

The Bedrock Virtual Machine

By Devin Rockwell

July 12, 2022

Contents

1	Introduction	2
1.1	Goals	2
1.2	Overview	2
2	Architecture	3
2.1	Memory	3
2.1.1	Registers	3
2.1.2	Stack	4
2.1.3	Heap	4
2.2	Byte code	4
2.2.1	Instructions	4
2.2.2	Operands	6
2.3	Executables	6
2.3.1	Sections	6
3	Assembly	8
3.1	Mnemonics	8
3.2	Operands	9
3.3	Labels	9
3.4	Data Items	10
3.5	Programs	10

Chapter 1

Introduction

1.1 Goals

Bedrock is a general purpose language vm designed with two goals in mind

1. Be fast
2. Be general dont favor one language over another

These goals mean that Bedrock's architecture is much cloaser to a physical cpu compared to most language vm's. They also mean that Bedrock provides no objects or typing.

1.2 Overview

Bedrock's architecture is register based unlike most vms which are stack based. this means that most of its instructions deal with registers rather than the stack. Bedrock does also have a stack which is mostly used for function calls and storing locals.

One of the few layers of abstraction bedrock provides over traditional cpu architectures is splitting the heap into blocks (see Subsection 2.1.3).

Chapter 2

Architecture

2.1 Memory

2.1.1 Registers

Bedrock has 256 general purpose registers numbered 0x00 to 0xFF in bytecode. Each one is 64 bits wide. Bedrock also has a number of special registers as shown in this table:

Table 2.1: Special registers

name	description	size
pc	This stores the current index into the program	64 bits
cmpflag	This stores the result of the last cmp instruction it ranges from 0x0 to 0x2 (see Table 2.2).	8 bits
sp	This stores the current stack top.	16 bits.

None of these special registers can be modified by the user directly. The values for cmpflag are shown below:

Table 2.2: Cmpflag values

value	use
0x0	equal
0x1	greater
0x2	less

2.1.2 Stack

Bedrock's stack is limited to 65535 entries. Each stack entry is 64 bits wide.

2.1.3 Heap

The heap can be any size indexable by 64 bit integers. Unlike traditional architectures each heap entry in bedrock is a block of memory rather than a single byte; This means that the heap has 2^{64} addresses with each address having up to 2^{64} bytes within it. The structure of these blocks is shown in this table:

Table 2.3: Memory block layout

name	size
length	64 bits
data	determined by length

2.2 Byte code

2.2.1 Instructions

This section defines each Opcode and its Operands for more information on the format of operands see Subsection 2.2.2. within the description of each opcode the operands are numbered op1-op4. Each instruction is shown in this table:

Table 2.4: Instructions

opcode	description	operands
0x00	halts execution	
0x01	$op1 = op2$	register, number
0x02	integer addition, $op3 = op1 + op2$	register, register, register
0x03	integer subtraction, $op3 = op1 - op2$	register, register, register
0x04	integer multiplication, $op3 = op1 * op2$	register, register, register
0x05	integer division, $op3 = op1 / op2$ and $op4 = op1 \% op2$	register, register, register, register
0x06	float addition, $op3 = op1 + op2$	register, register, register
0x07	float subtraction, $op3 = op1 - op2$	register, register, register

Continued on next page

Table 2.4: Instructions (Continued)

opcode	description	operands
0x08	float multiplication, $op3 = op1 * op2$	register, register, register
0x09	float division, $op3 = op1 \div op2$	register, register, register
0x0A	compares $op1$ and $op2$ as integers	register, register
0x0B	compares $op1$ and $op2$ as floats	register, register
0x0C	sets pc to $op1$	register
0x0D	sets pc to $op1$ if cmpflag 0x0	register
0x0E	sets pc to $op1$ if cmpflag $\neq$ 0x0	register
0x0F	sets pc to $op1$ if cmpflag = 0x1	register
0x10	sets pc to $op1$ if cmpflag = 0x1 or 0x0	register
0x11	sets pc to $op1$ if cmpflag = 0x2	register
0x12	sets pc to $op1$ if cmpflag = 0x2 or 0x0	register
0x13	allocates a memory block of length $op1$ and writes its address to $op2$	register, register
0x14	deallocates the memory block at $op1$	register
0x15	reads the byte at offset $op2$ from memory block $op1$ and writes it to $op3$	register, register, register
0x16	reads eight bytes beginning at offset $op2$ from memory block $op1$ and writes it to $op3$	register, register, register
0x17	writes the lower byte of $op3$ to offset $op2$ from memory block $op1$	register, register, register
0x18	writes $op3$ beginning at offset $op2$ from memory block $op1$	register, register, register
0x19	changes the size of block $op1$ to $op2$	register, register

Continued on next page

Table 2.4: Instructions (Continued)

opcode	description	operands
0x1A	gets the length of block op1 and writes it to op2	register, register
0x1B	prints the block op1 as a utf-8 string	register
0x1C	prints the block op1 as an integer	register
0x1D	prints the block op1 as a float	register
0x1E	prints op1 as an integer	register
0x1F	prints op1 as a float	register

For integer math overflow wraps around.

2.2.2 Operands

Bedrock has various types operands as shown in this table:

Table 2.5: Operands

name	description	width
register	represents a vm register	8 bits
number	represents an integer, IEEE 754 double, or pointer	64 bits

2.3 Executables

Executable programs are stored in the bedrock executable format (BEF). This format reserves the first 16 bytes for the header which begins with [66, 69, 70, 0, 0, 0, 0, 0]. The last eight bytes of the header are the length of the data section (see subsection 2.3.1). After the header is the data section and then code section.

2.3.1 Sections

A BEF executable has two sections the code and the data section. The data section is a list of data items. The format of each data item is shown below:

Table 2.6: Data Items

	tag	data
string	0	a null terminated utf8 string
integer	1	an unsigned 64 bit integer
float	1	a IEE 754 double precision float

The fact that floats and integers have the same tag is on purpose as their in memory representation is the same.

Chapter 3

Assembly

3.1 Mnemonics

Each opcode has a corresponding mnemonic. For what the opcodes do see subsection 2.2.1. The mnemonics are shown in this table:

Table 3.1: mnemonics

mnemonic	opcode
halt	0x00
load	0x01
add	0x02
sub	0x03
mul	0x04
div	0x05
addf	0x06
subf	0x07
mulf	0x08
divf	0x09
cmp	0x0A
cmpf	0x0B
jmp	0x0C
je	0x0D
jne	0x0E

Continued on next
page

Table 3.1: mnemonics
(Continued)

mnemonic	opcode
lg	0x0F
jge	0x10
jl	0x11
jle	0x12
alloc	0x13
dealloc	0x14
read	0x15
readn	0x16
write	0x17
writen	0x18
realloc	0x19
len	0x1A
print	0x1B
printi	0x1C
printf	0x1D
printri	0x1E
printrf	0x1F

In an instruction the operands are placed as a comma separated list after the mnemonic (e.g., load r0, 233)

3.2 Operands

Each operand type (see subsection 2.2.2) has a corresponding mapping in assembly. Register operands are represented as r followed by a number 0-255 (e.g., r0). While r255 is a valid register it is not recommended to be used as it's used by the assembler. number operands are decimal numbers with or without a decimal point (e.g., 2.3 or 233).

3.3 Labels

Labels are declared with the name followed by a colon (e.g., label:). Labels can be used as the argument to jump instructions (jmp, je, jne, jl, jle, jg, jge); This

actually generates two instructions one to load the label address to r255 and then jumps with that register. Labels that name data items (see section 3.4) can also be used as arguments to the instructions `print`, `printi`, and `printf`; Much like with jump instructions this generates two instructions.

3.4 Data Items

A data item is just a label followed by a value which is a string, float, or integer. A string is just a list of utf-8 characters surrounded in quotes. Integers and float look the same as their respective operand forms.

3.5 Programs

A complete program is of the form `".data (data items) .code (instructions)"`. So a hello world program looks like this

```
.data
    msg: "hello world"
.code
    print msg
```