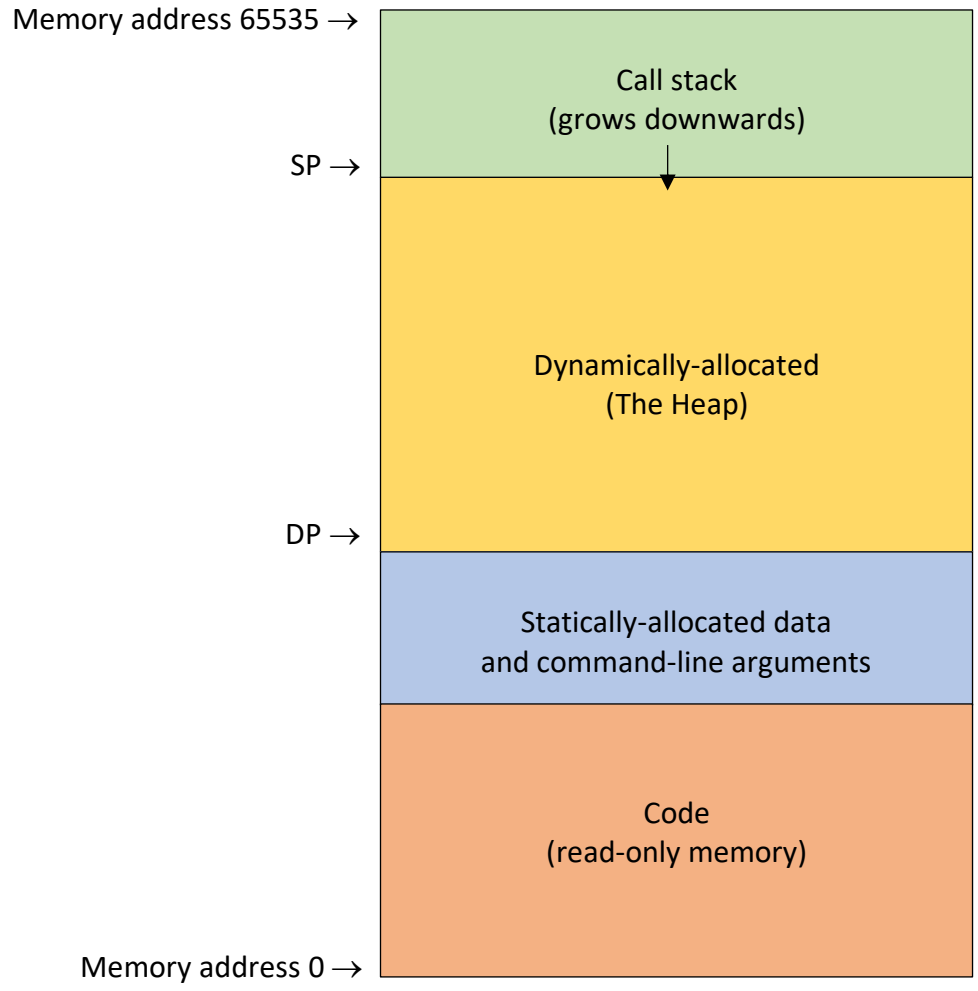
The Simple Stack Machine (SSM): Overview

The SSM Architecture

Memory: 65536 bytes. Why 65536? Because the SSM uses 16-bit (2-byte) memory addresses and $2^{16} = 65536$. The first byte of memory has address 0. The final byte of memory has address 65535. Different regions of the memory are used for different purposes:

Code region: the bottom part of memory, starting at memory address 0.
Data region (statically-allocated memory): immediately after the end of the code region. (Our compilers will use this to implement global variables.)
Dynamically-allocated memory region: immediately after the end of the statically allocated region. (Our compilers will use this to implement arrays and records.)
Call-stack region: the top part of memory, from memory address 65535 downwards. (Our compilers will use the call-stack to implement method calls.)



Registers: **PC** (Program Counter), **DP** (Dynamically-allocated-memory Pointer), **FP** (Frame Pointer), **SP** (Stack Pointer). The PC is what the SSM uses to keep track of which instruction to execute next (more about this shortly). For now, you can ignore the others.

Operand-stack (**opstack** for short). A stack of 4-byte words. Many of the SSM instructions consume (pop) items from the opstack and/or push new items on top. The arithmetic operations treat each word as a 32-bit twos-complement integer. (In general there can be *multiple* opstacks, since each frame on the call-stack also has an associated opstack. But we won't be using the call-stack for a while yet.)

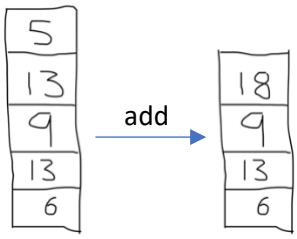
What about the CPU? The SSM is a *virtual* machine, so this is actually just code (our implementation in particular is a Java program). In essence a big switch-statement with one case for each opcode (see next section).

SSM Instructions

SSM code is a binary format: a sequence of instructions, where each instruction is itself a short sequence of bytes. Each instruction starts with a single byte – known as the *opcode* - which identifies the instruction and determines how many additional bytes form part of the complete instruction. For example, the **add** instruction is just 1-byte long, an opcode with no additional bytes, while a **jumpi** instruction is 3-bytes long, where the two bytes after the opcode specify which memory address the SSM should jump to. Some instructions use and modify main memory, some use and modify the opstack, and some do both. The full list of SSM instructions is provided in a separate document (*The SSM Instruction Set*).

More About the Operand-Stack

The SSM is a *stack-based* virtual machine. In a stack-based machine, many instructions make use of a stack of data items known as the *operand-stack* (*opstack* for short). Consider the **add** instruction: this pops two items off the opstack, adds them together, then pushes the result back on the opstack. For example:



Note that **add** only touches the top two items on the opstack; any items that happen to be below those items are left unchanged. Rather than draw pictures like the above, we will use a more compact notation (borrowed from the JVM Specification) to specify the effect of instructions on the opstack:

add: ***…, x, y → …, x+y***

In this notation:

- ***…, x, y*** on the left of the arrow indicates that add requires there to be at least two items on the opstack, with *y* being the value on top and *x* being the value immediately below *y*; the dots represent any items which might be on the opstack lower down (add doesn't care about these and won't modify them).
- ***…, x+y*** on the right of the arrow indicates that *x* and *y* have been removed from the opstack and replaced by their sum.

Of course, the SSM must also provide other instructions for getting data onto the operand stack in the first place. The most basic instruction for getting data on to the stack is **push**. A push instruction is five bytes long. The first byte is the opcode, the following four bytes make up the 4-byte word that will be pushed onto the opstack:

push ***x***: **… → …, *x***

In this notation:

- **…** on the left of the arrow represents whatever values happen to be on the opstack already (push doesn't care what they are and won't modify them)
- **…, *x*** on the right of the arrow indicates that the four-byte word ***x*** has been pushed on top of the opstack.

SSM Assembly Code

Since it is a binary format, SSM code is not human-readable (or writeable!). Instead of generating SSM binaries directly, our compilers will generate SSM *assembly* code. Assembly code is turned into SSM binary code by the SSM *assembler*. Here is a very simple SSM assembly program, together with some alternative representations of the binary code sequence that the assembler produces:

Assembly code	Byte sequence (binary)	Byte sequence (hexadecimal)	Byte sequence (decimal)

`push 1027`
`push -28`
`add`
`sysc OUT_DEC`
`halt`
	`00100100 00000000 00000000 00000100`
	`00000011 00100100 11111111 11111111`
	`11111111 11100100 00000110 00011001`
	`00000011 00000001`
		`24 00 00 04`
		`03 24 ff ff`
		`ff e4 06 19`
		`03 01`
			`036 000 000 004`
			`003 036 255 255`
			`255 228 006 025`
			`003 001`

More details about SSM assembly code are provided in a separate document (*The SSM Assembler*).