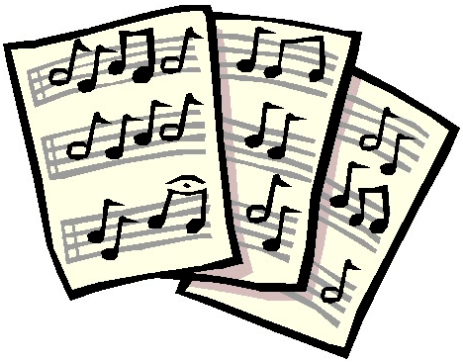
Duckpond Industries

The Ballad Language



1 Ballad

Ballad is a programming language expressed in music. It is an 'interpreted', Turing complete language which does not follow the von Neumann architecture. This is the case insofar as code memory is separate from static, non-executable memory as well as the actual, active memory. The Ballad VM has a series of registers as well as a stack, in which active data is processed.

The code memory cannot be modified during the execution of a program, nor can the static memory section. These are not placed on a heap or stack, as in a more conventional program, but are maintained separately from the active memory. The static memory can be of arbitrary size, but the code memory is restricted to a maximum of 256 bytes. The implementation of the Ballad VM should reflect this.

The stack is similarly limited in size to 256 bytes. It can be pushed and popped from, but it can also be addressed and written to directly in the style of a stack of fixed length backed by an array. A stack pointer internal to the VM should track the location of the top of the stack; the bottom is fixed to the zero position and the stack grows upwards. The stack is addressed by Ballad instructions ranging from 0x00 (the bottom) to 0xFF (the largest value).

All data is, by default, an unsigned eight bit integer. Modifications to this may be made by the program (for example, implementing two's complement or the like), but the Ballad VM should treat all numbers as such for the purpose of calculating overflow.

There are two sets of registers which Ballad uses. These are the math registers and the swap registers. One set is designed for and used by all arithmetic operations, the other set is designed for moving data between locations and as a general purpose register set. There are eight registers of each type and are addressed by a byte of value 0x00 to 0x07.

1.1 Decoding

Ballad programs consist of static data sections and code sections which are expressed as music in (at least) two-track MIDI files. The first track consists of the static memory section and the second of the code section. Other tracks are ignored.

Within a track, interpretation is ignored until one whole note (4 beats in 4/4) is found by the Ballad VM. All Ballad tracks must have this whole note at the beginning of a valid section, code or data. This note is used to decode the rest of the section, by providing the appropriate key, or MIDI note number. This key is used to demodulate the rest of the song, which is restricted to notes on the major scale, based on this key.

Demodulation occurs by first transposing the key of song to a common key (typically by subtracting the MIDI number of the tonic note mentioned above from the MIDI number of the given note to decode). The only relevant note in a chord is its root; all higher notes are ignored in this version of Ballad. Once the song is in a uniform key, it is necessary to obtain the offset *in the scale* of

the given note to decode (as opposed to its raw MIDI number). In any given Ballad program, there are only sixteen notes of the major scale given as chord roots or individual notes. To obtain Ballad bytecode from a Ballad song, it is necessary to map these to bytes.

The mapping of notes to bytes is done by first obtaining the aforementioned offset. This yields one of sixteen numbers, or 0x0 to 0xF. Taken in pairs, from the beginning of the Ballad song to the end, then converted into bytes, Ballad bytecode is obtained. For example, if the whole note at the start of one of the tracks was 'A-0', then the next two notes were 'A-0' then 'D-1', the resulting hex byte would be 0x0a, or newline in ASCII (where 0 represents the first octave, 1 represents the one above it). By applying this to each of the first two tracks of the Ballad song, a sequence of bytes can be obtained.

1.2 Static Memory

Static memory is used to provide predetermined input to a Ballad program. In other languages, this would usually take the form of constant character strings. It is indexed from zero by Ballad instructions and can be loaded via the static loading commands.

1.3 Code Memory

Code memory is a linear chain of instructions and their arguments. There are instructions which take between one and three arguments. Each instruction to a Ballad argument is one byte long, which makes calls fairly straightforward.

2 Ballad Instructions

This section describes the various Ballad instructions, their calling conventions and what registers are accessed by a given call, if any. Math registers are referenced as *m*, the general purpose, or swap registers, are referenced by *s* and immediate instructions (in unsigned single-byte representations) are indicated by *im*. If an instruction accepts no arguments, *None* is used.

2.1 Mathematical Instructions

All arithmetic operations imply unsigned integer arithmetic and overflow where appropriate.

Table 1: Mathematical Instructions

Instruction	Opcode	Arguments	Semantics
add	1	m0, m1	Add the values stored in m0 to m1 and store the result in m0
sub	2	m0, m1	Subtract the values stored in m1 from m0 and store the result in m0
mul	3	m0, m1	Multiply the values stored in m0 to m1 and store the result in m0
div	4	m0, m1	Divide m0 by m1 and store the result in m0
xor	5	m0, m1	Perform a bitwise exclusive or on the values stored in m0 to m1 and store the result in m0
and	6	m0, m1	Perform a bitwise 'and' on the values stored in m0 to m1 and store the result in m0
or	7	m0, m1	Perform a bitwise 'or' on the values stored in m0 to m1 and store the result in m0
or	8	m0, m1	Perform a bitwise inverse (0b00000000 $\rightarrow$ 0b11111111) on the values stored in m0 to m1 and store the result in m0
inc	9	m0	Increment the value in m0 by one and save it to m0
dec	10	m0	Decrement the value in m0 by one and save it to m0

2.2 Jump Instructions

These instructions change the instruction pointer (IP) in some way. Offsets range from 0 (the start of the code section) and can end at 255 (the end of the code section). When given, an immediate (*im*) value is an offset.

Table 2: Jump Instructions

Instruction	Opcode	Arguments	Semantics
jmp	11	im	Set the IP to the given offset
jeq	12	m0, m1, im	If the value in m0 is equal to the value stored in m1, set the IP to the offset
jne	13	m0, m1, im	If the value in m0 is not equal to the value stored in m1, set the IP to the offset
jlt	14	m0, m1, im	If the value in m0 is less than the value stored in m1, set the IP to the offset
jgt	15	m0, m1, im	If the value in m0 is greater than the value stored in m1, set the IP to the offset
jlte	16	m0, m1, im	If the value in m0 is less than or equal to the value stored in m1, set the IP to the offset
jgte	17	m0, m1, im	If the value in m0 is greater than or equal to the value stored in m1, set the IP to the offset
ret	28	m0	Set the IP to the value stored in m0

2.3 Memory Instructions

These instructions deal with moving data in and out of static memory and the stack. Typically, the value to move is stored in an s register and the offset in the target memory location is stored in an m register.

Table 3: Memory Instructions

Instruction	Opcode	Arguments	Semantics
push	18	s0	Put the value stored in s0 at the top of the stack, then increment the stack pointer
pop	19	s0	put the value on the top of the stack into s0, then decrement the stack pointer
lstat	23	s0, m0	Fetch the byte stored in static memory from the offset stored in m0 and put it into s0
stget	24	s0, m0	Fetch the value stored on the stack at the offset stored in m0 and put it into s0
stput	25	m0, s0	Put the value stored in s0 onto the stack at the offset stored in m0

2.4 Move Instructions

These instructions handle moving data between registers and putting immediate data into registers. They also handle things like immediate loading from within the code. As usual, assume unsigned bytes.

Table 4: Move Instructions

Instruction	Opcode	Arguments	Semantics
movim	30	m0, im	This stores the value of the immediate into the math register indexed by m0
movis	29	s0, im	This stores the value of the immediate into the swap register indexed by s0
movrm	21	m0, s0	This moves the value stored in the swap register s0 into the math register indexed by m0
movrs	21	s0, m0	This moves the value stored in the math register m0 into the swap register indexed by s0

2.5 Utility Instructions

These instructions handle 'system' level instructions, reading and writing from the user.

Table 5: Move Instructions

Instruction	Opcode	Arguments	Semantics
read	27	m0, m1	This command reads in bytes numbering the value stored in m1 onto the stack at the address (offset) given by the value stored in m0
print	26	s0, s1	This command prints out bytes numbering the value stored in s1 from the stack at the address (offset) given by the value stored in s0