Stack

A stack-based virtual machine for physical devices.

19th September 2018

Stack (name may change) is a stack-based virtual machine for physical computing. The final system will consist of (i) a web-based development environment with a simulator, (ii) a USB HID programming interface, and (iii) various hardware implementations of the virtual machine supporting output devices such as RGB LED, sounders, and motors. The aim is to provide a fun programming platform using low-cost battery-powered physical devices. Being stack-based, building compliers for custom domain-specific languages is relatively straightforward. To date, parts (i) and (ii) are complete, and (iii) is currently being prototyped.

Stack Instruction Set

The Stack instruction set consists of two parts: (i) a core set of instructions that all virtual machines must implement, and (ii) an optional set which provide specific input and output functionality. The optional instructions encode their stack manipulation within the opcode, so backward compatibility is maintained as additional optional instructions are added to the instruction set. Optional instructions can easily be added to the assembler and compiler without their implementation details being known.

Core Instructions

The core instructions are shown below. The instructions manipulate two stacks: the operand stack, S, and the return address stack, R, The sizes of the stacks is denoted by $\#S$ and $\#R$ respectively. The top of both stacks is referred to by the index 1.

The program, and any associated data, is encoded as a set of bytes, P, in the program space, indexed from 0, with the size of the program being $\#P$ bytes.

Instruction		Effect	Description	Preconditions
`0x00`	`ADD`, `+`	ab — x	Pops a and b from the operand stack and pushes back $a + b$.	$\#S \geq 2$
`0x01`	`SUB`, `-`	ab — x	Pops a and b from the operand stack and pushes back $a - b$.	$\#S \geq 2$
`0x02`	`MUL`, `*`	ab — x	Pops a and b from the operand stack and pushes back $a \times b$.	$\#S \geq 2$
`0x03`	`DIV`, `/`	ab — x	Pops a and b from the operand stack and pushes back $a \div b$.	$\#S \geq 2$, $b > 0$

`0x04`	`MOD`	ab — x	Pops a and b from the operand stack and pushes back a mod b	$\#S \geq 2$, $b > 0$
`0x05`	`INC`	a — x	Pops a from the operand stack and pushes back $a + 1$.	$\#S \geq 1$
`0x06`	`DEC`	a — x	Pops a from the operand stack and pushes back $a - 1$.	$\#S \geq 1$
`0x07`	`MAX`	ab — x	Pops a and b from the operand stack and pushes back $\mathrm{MAX}(a, b)$.	$\#S \geq 2$
`0x08`	`MIN`	ab — x	Pops a and b from the operand stack and pushes back $\mathrm{MIN}(a, b)$.	$\#S \geq 2$
`0x09`	`LT`, `<`	ab — x	Pops a and b from the operand stack and pushes back 1 if $a < b$, and 0 otherwise.	$\#S \geq 2$
`0x0A`	`LE`, `<=`	ab — x	Pops a and b from the operand stack and pushes back 1 if $a \leq b$, and 0 otherwise.	$\#S \geq 2$
`0x0B`	`EQ`, `=`	ab — x	Pops a and b from the operand stack and pushes back 1 if $a = b$, and 0 otherwise.	$\#S \geq 2$
`0x0C`	`GE`, `>=`	ab — x	Pops a and b from the operand stack and pushes back 1 if $a \geq b$, and 0 otherwise.	$\#S \geq 2$
`0x0D`	`GT`, `>`	ab — x	Pops a and b from the operand stack and pushes back 1 if $a > b$, and 0 otherwise.	$\#S \geq 2$
`0x0E`	`DROP`	a —	Pops a from the operand stack and discards it.	$\#S \geq 1$
`0x0F`	`DUP`	a — aa	Pops a from the operand stack and pushes back two copies.	$\#S \geq 1$
`0x10`	`NDUP`	$bc2$ — bcb	Pops N from the operand stack, and pushes back to the stack a copy of the value at depth N in the stack. `1 NDUP` is equivalent to `DUP`.	$N > 0$, $\#S > N$
`0x11`	`SWAP`	ab — ab	Pops a and b from the operand stack and pushes them back in the reverse order.	$\#S \geq 2$
`0x12`	`ROT`	abc — bca	Rotates top three values on the stack (bottom to top).	$\#S \geq 3$
`0x13`	`NROT`	$abcd4$ — $bcda$	Pops N from the stack, and rotates top N values on the stack. `3 NROT` is equivalent to `ROT`, `2 NROT` is equivalent to `SWAP`, and `1 NROT` is equivalent to `NOP`.	$N > 0$, $\#S > N$
`0x14`	`TUCK`	abc — cab	Rotates top three values on the stack (top to bottom).	$\#S \geq 3$
`0x15`	`NTUCK`	$abcd4$ — $dabc$	Pops N from the stack, and tucks top N values on the stack. `3 NTUCK` is equivalent to `TUCK`, `2 NTUCK` is equivalent to `SWAP`, and `1 NTUCK` is equivalent to `NOP`.	$N > 0$, $\#S > N$
`0x16`	`SIZE`	— x	Pushes $\#S$ onto the operand stack.	—
`0x17`	`NRND`	n — x	Pops N from the operand stack, and pushes back onto the stack a random number between 0 and $N - 1$.	$\#S \geq 1$, $N > 1$

`0x18`	`PUSH`	— x	Takes the 8-bit two's complement number encoded in the instruction, and pushes this onto the operand stack.	—
`0x19`	`PUSH`	— x	Takes the 16-bit two's complement little-endian number encoded in the instruction, and pushes this onto the operand stack.	—
`0x1A`	`FETCH`	a — x	Pops an address from the operand stack, and pushes back the 16-bit two's complement little-endian number found there.	$\#S \geq 1$, $0 \leq a < \#P - 1$
`0x1B`	`CALL`	a —	Pushes the current program counter value onto the return address stack, pops the destination from the operand stack, and jumps to that address.	$\#S \geq 1$, $0 \leq a < \#P - 1$
`0x1C`	`RET`	—	Pops an address from the return address stack, and jumps there.	—
`0x1D`	`JMP`	a —	Pops the destination from the operand stack, and jumps to that address.	$\#S \geq 1$, $0 \leq a < \#P - 1$
`0x1E`	`CJMP`	ab —	Pops the destination, b, from the operand stack, and jumps to that address if a ! $= 0$.	$\#S \geq 2$, $0 \leq b < \#P - 1$
`0x1F`	`WAIT`	d —	Pops d from the operand stack, and performs a blocking wait of d milliseconds	$0 \leq d \leq 32767$
`0x20`	`HALT`	—	Halts program execution	—

All core instructions are encoded as a single byte with the exception of `PUSH` which comes in two variants, opcodes `0x18` and `0x19` which are followed by either one or two bytes representing an 8-bit or a 16-bit little-endian two's complement number. The internal representation of the address and operand stack are 32-bit integers, and ll arithmetic operations saturate, rather than overflow.

Optional Instructions

Optional instructions are encoded as two bytes: the first byte is the opcode, and the second byte is a representation of how the instruction manipulates the operand stack. The most four most significant bits represent the number of values that are pushed to the operand stack, and the four least significant bits represent the number of values that are popped from the operand stack. The opcode of all optional instructions is `0x80` or greater. In this way, the virtual machine can execute optional instructions that it doesn't actually support, by simply popping and pushing dummy values to the operand stack.

Instruction		Effect	Description	Preconditions
`0x80`,`0x01`	`SLEEP`	d —	Pops d from the operand stack and sleeps in a low power mode for d seconds. All outputs are disabled, and on waking program execution starts from address 0 with empty operand and return address stacks.	$\#S \geq 1$, $0 \leq d \leq 32767$
`0x81`,`0x01`	`TONE`	f —	Pops f from the operand stack and starts playing a tone of frequency f Hz. Calling `0 TONE` will silence the tone.	$\#S \geq 1$, $0 \leq f \leq 32767$

`0x82,0x02`	`BEEP`	fd —	Pops f and d from the operand stack. Plays a tone of frequency f Hz for d milliseconds. This is a blocking call and program execution waits until the tone has completed.	$\#S \geq 2$, $0 \leq f \leq 32767$, $0 \leq d \leq 32767$
`0x83,0x03`	`RGB`	rgb —	Pops r, g and b from the operand stack and sets the LED colour appropriately. Calling `0 0 0 RGB` will turn off the LED.	$\#S \geq 3$, $0 \leq r, g, b \leq 255$
`0x84,0x01`	`COLOUR`	c —	Pops c from the operand stack and sets the LED colour appropriately. Colours are encoded as RGB using the three least significant bits, so 0 - off, 1 - blue, 2 - green, 3 - cyan, 4 - red, 5 - magenta, 6 - yellow and 7 - white.	$\#S \geq 1$, $0 \leq c \leq 7$
`0x85,0x02`	`FLASH`	cd —	Pops c and d from the operand stack and flashes the LED with the appropriate colour for d milliseconds. This is a blocking call and program execution waits until the flash has completed.	$\#S \geq 2$, $0 \leq c \leq 7$, $0 \leq d \leq 32767$
`0x86,0x10`	`TEMP`	— t	Pushes current temperature in degrees centigrade to the operand stack.	$\#S \geq 0$
`0x87,0x30`	`ACCEL`	— xyz	Pushes current accelerometer x, y and z axis readings to the operand stack (1g = 1024). Maximum reading is 8g.	$\#S \geq 0$
`0x88,0x02`	`PIXEL`	cp —	Pops c and p from the operand stack and lights LED number p the appropriate colour, leaving other LED unchanged.	$\#S \geq 2$, $0 \leq c \leq 7$, $1 \leq p \leq 9$

Status Codes

The virtual machine uses the following status codes. Codes other than 0 and 1 indicate an error status and the execution of the program will stop.

Code	Name	Description
0	OKAY	Normal state when the virtual machine is executing code.
1	HALT	The `HALT` instruction has been executed.
2	INVALID ADDRESS	An attempt was made to decode, fetch or jump to an address which is outside of the program space.
3	INVALID INSTRUCTION	An attempt to decode an instruction failed. Typically caused by an error converting to byte code by hand, or by jumping to the data sector or inside a `PUSH` instructions.
4	INVALID OPERAND	One of the values popped from the operand stack by an instruction was invalid (e.g. attempting to divide by zero).
5	STACK OVERFLOW	Maximum stack size exceeded (either the operand stack or the return address stack).
6	STACK UNDERFLOW	An instruction attempted to pop a value from a stack, and found it to be empty (either the operand stack or the return address stack).

Stack Assembler

The Stack assembler converts assembly language to machine code as you type. Instructions are case insensitive, and numbers can be represented either as decimal values, or as 8-bit and 16-bit two's complement hexadecimal values (e.g. so that −1, `0xFF` and `0xFFFF` are equivalent). The assembler will switch between the two variants of `PUSH` depending on the number of bytes required to represent any value.

Below, is a minimal program that can be compiled and loaded into the simulator. It plays an 500 Hz note for 1 second, and then exits. The equivalent byte code consists of 9 bytes: two three-byte `PUSH` instructions encoding 500 and 1000 as little-endian 16-bit two's complement numbers, two bytes to represent the `BEEP` instruction which is part of the optional instruction set, and an additional `HALT` instruction inserted by the assembler.

```
500 1000 beep
```

(a) Assembly code.

```
0x0000: 0x19 0xF4 0x01
0x0003: 0x19 0xE8 0x03
0x0006: 0x82 0x02
0x0008: 0x1A
```

(b) Byte code.

The assembler will also resolve labels for `CALL`, `JMP`, `CJMP` and `FETCH` instructions. Labels end with a colon and are case sensitive alphanumeric names which must start with a letter. The corresponding address omits the colon. It also recognises useful constants; specifically, the colours (e.g. `black`, `blue`, `green`, `cyan`, `red`, `magenta`, `yellow` and `white`) and the frequency of musical notes (e.g. `D4`, `D#4` and `Db4`).

The code below shows the same minimal program with the `BEEP` instruction moved to a procedure which is called from the main code and is marked with a label. The equivalent byte code is 13 bytes. Note that the assembler has resolved the address of the start of the `play` procedure to be `0x0007` and this is pushed to the operand stack in little-endian format as `0x07` using opcode `0x18` before the `CALL` instruction.

```
  A4 play call
  halt
play:
  1000 beep
  ret
```

(a) Assembly code.

```
0x0000: 0x19 0xEE 0x01
0x0003: 0x18 0x07
0x0006: 0x1B
0x0007: 0x20
0x0008: 0x19 0xE8 0x03
0x000B: 0x82 0x02
0x000D: 0x1C
```

(b) Byte code.

The assembler also allows for the storage of array type data within the program code. Values within this area and may be pushed to the operand stack using the `FETCH` instruction. Data is separated from the rest of the program by either placing it at the end of the program with a `.data` segment label, or at the start of the program between a pair of `.data` and `.code` segment labels. The address of the data can be referenced from the `.data` segment label, or from any additional label inserted in this space. The actual data is always positioned after the executable instructions, with an additional `HALT` instruction automatically inserted if required.

The assembler also allows the insertion of raw opcode. These are placed between square brackets and

should represent single byte hex values (e.g. `[0x20]` or `[0x82 0x02]`). The opcode blocks cannot be nested and can only appear in the code segment.

Procedural Music

The example code in the assembler plays the Fibonacci sequence in modulo 7, which repeats every 16 notes, and sounds quite nice. The frequencies of the 7 notes of the scale are stored in the data area.

```
.data
  B4 C5 D5 E5 F5 F#5 G5
.code
  33 6 1
loop:
  dup rot + 7 mod
  dup colour
  dup play call
  rot dec dup 4 ntuck
  0 > loop cjmp
  halt
play:
  2 * data + fetch
  200 beep
  50 wait
  ret
```

Fibonacci

Finally, here is an example of calculating the Fibonacci sequence calculated both iteratively and recursively.

```
  12 fibonacci call
  halt
fibonacci:
  dup 1 > isGreaterThanOne cjmp
  ret
isGreaterThanOne:
  0 1
loop:
  dup tuck +
  rot 1 - dup 4 ntuck
  1 > loop cjmp
  rot drop swap drop
  ret
```

(a) Iterative Fibonacci.

```
  12 fibonacci call
  halt
fibonacci:
  dup 1 > isGreaterThanOne cjmp
  ret
isGreaterThanOne:
  dup
  1 - fibonacci call
  swap
  2 - fibonacci call
  +
  ret
```

(b) Recursive Fibonacci.

The recursive calculation is quite slow. It is much more readable though.

Accelerometer

The following code calculates $\sqrt{a_x^2 + a_y^2 + a_z^2}$ using a loop to approximate the square root function to provide an estimate of the net acceleration felt by the device.

```
acceleration:
  accel
  dup * rot
  dup * rot
  dup * rot
  + + 0
loop:
  2 ndup 2 ndup
  dup * < done cjmp
  50 + loop jmp
done:
  swap drop ret
```

Future Work

Future work will demonstrate Stack on a number of different hardware platforms.