The Immortal Virtual Machine Instruction Set Architecture

Thor Kristoffersen
Norwegian Computing Center

June 16, 2023

The remainder of this document is a semi-formal specification of the Immortal Virtual Machine, written in a style more or less similar to a processor data sheet.

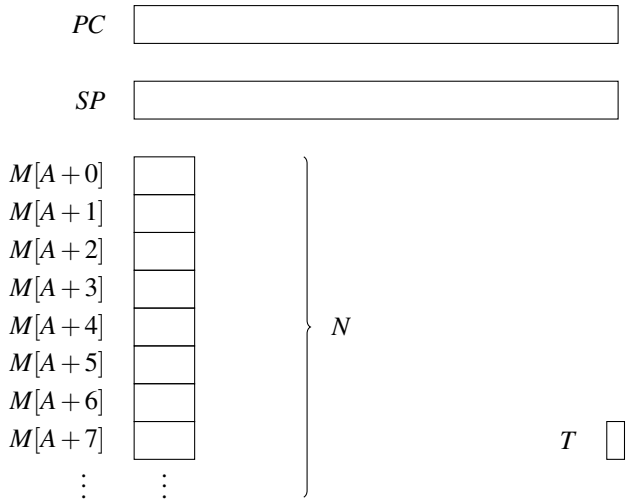
Immortal Virtual Machine
Instruction Set Architecture

1 Programming Model

The IVM is a pure stack-based machine: it has a program counter and a stack pointer, but no general-purpose registers. The programming model consists of the following elements:

- *Memory*: An array of 8-bit locations, $M[A..A+N-1]$, where $0 \leq A < 2^{64}$ and $0 < N \leq 2^{64}$.
- *Program Counter*: A 64-bit register, *PC*, that points to the next instruction to be fetched or to any immediate operands of an instruction. The initial value of *PC* is A.
- *Stack Pointer*: A 64-bit register, *SP*, that points to the top of the stack, which is the memory region from $M[SP]$ to $M[M+N-1]$, inclusive. The initial value of *SP* is $(M+N) \bmod 2^{64}$.
- *Terminate Flag*: A 1-bit flag, T, that is set to 1 when the machine has terminated. The initial value of T is 0.

These elements are shown graphically in the following figure.



2 Basic Definitions

Definition 1 (Floor)**.** $\lfloor x \rfloor$ is the unique integer such that $\lfloor x \rfloor \leq x < (\lfloor x \rfloor + 1)$.

Definition 2 (Integer division)**.**

$$x \operatorname{div} y = \left\lfloor \frac{x}{y} \right\rfloor$$

Definition 3 (Modulo)**.**

$$x \bmod y = x - y \left\lfloor \frac{x}{y} \right\rfloor, \quad y > 0$$

Definition 4 (Bit value)**.** For any integer value, x, the notation $x.\text{bit}[i]$ refers to the value of bit i in x.

Definition 5 (Octet value)**.** For any integer value, x, the notation $x.\text{octet}[i]$ refers to the integer made up of the bit sequence from $x.\text{bit}[8i+7]$ to $x.\text{bit}[8i]$, inclusive.

3 Basic Functions

The following functions are needed by some instructions. For each function, its arguments and its result are all 64-bit integer values, except where otherwise noted.

Definition 6 (Conditional)**.**

$$\text{if}(e,c,a) = \begin{cases} c & \text{if } e \text{ is true} \\ a & \text{otherwise} \end{cases}$$

Definition 7 (Addition)**.**

$$\mathrm{add}(x,y) = (x+y) \bmod 2^{64}$$

Definition 8 (Multiplication)**.**

$$\mathrm{mul}(x,y) = (xy) \bmod 2^{64}$$

Definition 9 (Integer division)**.**

$$\mathrm{div}(x,y) = \begin{cases} q \mid x = qy + r \wedge 0 \leq r < y & \text{if } x > 0 \wedge y > 0 \\ 0 & \text{otherwise} \end{cases}$$

Definition 10 (Integer remainder)**.**

$$\mathrm{div}(x,y) = \begin{cases} r \mid x = qy + r \wedge 0 \leq r < y & \text{if } x > 0 \wedge y > 0 \\ 0 & \text{otherwise} \end{cases}$$

Definition 11 (Binary power)**.**

$$\mathrm{pow2}(x) = \begin{cases} 2^x & \text{if } x < 64 \\ 0 & \text{otherwise} \end{cases}$$

Definition 12 (Bitwise boolean "and")**.**

$$\mathrm{and}(x,y) = z \mid \forall i \in \{0,\ldots,63\} \mid z.\mathrm{bit}[i] = \begin{cases} 1 & \text{if } x.\mathrm{bit}[i] = 1 \wedge y.\mathrm{bit}[i] = 1 \\ 0 & \text{otherwise} \end{cases}$$

Definition 13 (Bitwise boolean "or")**.**

$$\mathrm{or}(x,y) = z \mid \forall i \in \{0,\ldots,63\} \mid z.\mathrm{bit}[i] = \begin{cases} 1 & \text{if } x.\mathrm{bit}[i] = 1 \vee y.\mathrm{bit}[i] = 1 \\ 0 & \text{otherwise} \end{cases}$$

Definition 14 (Bitwise boolean "not")**.**

$$\mathrm{not}(x) = z \mid \forall i \in \{0,\ldots,63\} \mid z.\mathrm{bit}[i] = \begin{cases} 1 & \text{if } x.\mathrm{bit}[i] = 0 \\ 0 & \text{otherwise} \end{cases}$$

Definition 15 (Bitwise boolean "exclusive or")**.**

$$\mathrm{xor}(x,y) = z \mid \forall i \in \{0,\ldots,63\} \mid z.\mathrm{bit}[i] = \begin{cases} 1 & \text{if } x.\mathrm{bit}[i] \neq y.\mathrm{bit}[i] \\ 0 & \text{otherwise} \end{cases}$$

4 Memory Access Procedures

Five basic procedures for memory access are used as building blocks for the instructions.

4.1 Pseudocode Elements

We introduce the following pseudocode elements to describe the procedures in this section.

- $x := v$
  Assign x the value v.
- **var** $x := v$
  Declare variable x, assigning it the value of v.
- **for** i **in** $m \ldots n$ **do** $S(i)$
  Evaluate $S(i)$ $n-m$ times, with i successively bound to every integer in the range $\{m,\ldots,n-1\}$.
- **return** v
  Return the value of v.

4.2 General Memory Access Operations

There are two basic procedures for general memory access that instructions can use to store integers to or load integers from a given memory address. The memory is 8 bits wide, and integers can be 8, 16, 32, or 64 bits wide, so they are stored from the given memory address in little-endian format. An 8-bit integer is trivially mapped to the specified memory address.

The procedure store$(n,\ a,\ x)$ stores an integer, x, as n octets starting at memory address a. It is defined in pseudocode as follows:

$$\text{store}(n,\ a,\ x)\ \equiv\ \textbf{for}\ i\ \textbf{in}\ 0\ldots n\ \textbf{do}\ M[a+i]\ :=\ x.\text{octet}[i] \tag{1}$$

The procedure load$(n,\ a)$ returns an integer loaded from n octets starting at memory address a. It is defined in pseudocode as follows:

$$\begin{aligned}\text{load}(n,\ a)\ \equiv\ &\textbf{var}\ x\ :=\ 0\\ &\textbf{for}\ i\ \textbf{in}\ 0\ldots n\ \textbf{do}\ x.\text{octet}[i]\ :=\ M[a+i]\\ &\textbf{return}\ x\end{aligned} \tag{2}$$

4.3 Stack Operations

The stack operations are defined in terms of the general memory access operations, using the stack pointer as the memory address. All stack operations work on 8 octets at a time, so arguments and results are assumed to be 64-bit integers. For this reason the stack operations also decrement and increment the stack pointer in multiples of 8.

The procedure push(x) pushes an integer, x, on the stack. It is defined in pseudocode as follows:

$$\begin{aligned}\text{push}(x)\ \equiv\ &SP\ :=\ (SP-8)\bmod 2^{64}\\ &\text{store}(8,\ SP,\ x)\end{aligned} \tag{3}$$

The procedure pop() returns an integer popped off the stack. It is defined in pseudocode as follows:

$$\begin{aligned}\text{pop}()\ \equiv\ &\textbf{var}\ x\ :=\ \text{load}(8,\ SP)\\ &SP\ :=\ (SP+8)\bmod 2^{64}\\ &\textbf{return}\ x\end{aligned} \tag{4}$$

4.4 Fetch Operation

The procedure fetch(n) fetches n octets relative to the program counter, incrementing it by the same number. It is used both to fetch instructions and to fetch immediate operands.

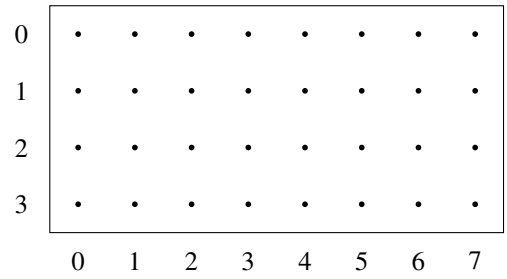
$$\begin{aligned}\text{fetch}(n)\equiv\ &\textbf{var}\ x\ :=\ \text{load}(n,\ PC)\\ &PC\ :=\ (PC+n)\bmod 2^{64}\\ &\textbf{return}\ x\end{aligned} \tag{5}$$

5 Device Access Procedures

This section describes the procedures for device access. Since the devices interface the machine with the real world, the semantics can be described only informally.

5.1 Image Input

The *Image Input* device allows the machine to consume an image as a two-dimensional array of points of light intensity values. The following figure shows an example of such an array, consisting of 32 sampling points arranged in 8 columns and 4 rows.



As shown, both columns and rows are numbered consecutively, starting at 0. The spacing between the sampling points must be uniform in both horizontal and vertical directions, and an anti-aliasing filter must be employed to limit the bandwidth of the image to satisfy the Nyquist-Shannon sampling theorem.

Each picture element detects the intensity of light transmitted or reflected at a sampling point in that particular position of the image, represented as one of 256 intensity levels, from 0 (minimum intensity) to 255 (maximum intensity). Values between 0 and 255 represent intermediate intensities between these extremes.

Definition 16 (Read frame)**.** The readframe() operation reads a new frame and returns the number of columns, c, and the number of rows r.

$$(c, r) = \text{readframe}()$$

Definition 17 (Read pixel)**.** The readpixel() operation returns the intensity, z, of the point at column x and row y.

$$z = \text{readpixel}(x, y)$$

5.2 Image Output

The *Image Output* device allows the machine to produce an image represented as a two-dimensional array of points of color space values. Moving images can be produced as a sequence of images.

Definition 18 (New frame)**.** The newframe() operation finishes and renders the frame constructed so far, and it sets the width of the next frame to w, the height to h, and the sample rate to r.

$$\text{newframe}(w, h, r)$$

Definition 19 (Set pixel)**.** The setpixel() operation sets the red value to r, the green value to g, and the blue value to b for the pixel at column x and row y.

$$\text{setpixel}(x, y, r, g, b)$$

5.3 Audio Output

The *Audio Output* device allows the machine to produce a two-channel audio signal encoded digitally using Linear Pulse Code Modulation. The device must create an audio signal passing through a series of magnitude values specified by the program. The bandwith of this audio signal must be less than half of the sampling frequency. Each channel value is in the range $\{0, \ldots, 2^{16} - 1\}$.

Definition 20 (Add sample)**.** The addsample() operation sets the audio signal magnitude of the left channel to l and the one of the right channel to r.

$$\text{addsample}(l, r)$$

5.4 Text Output

The *Text Output* device allows the machine to produce a stream of text.

Definition 21 (Put character)**.** The putchar() operation produces the character with Unicode code point c.

$$\text{putchar}(c)$$

5.5 Octet Output

The *Text Output* device allows the machine to produce a stream of 8-bit numbers.

Definition 22 (Put byte)**.** The putbyte() operation produces the octet x.

$$\text{putbyte}(x)$$

5.6 Text Input

The *Text Input* device allows the machine to read a stream of text.

Definition 23 (Read character)**.** The readchar() operation reads a new character, whose Unicode code point is c.

$$c = \text{readchar}()$$

6 Instruction Semantics

The following table summarizes the instruction semantics.

Hex	Mnemonic	Comment	Immediate	Pop	Explicit effects	Push
00	EXIT	Stop execution	–	–	$T := 1$	–
01	NOP	No operation	–	–	–	–
02	JUMP	Jump to address	–	a	$PC := a$	–
03	JZ_FWD	Jump forward on zero	$(1)d$	x	$PC := PC + \text{if}(x = 0, d, 0)$	–
04	JZ_BACK	Jump backward on zero	$(1)d$	x	$PC := PC - \text{if}(x = 0, d + 1, 0)$	–
05	SET_SP	Set stack pointer	–	a	$SP := a$	–
06	GET_PC	Get program counter	–	–	–	PC
07	GET_SP	Get stack pointer	–	–	–	SP
08	PUSH0	Push literal zero	–	–	–	0
09	PUSH1	Push 1 immediate octet	$(1)x$	–	–	x
0A	PUSH2	Push 2 immediate octets	$(2)x$	–	–	x
0B	PUSH4	Push 4 immediate octets	$(4)x$	–	–	x
0C	PUSH8	Push 8 immediate octets	$(8)x$	–	–	x
10	LOAD1	Load 1 memory octet	–	a	–	load(1, a)
11	LOAD2	Load 2 memory octets	–	a	–	load(2, a)
12	LOAD4	Load 4 memory octets	–	a	–	load(4, a)
13	LOAD8	Load 8 memory octets	–	a	–	load(8, a)
14	STORE1	Store 1 memory octet	–	a,x	store(1, a, x)	–
15	STORE2	Store 2 memory octets	–	a,x	store(2, a, x)	–
16	STORE4	Store 4 memory octets	–	a,x	store(4, a, x)	–
17	STORE8	Store 8 memory octets	–	a,x	store(8, a, x)	–
20	ADD	Add	–	y,x	–	add(x,y)
21	MULT	Multiply	–	y,x	–	mul(x,y)
22	DIV	Divide	–	y,x	–	div(x,y)
23	REM	Find remainder	–	y,x	–	rem(x,y)
24	LT	Less than	–	y,x	–	if$(x < y, -1, 0)$
28	AND	Bitwise “and”	–	y,x	–	and(x,y)
29	OR	Bitwise “or”	–	y,x	–	or(x,y)
2A	NOT	Bitwise “not”	–	x	–	not(x,y)
2B	XOR	Bitwise “exclusive or”	–	y,x	–	xor(x,y)
2C	POW2	Binary power	–	x	–	pow2(x)
30	CHECK	Check machine version	–	x	$T := \text{if}(x > 2, 1, 0)$	–
F8	READ_CHAR	Read character	–		$c := \text{readchar}()$	c
F9	PUT_BYTE	Put byte	–	x	putbyte(x)	–
FA	PUT_CHAR	Put character	–	c	putchar(c)	–
FB	ADD_SAMPLE	Put audio sample	–	r,l	addsample(l,r)	–
FC	SET_PIXEL	Put pixel	–	b,g,r,y,x	setpixel(x,y,r,g,b)	–
FD	NEW_FRAME	Output frame	–	r,h,w	newframe(w,h,r)	–
FE	READ_PIXEL	Get pixel	–	x,y	$z :=$ readpixel(x,y)	z
FF	READ_FRAME	Input frame	–	–	$(c,r) :=$ readframe()	c,r

The instruction cycle proceeds as follows:

1. Execute $c :=$ fetch(1), and locate the table entry whose “Hex” column value is c.
2. Execute $x :=$ fetch(n) for every variable, $(n)x$, in the “Immediate” column of the entry.
3. Execute $v :=$ pop() for every variable v listed in the “Pop” column of the entry.
4. Execute all operations listed in the “Explicit effects” column of the entry.
5. Execute push(e) for every expression e listed in the “Push” column of the entry.

This cycle is repeated until $T = 1$.