Programming Languages 3

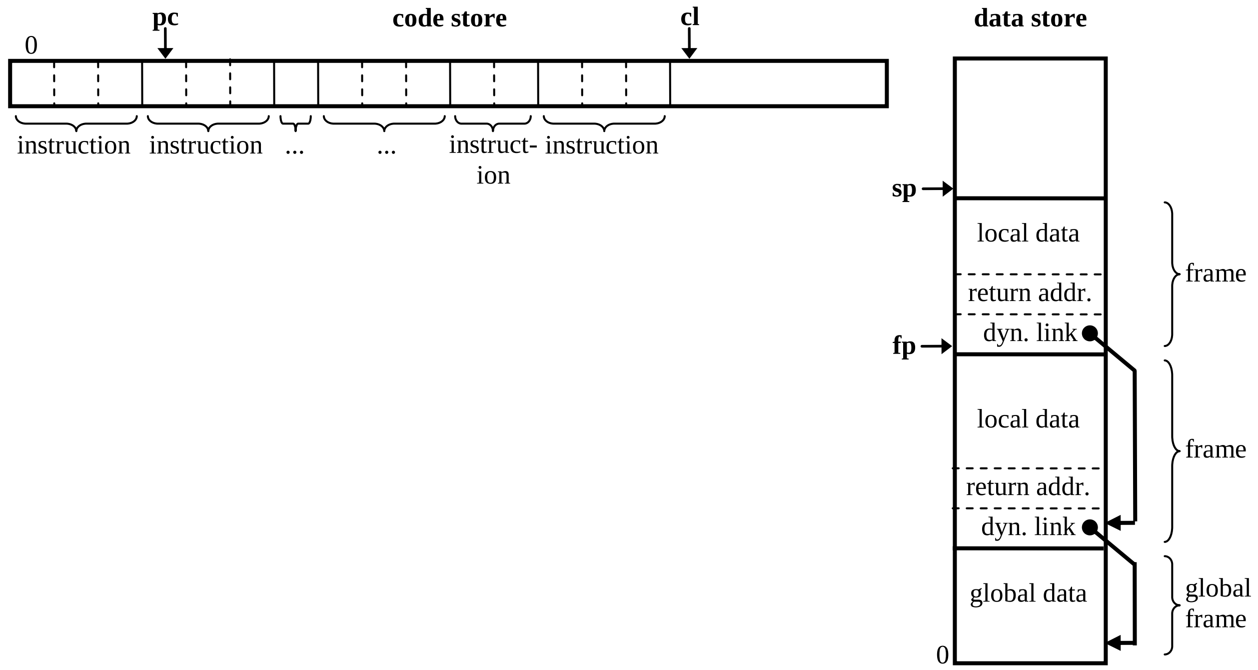
SVM Specification

SVM is a simple virtual machine. It is suitable for execution of programs in simple imperative languages.

1 Machine state

The virtual machine consists of:

- a **code store** of 32,768 bytes, which contains the program's instructions (*diagram below left*)
- a register **pc** (program counter), which points to the opcode of the next instruction to be fetched
- a register **cl** (code limit), which points to the first free byte after the stored program code
- a **data store** of 32,768 words (each of 32 bits), which contains the program's data (*diagram below right*)
- a register **sp** (stack pointer), which points to the first free word above the top of the stack
- a register **fp** (frame pointer), which points to the first word of the topmost frame
- a register **status**, which indicates whether the program is running, halted, or failed.



The data store contains a stack. The stack contains a **frame** for each currently active routine (procedure or function). Each frame contains:

- a **dynamic link**, which points to the first word of the underlying frame
- a **return address**, which points to the instruction following the `CALL` that activated the routine
- local data (arguments, local variables, expression evaluation results)

The base of the stack is always occupied by a **global frame**. The global frame contains only global data (with no dynamic link or return address).

2 Instruction set

Each instruction occupies 1, 2, or 3 bytes. The first byte of each instruction is its opcode.

Opcode	*Bytes*	*Mnemonic*	*Behaviour*
0	1+2	`LOADG` *d* (load global)	$w \leftarrow$ word at global address d; push w on to stack
1	1+2	`STOREG` *d* (store global)	pop w from stack; word at global address $d \leftarrow w$
2	1+2	`LOADL` *d* (load local)	$w \leftarrow$ word at local address (fp+d); push w on to stack
3	1+2	`STOREL` *d* (store local)	pop w from stack; word at local address (fp+d) $\leftarrow w$
4	1+2	`LOADC` *v* (load constant)	push v on to stack
6	1	`ADD` (add)	pop w_2 from stack; pop w_1 from stack; push ($w_1 + w_2$) on to stack
7	1	`SUB` (subtract)	pop w_2 from stack; pop w_1 from stack; push ($w_1 - w_2$) on to stack
8	1	`MUL` (multiply)	pop w_2 from stack; pop w_1 from stack; push ($w_1 \times w_2$) on to stack
9	1	`DIV` (divide)	pop w_2 from stack; pop w_1 from stack; push (w_1 / w_2) on to stack, discarding any remainder
10	1	`CMPEQ` (compare equal)	pop w_2 from stack; pop w_1 from stack; push (if $w_1 = w_2$ then 1 else 0) on to stack
12	1	`CMPLT` (compare less than)	pop w_2 from stack; pop w_1 from stack; push (if $w_1 < w_2$ then 1 else 0) on to stack
13	1	`CMPGT` (compare greater than)	pop w_2 from stack; pop w_1 from stack; push (if $w_1 > w_2$ then 1 else 0) on to stack
14	1	`INV` (invert)	pop w from stack; push (if $w = 0$ then 1 else 0) on to stack
15	1	`INC` (increment)	pop w from stack; push ($w + 1$) on to stack
16	1	`HALT` (halt)	status $\leftarrow$ halted
17	1+2	`JUMP` *c* (jump)	pc $\leftarrow c$
18	1+2	`JUMPF` *c* (jump if false)	pop w from stack; if $w = 0$ then pc $\leftarrow c$
19	1+2	`JUMPT` *c*	pop w from stack;

		(jump if true)	if $w \neq 0$ then pc ← *c*
20	1+2	`CALL` *c* (call)	if *c* is the address of an input/output routine then: execute that routine else: push fp (dynamic link) on to stack; push pc (return address) on to stack; fp ← address where dynamic link is stored; pc ← *c*
21	1+1	`RETURN` *r* (return)	pop result (*r* words) from stack; pop topmost frame down to address fp; fp ← dynamic link; pc ← return address; push result on to stack
22	1+1	`COPYARG` *s* (copy arguments)	move arguments (*s* words) into topmost frame, swapping with dynamic link and return address

Notes:

- *c* denotes an address within the code store
- *d* denotes a global address or local address offset within the data store
- *r* denotes the number of words to be returned by a routine
- *s* denotes the number of words to be passed into a routine
- *v* denotes a 16-bit value.

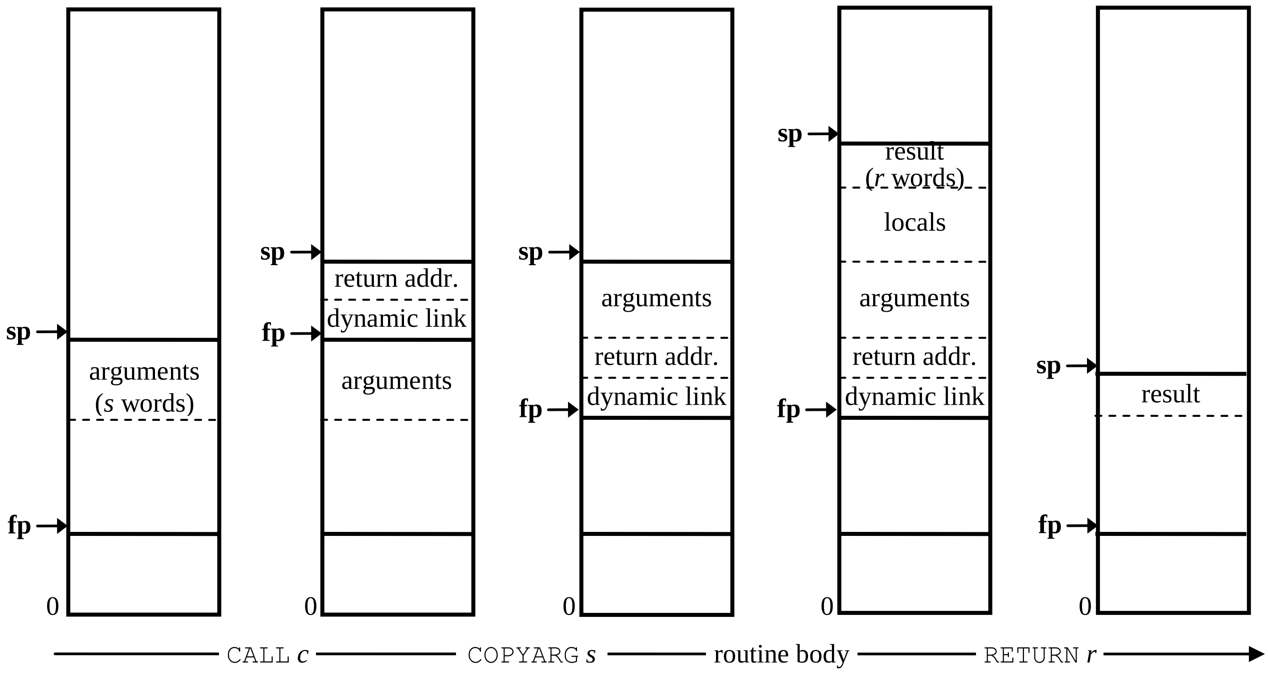
3 Routines and frames

A **routine** is a piece of code that is invoked by a `CALL` instruction. When the routine executes a `RETURN` instruction, control returns to the instruction following the `CALL`.

Throughout the routine's activation, a frame in the stack holds its local data. A `CALL` instruction pushes a new frame on to the stack. A `RETURN` instruction pops a frame off the stack.

If a routine has any arguments, the caller is required to push these arguments on to the stack immediately before executing a `CALL` instruction. The called routine should start by using the `COPYARG` instruction to move these arguments into its own frame. The `COPYARG` instruction simply swaps these arguments with the dynamic link and return address.

The effect of these instructions on the data store is illustrated in the diagram below. It is assumed that the called routine has arguments totalling *s* words and a result of *r* words.


sp
arguments
(s words)
fp
0
sp
return addr.
dynamic link
fp
arguments
0
sp
arguments
return addr.
dynamic link
fp
0
sp
result
(r words)
locals
arguments
return addr.
dynamic link
fp
0
sp
result
fp
0
CALL c
COPYARG s
routine body
RETURN r