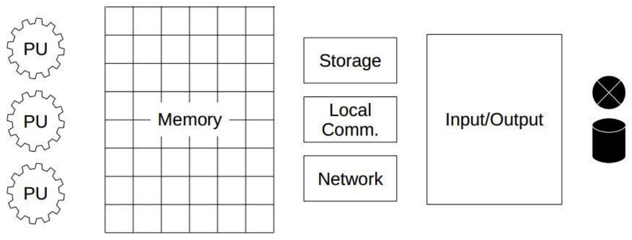
BOOTSTRAP LOADER

Author, Alan Israel M

Abstract— **To study and analyse a virtual machine and make the virtual machine able to load instructions from input and store them in memory for execution. To be able to compute a binary function with its two arguments by taking in instructions through the machine and executing.**

I. Introduction

A virtual machine is a conceptual machine built on a physical computer. A computing machine consists of processing units, memory and input/output (storage, local communications, remote communications or network and device input/output).



II. Problem statement

To build a new VM that reads a sequence of codes from standard input, load them into a given starting address and jump to that location to execute them. Input instructions should compute any one binary operation (add, sub, mul, div, max,min, ...) on two values read from standard input and output the result and also save it inmemory. The code should save the state into an Image file and resume from this Image to prove that the resume path also computes correctly.

Input sequence syntax is : <start-addr> <count> <instructions .... > <arg1> <arg2>.

III. Research

A series of virtual machines were analyzed to study the concept of a stored program computing machine. The machine is called virtual because it is a purely conceptual creation (program) on one's laptop (host). It depends on the host's facilities for actual interactions.

The first machine – **AVM** – is a 8-bit computer with one processing unit and 64 bytes of memory. It has seven instructions – **HLT**, **MOV**, **ADD**, **SUB**, **JNZ**, **JMR**, **MPC**. On startup, it executes instructions from index 0 in memory.

BVM introduces two new instructions – **IN** for reading numbers from standard input and **OUT** for writing numbers to standard output. Its processing core uses a single flag register, so its JNZ instruction can test for any flag instead of just zero. Its memory is now increased to its maximum of 128 bytes.

CVM modifies **HLT** instruction into **SUS** (suspend). Before halting, it writes the contents of memory into a file. On running the program again with this file, its memory contents (not registers) are restored before starting. It no longer needs to read a, b from standard input.

DVM introduces two new instructions **AT** (at:put:) and **ATP** (put:) to allow memory indexes to be variables, so we can read locations mem[r] or store into mem[r]. AT takes index register to read from mem and then stores it in another location, while ATP takes an expression and an index register to store into memory.

Then a new machine was built which does not introduce any new instructions, but modifies only the initial program stored in memory. This program now reads in a sequence of codes, stores them in memory and then jumps to its starting address to run it. It has become an independent virtual machine that can modify its own behavior. Its initial program that loads another program is known as **bootstrap loader** or simply **boot** and the program that it loads is known as the **kernel**.

IV. Resources and methods

PU – The processing unit is the core of the machine which executes the instructions in the memory, it has the following:

- *program counter*
- *flag registers*
- *8 registers*
- *Pointer to memory*
- *Memory size*
- *Opcode functions*

Memory – The VM has **128** bytes of storage.

Encoding Scheme : Addresses, registers and numbers are packed into a byte with **b0** (lsb) indicating the type. Type 0 has the value encoded directly in bits **b7..b1** while Type 1 has the address bits in those bits.

For example to access memory location [63] , the address is encoded as 63*2 = 126 and stored in memory. For storing value, it is encoded as [n*2 + 1].

There are eleven opcodes offered by the Virtual Machine. Those are :

1) **SUS** – Suspend execution of instructions
2) **MOV** – MOV rval lval moves value in location at rval to location at lval.
3) **ADD** – ADD expr addr does an add operation and stores value in addr.
4) **SUB** – SUB expr addr does a subtract operation and stores value in addr.
5) **JIF** - JIF flag, addr - set PC to addr if flag is false
6) **JMR** - JMR expr - jump to given address
7) **MPC** - MPC reg - save pc contents in a register
8) **IN** - IN ch, addr puts ch in address addr
9) **OUT** – OUT ch, addr outputs value present in addr
10) **AT** - AT @ix, addr does an indirect addressing by putting value in ix to addr
11) **ATP** - ATP expr, @ix also does indirect addressing by putting expr in the value ix

Using the above resources the VM was built.

V. CODE FLOW AND ANALYSIS

The code consists of two parts :

1) **BOOT**
2) **KERNEL**

The BOOT code is the hardcoded instructions the VM must execute on bootup. This BOOT then has to read a sequence of codes, load them into memory and start executing them in order to compute a binary function such as add / multiply. This is the KERNEL part of the code. The pseudo code is as follows:

```
uint8 KSTART, KLEN, A, B, RESULT;
BOOT() {
    COLDBOOT: INPUT KSTART, KLEN;
    for (r = 0; r < KLEN; r++) input MEM[KSTART+r];
    WARMBOOT: input A, B;
            jump to MEM[KSTART];
    KEXIT:
            suspend();
            jump to WARMBOOT;
}

; sample kernel
KERNEL() {
        RESULT = A * B;
        OUTPUT RESULT
        jump to KEXIT;
}
```

The **address encoding** inside memory is done as follows:

[memory_location] = **address*2**

If the code address stored in the memory is less than or equal to 16 then it indicates register addressing. If it is greater than 16 then it memory addressing.

In the VM, the **values** are encoded as follows:

[memory_location] = **value*2 + 1**

The virtual machine offers the user to input a binary function along with its two arguments. In order to enter the instructions to compute the function into the memory, the user must input the start of the memory location in which he chooses to write the instructions (**KSTART**) and also the number of instructions he chooses to write (**KLEN**).

The virtual machine takes these parameters as input through the use of the **IN** opcode. The bootloader does the job of taking input these parameters. The IN opcode takes in the input and encodes it as **(n*2 + 1)** to indicate that a value has been entered. It is then stored in the memory as the value **n**.

On storing the value KSTART and KLEN, the virtual machine is then ready to take the instructions as input and store it in the memory location. The VM takes input the instructions which has the opcode and the encoded addresses and values needed to perform the binary operation.

Here one must ensure that the addresses and values are encoded by following the encoding scheme as the processing unit of the virtual machine decodes each and every address and value prior to the execution of the instruction.

The memory space is 128 bytes and each memory location can store the length of an **unsigned character** which is **8 bytes**. Since the IN opcode takes the input n and encodes it to indicate the memory that a value has been input, it can be observed that only an input from **0-127** can be fed to the machine.

It is also known that the addresses are encoded by left shifting the address. Because of this we cannot feed encoded addresses more than that which indicates location 64 in the memory. The locations **0-63** can be encoded safely as we require only **7** bits to encode them. But when we reach 64, the encoded address is 128 which exceeds 7 bits and the IN opcode cannot encode this as there will be an **overflow** as the Processing unit is of 8 bits.

Hence it can be concluded that the **values** between **0-127** and **addresses** between **0-63** can be safely encoded into the memory through the use of IN opcode. This is the main reason why the kernel code has to compute the operation using only the **registers** and not access any **absolute addresses**.

VI. OBSERVATIONS AND CODE

The bootloader has instructions spanning **47** memory locations. This is written from the location 0 in the memory. Since it is unable to access the variables stored in the memory after location 64 through the IN opcode, the variables and the exit status address are stored immediately after the bootloader section, but ensured that they occupy locations within 64.

This is due to the overflow which occurs when trying to access memory location above 64 through the kernel instructions which are input via the IN opcode. So the variables required to be accessed in the kernel are stored in the memory locations before mem[63].

The following is the bootloader code:

```
// COLDBOOT
        IN, IO_NUM, R0,
        MOV, R0, addr_(KSTART),
        IN, IO_NUM, R1,
        MOV, R1, addr_(KLEN),
        MOV, addr_(KSTART), R2,

// INPUT INSTRUCTIONS
        MPC, R7,
        IN, IO_NUM, R0,
        ATP, R0, R2,
        ADD, N1, R2,
        SUB, N1, R1,
        JIF, FL_ZERO, R7,

// WARMBOOT :
        MPC, addr_(WARMBOOT),
        IN,   IO_NUM,  R1,
        IN,   IO_NUM,  R2,
        MOV,  R1,      addr_(_A),
        MOV,  R2,      addr_(_B),
        JMR, addr_(KSTART),

// KEXIT
        [KEXIT] = SUS, JMR, addr_(WARMBOOT),
        0xee, //end of instructions
```

The special memory fields used for the above bootloader to store variables and the exit status addresses are as follows:

```
#define BOOT_SIZE  47
#define KSTART          (BOOT_SIZE+1)
#define KLEN            (BOOT_SIZE+2)
#define WARMBOOT        (BOOT_SIZE+3)
#define _A              (BOOT_SIZE+4)

#define _B              (BOOT_SIZE+5)
#define RESULT          (BOOT_SIZE+6)
#define KEXIT           (BOOT_SIZE+7)
```

The kernel is written from the address specified in KSTART and spans KLEN number of instructions. The kernel instructions execute the binary function, store the result in the memory and jump to the exit status. The location at which the result has to be stored should be known by the user and also the address of the Kernel exit status. The user should also know that the arguments which the bootloader read for executing the binary function is stored in **R1** and **R2** registers. The Kernel code for **Addition** instruction is as follows:

```
2 4 2
1 107 14
10 2 14
8 1 2
1 109 0
5 0
```

The assembly code of the above defined code is as follows:

```
ADD R2 R1
MOV #107 R7
ATP R1 R7
OUT STD_OUT R1
MOV #109 R0
JMR R0
```

From the above we can see that the result is computed and also stored in a memory location. Then jump to kernel exit address to suspend the machine.

The kernel code for **multiplication** instruction is as follows:

```
3 0 0
6 6
2 2 0
3 3 4
4 1 6
1 107 14
10 0 14
8 1 0
1 109 0
5 0
5 6
```

The assembly code for the above defined code is:

```
SUB R0 R0
MPC R3
ADD R1 R0
SUB N1 R2
JIF FL_ZERO R3
MOV #107 R7
ATP R0 R7
OUT IO_NUM R0
MOV #109 R0
JMR R0
```

After taking input the instructions, the **WARMBOOT** takes place which takes in input the two arguments for the binary function and transfers control to the kernel instructions to compute the function. Once the computation is done the kernel should save the result in the memory and jump to the kernel exit state. In order to do so, the user must be notified of the address of the result and the address of the kernel exit state. It also must be ensured that these addresses are below the value 63 to be encoded safely without overflow.

VII. ISSUES FACED

Since the IN opcode could only input data of **7 bits**, the addresses above location **63** were unable to access. The bootloader contained the variables to be used in the kernel code. Although the kernel code does not use the absolute addresses of the input arguments A,B as they were stored in the registers to enable easier computation, the kernel code had to store the result in the memory location specified by the bootloader. This location was also a part of the bootloader program which was written in 47 memory locations. This was opposed to the normal convention is which the instructions are written from top to down, and the data written from bottom to top in the memory location.

The **KEXIT** status also had to be part of the **bootloader** because of IN instruction unable to feed in absolute addresses more than location 64.

VIII. RESULTS

It was concluded that the kernel instructions should not be accessing the absolute addresses of the memory and deal only with the free memory space of the registers. Although as the kernel code had to store the result and jump to KEXIT address, these special addresses were part of the bootloader program.

It was observed that on saving the image, the bootloader saved the contents of the memory and program counter in the image file. On loading the image, it started from the **WARMBOOT** and not the **COLDBOOT**. Hence the COLDBOOT is just to start the machine and fix the kernel space and instructions. The image can be executed a number of times without the requirement of booting the system again and only passing the arguments to the binary function for execution. But it was observed that the contents of the registers were lost.

IX. REFERENCES

1) IIITB LMS VM Notes Folder
- *Avm.c*
- *Bvm.c*
- *Cvm.c*
- *Dvm.c*

2) IIITB LMS VM Discussion Forum