != nil {
164         return err
165     }
166     // Write last address of code block to register 0
167     vm.VMem.Registers[0] = int32(vm.VMem.CodeSize)
168
169     // Invoke interpreter to execute program
170     if err := vm.Interpreter.Interpret(); err != nil {
171         return err
172     }
173     return nil
174 }
```

Figure 2: The `vm.Exeute()` function in the vm/vm.go package.



```
117 func (vm *VirtualMachine) ParseInstructions(sourceCode *os.File) error {
118     scanner := bufio.NewScanner(sourceCode)
119     for scanner.Scan() {
120         sourceInstruction := scanner.Text()
121         parser, err := parser.New(sourceInstruction)
122         if err != nil {
123             return err
124         }
125         // get bytecode instruction from source instruction
126         byteCodeInstr, err := parser.Instruction()
127         if err != nil {
128             return err
129         }
130         // write bytecode instruction to virtual memory
131         vm.VMem.Code[vm.VMem.CodeSize] = byteCodeInstr
132         vm.VMem.CodeSize += 1
133     }
134     return nil
135 }
```

Figure 3: The `vm.ParseInstructions()` function in the vm/vm.go package.

5 Final Stage: Instruction Emulation

```
type Interpreter struct {
    Registers []int32
    Code []instructions.Instruction
      PC int32
}
```

Recall that when we initialized a new virtual machine, it initializes a new interpreter instance and passes it a reference to the virtual memory, so initializing a new virtual machine instance initializes an interpreter with a reference to an underlying int32 array and instruction array.

Referring to the previous code show in figure 2, on line 70, when the interpreter is invoked; it now has access to the guest memory image and the emulated CPU registers. The interpreter.Interpret() function is shown on the next page as well as some of the DecodeAndDispatch() loop and some of the function list.

5.1 The Interpreter's Decode and Dispatch Centre

```
-------------------------------------------------------------
// Interpret reads each bytecode instruction in the virtual memory codeblock
// and calls the decode and dispatch routine for each instruction. After
// the source code is parsed as bytecode into the virtual memory codeblock,
// the VM writes the size of the codeblock into register 0 so the interpreter
// knows the start and end addresses of the address space where the bytecode
// is stored, i.e., where the interpreter has permission to access.
func (interp *Interpreter) Interpret() error {
    lastAddr,_ := interp.ReadFrom(0)
    for interp.PC < lastAddr {
        if err := interp.DecodeAndDispatch(interp.Code[interp.PC]); err != nil {
            return err
        }
        interp.PC++
    }
    return nil
}
--------------------------------------------------
// DecodeAndDispatch reads a bytecode instruction to obtain the OpCode for the
// current instruction and, if a valid opcode is obtained, the interpreter
// gets any additional information needed from the bytecode, depending on the
// type of instruction, and then dispatches to the appropriate routine to
// execute the instruction.
func (interp *Interpreter) DecodeAndDispatch(instr instructions.Instruction) error {
    switch instr.GetOpCode() {

     // LDI
    case OPCODE_LDI:
        if err := interp.LoadImmediate(instr.GetArg1(),instr.GetArg2()); err != nil {
            return err
        }
        return nil

    // JUMP
    case OPCODE_JUMP:
        if err := interp.JumpTo(instr.GetArg1()); err != nil {
            return err
        }
        return nil

    // ADD
    case OPCODE_ADD:
        if err := interp.Add(instr.GetArg1(),instr.GetArg2()); err != nil {
            return err
        }
        return nil

    // ADDV
    case OPCODE_ADDV:
        if err := interp.AddV(instr.GetArg1(),instr.GetArg2()); err != nil {
            return err
        }
        return nil

```

5.2 Interpreter: Exception Emulation

The checkJump function is used to emulate the exception cases that can arise from a jump instruction. After all instructions are parsed into the virtual memory code block, the virtual machine vm.Execute() control point writes the number of instructions into R0. When a JUMP k instruction is found, the interpreter dispatches to the JumpTo(k) routine which calls CheckJump to test for the exception cases.

Note on Register Mapping

Recall in the Susan ISA, R0 holds the PC. During the parsing stage, this is not being incremented because the source code is not directly being executed, only written as input for the interpreter. From the point of view of the rest of the world, it's just regular data. Because the interpreter is the part of the VM that does the actual execution, R0 is mapped to interpreter.PC., i.e., interpreter.PC emulates R0. This is used in the CheckJump to see if a Jump instruction would ask the scanner to move backward.

In our interpreter, R0, lets call it VR[0] holds the total number of instructions in the code block. This is used to check if a Jump instruction would cause the scanner to run off the tape.

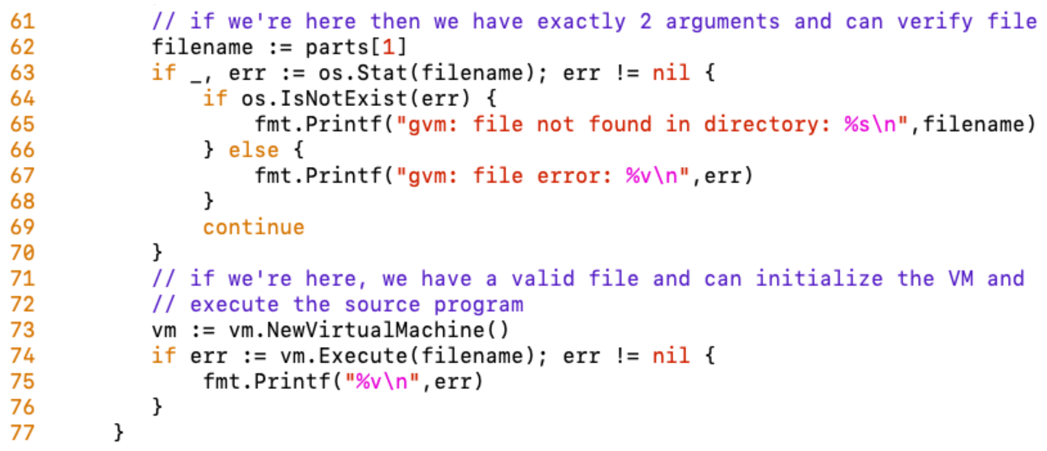
If either case is true, the interpreter returns an error as Susan would, and the program exits.

```
-------------------------------------------------------
// CheckJump validates the requested jump address provided as an argument to a
// JUMP instruction.
//
// If the address provided is less than the current address, then an infinite loop
// warning error is returned.
//
// If the address provided is outside of the virtual memory codeblock, where the last
// address of the codeblock is provided in register 0, then a segmentation violation
// error is returned as the JUMP address is not a valid memory address.
func (interp *Interpreter) CheckJump(addr int32) error {
    lastAddr,_ := interp.ReadFrom(0)
    if addr <= interp.PC {
        return fmt.Errorf("gvm: JUMP at addr %d to %d: infinite loop warning.",interp.PC,addr)
    }
    if addr > lastAddr - 1 {
        return fmt.Errorf("gvm: JUMP addr invalid: segmentation violation.")
    }
    return nil
--------------------------------------------------
// JUMP routine: JumpTo
// JumpTo validates the jump address with the CheckJump function and,
// if no error is returned, then the PC is updated to the jump to
// address - 1 (as the PC is incremented when returned). The address
// is converted to an integer
func (interp *Interpreter) JumpTo(address int32) error {
    if err := interp.CheckJump(address); err != nil {
        return err
    }
    interp.PC = address - 1
    return nil
}
```

6 Run the Machine!

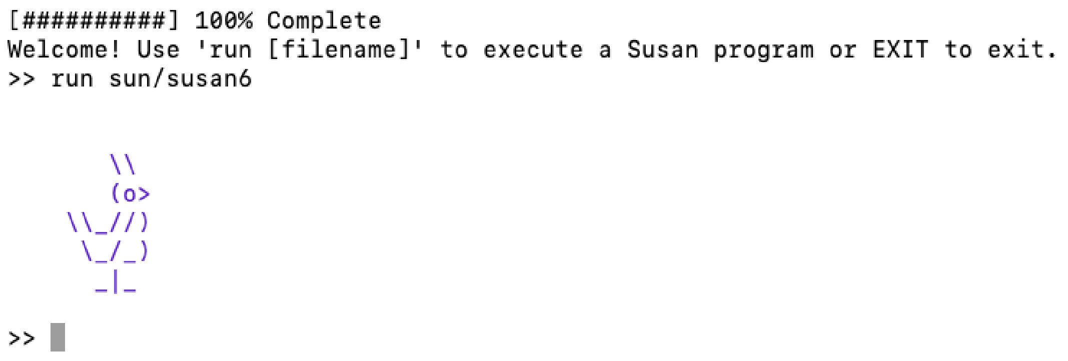
The virtual machine is started by executing `go run main.go`. When the machine starts up, a welcome message is printed and the user is prompted for a file.

To execute a program, the user can create a text file containing a list of instructions within the Susan IS (Appendix A), and enter 'run filepath/filename' to run the program on the virtual machine. The file path must be verified before any virtualisation steps begin. Once the user has provided a valid file, a new VirtualMachine instance is initialized with VirtualMemory and an Interpreter, as outlined in the previous section. Once the initialization is complete, control is transferred to vm.Execute() (in vm/vm.go), passing the the filename as the parameter. The implementation of these steps is shown in Figure 4. Figure 5 is a screenshot of the execution environment after running the program susan6 on the VM. (The main execution loop is in appendix C).



```go
        // if we're here then we have exactly 2 arguments and can verify file
        filename := parts[1]
        if _, err := os.Stat(filename); err != nil {
            if os.IsNotExist(err) {
                fmt.Printf("gvm: file not found in directory: %s\n",filename)
            } else {
                fmt.Printf("gvm: file error: %v\n",err)
            }
            continue
        }
        // if we're here, we have a valid file and can initialize the VM and
        // execute the source program
        vm := vm.NewVirtualMachine()
        if err := vm.Execute(filename); err != nil {
            fmt.Printf("%v\n",err)
        }
    }
```

Figure 4: Main execution loop: verifying file and initializing VM.

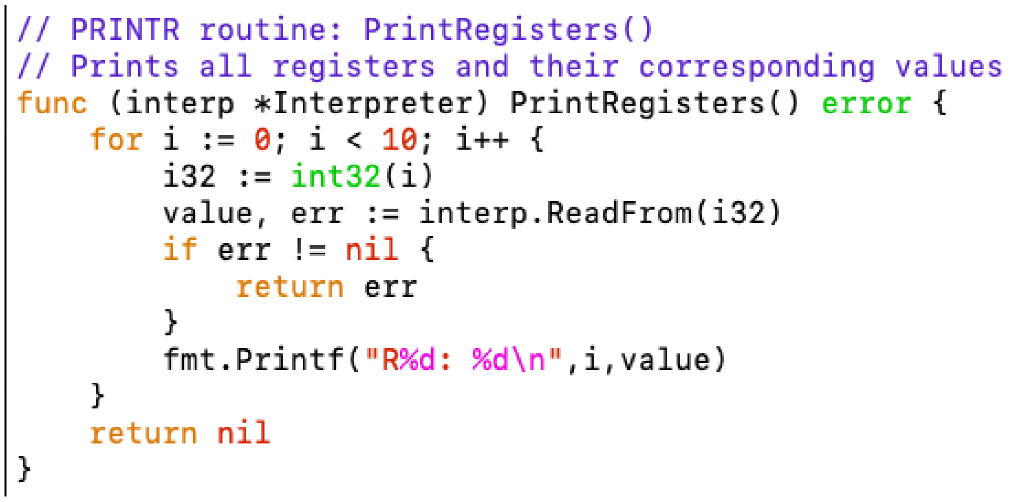


```
[##########] 100% Complete
Welcome! Use 'run [filename]' to execute a Susan program or EXIT to exit.
>> run sun/susan6


       \\
       (o>
    \\_//)
     \_/_)
      _|_

>> 
```

Figure 5: Start the program and run sun/susan6

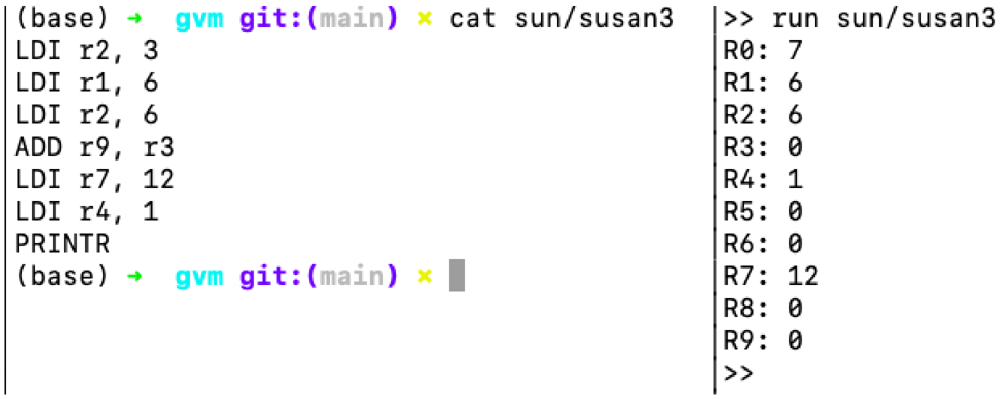
6.1 Register Emulation

The following figures demonstrate the emulation of the CPU registers. Figure 9 shows the interpreter's `PRINTR` routine. Figure 7 contains two figures: the left figure shows the contents of a Susan program which modify the CPU registers and then prints the register contents, and the right figure shows the result of running the program in the virtual machine. Notice that R0 contains the number of instructions, which is equivalent to what R0 would contain if we ran the program on the Susan device. Recall that R0 contained the PC on the Susan device and the virtual machine wrote the final instruction count into R0 after all instructions in the program were parsed into the guest memory image. As we can see, the register state in the virtual machine is equivalent to what it would be if we ran the program on the Susan device.



```
// PRINTR routine: PrintRegisters()
// Prints all registers and their corresponding values
func (interp *Interpreter) PrintRegisters() error {
    for i := 0; i < 10; i++ {
        i32 := int32(i)
        value, err := interp.ReadFrom(i32)
        if err != nil {
            return err
        }
        fmt.Printf("R%d: %d\n",i,value)
    }
    return nil
}
```

Figure 6: The interpreter routine for the `PRINTR` routine.



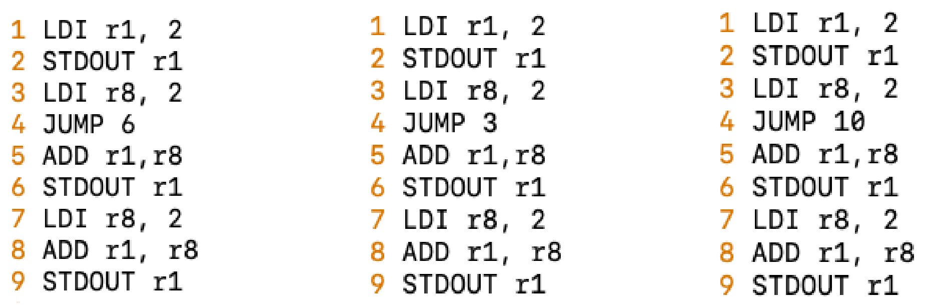
```
(base) → gvm git:(main) ✗ cat sun/susan3
LDI r2, 3
LDI r1, 6
LDI r2, 6
ADD r9, r3
LDI r7, 12
LDI r4, 1
PRINTR
(base) → gvm git:(main) ✗
```

```
>> run sun/susan3
R0: 7
R1: 6
R2: 6
R3: 0
R4: 1
R5: 0
R6: 0
R7: 12
R8: 0
R9: 0
>>
```

Figure 7: Left: program contents. Right: the results of running the program in the virtual machine.

6.2 Exception Emulation

The following figures provide an example of emulating the `JUMP` instruction. Figure 8 displays the contents of 3 programs: sun/susan2 which includes a JUMP skipping the ADD r1, r8 instruction. Running sun/susan2, the program in Figure 8a shows the register contents unmodified on the second STDOUT r1 as a result of skipping the ADD. Figure 8b and 8c show programs sun/infinite and sun/off page which contain programs which would result in the scanner head receiving a request to move backwards and running off of the tape, respectively. Figure 9 shows the result of running the programs in the virtual machine (although, the error messages along the lines of ”scanner head: move backward request” and ”scanner head: move forward off tape request” would have been more appropriate capturing the essence of simulating the scanner exceptions).



```
1 LDI r1, 2
2 STDOUT r1
3 LDI r8, 2
4 JUMP 6
5 ADD r1,r8
6 STDOUT r1
7 LDI r8, 2
8 ADD r1, r8
9 STDOUT r1
```

(a) Legal jump from PC = 4 to PC = 6.

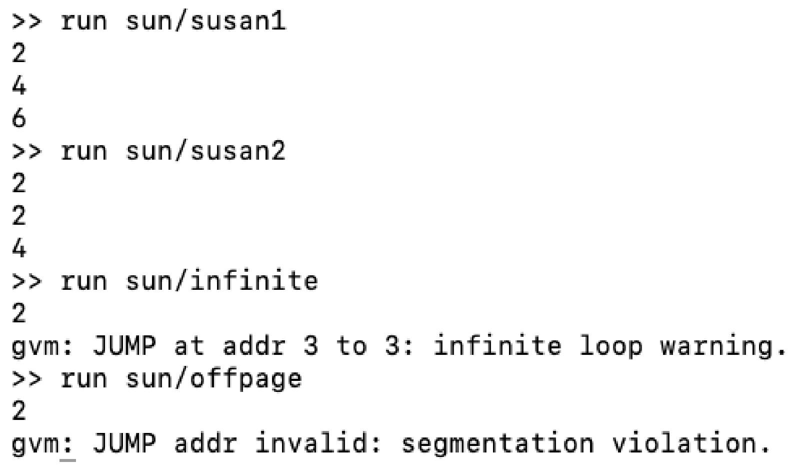
```
1 LDI r1, 2
2 STDOUT r1
3 LDI r8, 2
4 JUMP 3
5 ADD r1,r8
6 STDOUT r1
7 LDI r8, 2
8 ADD r1, r8
9 STDOUT r1
```

(b) Illegal jump from PC = 4 to PC = 3

```
1 LDI r1, 2
2 STDOUT r1
3 LDI r8, 2
4 JUMP 10
5 ADD r1,r8
6 STDOUT r1
7 LDI r8, 2
8 ADD r1, r8
9 STDOUT r1
```

(c) Illegal jump from PC = 4 off the "tape".

Figure 8: Emulating `JUMP` instruction.



```
>> run sun/susan1
2
4
6
>> run sun/susan2
2
2
4
>> run sun/infinite
2
gvm: JUMP at addr 3 to 3: infinite loop warning.
>> run sun/offpage
2
gvm: JUMP addr invalid: segmentation violation.
```

Figure 9: The results of running each program in the virtual machine.

Very Brief Discussion

This was a fun project. The textbook warnings made me a bit wary going into this. The Go runtime tools took care of all the things they warned me about. The runtime has some interesting features for threading designed for Goroutines as well as listening channels for doing interrupts which is something I'm interested in doing as I only had to manually handle 1 exception case. It would be fun to see what challenges arise/don't arise from adding in some interrupts using the Go runtime helper functions.

Appendix

A Appendix: Susan Instruction Set Summary

Mnemonic	Operands	Description	Operation
STDOUT	Rd	Print register value	
LDI	Rd, K	Load Immediate	$Rd \leftarrow K$
JUMP	k	Jump	PC $\leftarrow K$
ADD	Rd, Rr	Add register values	$Rd \leftarrow Rd + Rr$
PRINTR		Print all registers	
ADDV	Rd, Rr	Visual mode add	$Rd \leftarrow Rd + Rr$
DRAW	$s	Draw shape	
BLINK	$s	Blink shape	

- Rd: Destination (and source) register in register file
- Rr: Source register in the register file
- K: Constant data
- k: constant address

B Appendix: Lexer Code

B.1 Get Command String

```
// Command builds command string in user input. Commands are uppercase strings
// with a maximum length of 5.
// If the command string exceeds the maximum length, then an error is returned
// with an empty string. Otherwise, the command string is returned for further
// verification.
func (lex *Lexer) Command() (string, error) {
var builder strings.Builder
for lex.CurrentChar != 0 && unicode.IsUpper(rune(lex.CurrentChar)) {
if builder.Len() > 10 {
return "", fmt.Errorf("gvm: invalid command: max length reached (10).")
}
builder.WriteRune(rune(lex.CurrentChar))
lex.GetNextChar()
}
return builder.String(), nil
}
```

B.2 Get Integer String

```
// Integer parses a multi-digit integer string and returns it as an int32 value.
// int32 values are used to be consistent with the ISA of the native OS which
// the source code was written for as register values in the ISA hold int32 values.
func (lex *Lexer) Integer() (int32, error) {
if !lex.Register() {
if !lex.Delimiter() {
return 0, fmt.Errorf("gvm: syntax error: unexpected INT (missing delimiter)")
}
}
integerString := ""
for lex.CurrentChar != 0 && unicode.IsDigit(rune(lex.CurrentChar)) {
integerString += string(lex.CurrentChar)
lex.GetNextChar()
}
integer, err := strconv.ParseInt(integerString, 10, 32) // returns an int64 value
if err != nil {
return 0, fmt.Errorf("gvm: lexer: failed to convert input to integer %w",err)
}
// check for integer overflow
if integer > math.MaxInt32 {
return 0, fmt.Errorf("gvm: register integer overflow error")
}
// check for integer underflow
if integer < math.MinInt32 {
return 0, fmt.Errorf("gvm: register integer underflow error")
}
integer32 := int32(integer) // convert to int32
return integer32, nil
}
```

Notes on errors: there were a lot to check for and I checked for them all.

C Appendix: Main Execution Loop

```
    for {
fmt.Print(">> ")
if !scanner.Scan() {
fmt.Println("gvm: error reading from STDIN channel: exiting program.\n")
break // EOF or error
}
input := scanner.Text()
// user hits enter
if input == "" {
continue
}
// splitting input
parts := strings.Fields(input)
// user exits (case insensitive)
if strings.EqualFold(parts[0], "EXIT") {
break
}
if parts[0] != "run" {
fmt.Printf("gvm: invalid input: use 'run [filename]' to execute program or EXIT to exit.\n")
continue
}
// if we're here, then parts[0] is run
if len(parts) < 2 {
fmt.Printf("gvm: missing filename\n")
continue
}
if len(parts) > 2 {
fmt.Printf("gvm: too many arguments\n")
continue
}
// if we're here then we have exactly 2 arguments and can verify file
filename := parts[1]
if _, err := os.Stat(filename); err != nil {
if os.IsNotExist(err) {
fmt.Printf("gvm: file not found in directory: %s\n",filename)
} else {
fmt.Printf("gvm: file error: %v\n",err)
}
continue
}
// if we're here, we have a valid file and can initialize the VM and
// execute the source program
vm := vm.NewVirtualMachine()
if err := vm.Execute(filename); err != nil {
fmt.Printf("%v\n",err)
}
}
```

D Appendix: Interpreter Code

D.1 ADDV routine

```go
func (interp *Interpreter) AddV(ri, rj int32) error {
    value1, err := interp.ReadFrom(ri)
    if err != nil {
        return err
    }
    value2, err := interp.ReadFrom(rj)
    if err != nil {
        return err
    }
    if value1 > 10 || value2 > 10 {
        return fmt.Errorf("gvm: ADDV instruction: please use values less than 10 for visual add mode")
    }
    i := int(value1)
    j := int(value2)
    k := int32(i + j)
    interp.WriteTo(ri, k)

    fmt.Printf("%d + %d ",i,j)

    // colour string functions for the first i stars
    starColor1 := color.New(color.FgRed).SprintFunc()
    message1 := strings.Repeat("* ",i)
    for _, char := range message1 {
        fmt.Print(starColor1(string(char))) // applies function to string
        time.Sleep(100 * time.Millisecond)
        os.Stdout.Sync()
    }

    time.Sleep(100 * time.Millisecond)
    fmt.Printf("+ ")
    time.Sleep(100 * time.Millisecond)

    // the j stars
    message2 := strings.Repeat("* ",j)
    starColor2 := color.New(color.FgBlue).SprintFunc()
    for _, char := range message2 {
        fmt.Print(starColor2(string(char)))
        time.Sleep(100 * time.Millisecond)
        os.Stdout.Sync()
    }

    time.Sleep(100 * time.Millisecond)
    fmt.Printf("= ")

    // the i + j stars
    message3 := strings.Repeat("* ",i+j)
    starColor3 := color.New(color.FgGreen).SprintFunc()
    for _, char := range message3 {
        fmt.Print(starColor3(string(char)))
        time.Sleep(100 * time.Millisecond)
        os.Stdout.Sync()
    }
    fmt.Println("")
    return nil
}
--------------
\\DRAW HEART ROUTINE
--------------
// DrawHeart prints a heart shape to the screen. If blink is
// set to 1 then the heart will blink on the screen.
func (interp *Interpreter) DrawHeart(blink int) {
    heart := ‘
    ."". ."".
    |   ’   |
     \     /
      ’. .’
        ’
‘
    if blink == 0 {
        color.Red(heart)
    } else {
        blinkHeart := color.New(color.FgRed, color.BlinkSlow).SprintFunc()
        print(blinkHeart(heart))
    }
    return
}
```

References

[1] https://go.dev/tour/moretypes/7