VIRTUAL MACHINE

FUNDAMENTALS

Shreetesh M

Click here to access this presentation live

MAJOR TYPES OF VIRTUAL MACHINES

MAJOR TYPES OF VIRTUAL MACHINES

System VM

- Also known as Hardware VM
- Simulates an entire computer system to behave like another system
- For example, running Mint OS inside a Windows machine
- Examples: Oracle VirtualBox, VMWare

Language VM

- Also known as Process VM
- Simulates only a runtime environment
- For example, converting Java bytecode to machine code in the JRE
- Examples: JVM, Dalvik, CLR

MAJOR TYPES OF VIRTUAL MACHINES

System VM

- Also known as Hardware VM
- Simulates an entire computer system to behave like another system
- For example, running Mint OS inside a Windows machine
- Examples: Oracle VirtualBox, VMWare

Language VM

- Also known as Process VM
- Simulates only a runtime environment
- For example, converting Java bytecode to machine code in the JRE
- Examples: JVM, Dalvik, CLR

MAJOR TYPES OF VIRTUAL MACHINES

System VM

- Also known as Hardware VM
- Simulates an entire computer system to behave like another system
- For example, running Mint OS inside a Windows machine
- Examples: Oracle VirtualBox, VMWare

Language VM

- Also known as Process VM
- Simulates only a runtime environment
- For example, converting Java bytecode to machine code in the JRE
- Examples: JVM, Dalvik, CLR

MAJOR TYPES OF VIRTUAL MACHINES

System VM

- Also known as Hardware VM
- Simulates an entire computer system to behave like another system
- For example, running Mint OS inside a Windows machine
- [illegible]

Language VM

- Also known as Process VM
- Simulates only a runtime environment
- For example, converting Java bytecode to machine code in the JRE
- [illegible]

MAJOR TYPES OF VIRTUAL MACHINES

System VM

- Also known as Hardware VM
- Simulates an entire computer system to behave like another system
- For example, running Mint OS inside a Windows machine
- Examples: Oracle VirtualBox, VMWare

VirtualBox

Language VM

- Also known as Process VM
- Simulates only a runtime environment
- For example, converting Java bytecode to machine code in the JRE
- Examples: JVM, Dalvik, CLR

MAJOR TYPES OF VIRTUAL MACHINES

System VM

- Also known as Hardware VM
- Simulates an entire computer system to behave like another system
- For example, running Mint OS inside a Windows machine
- Examples: Oracle VirtualBox, VMWare

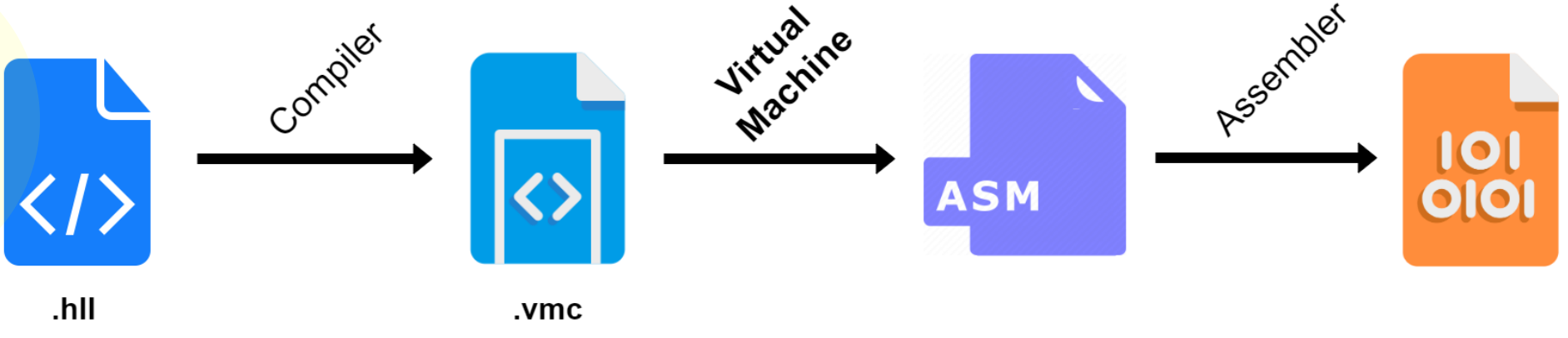
VirtualBox

Language VM

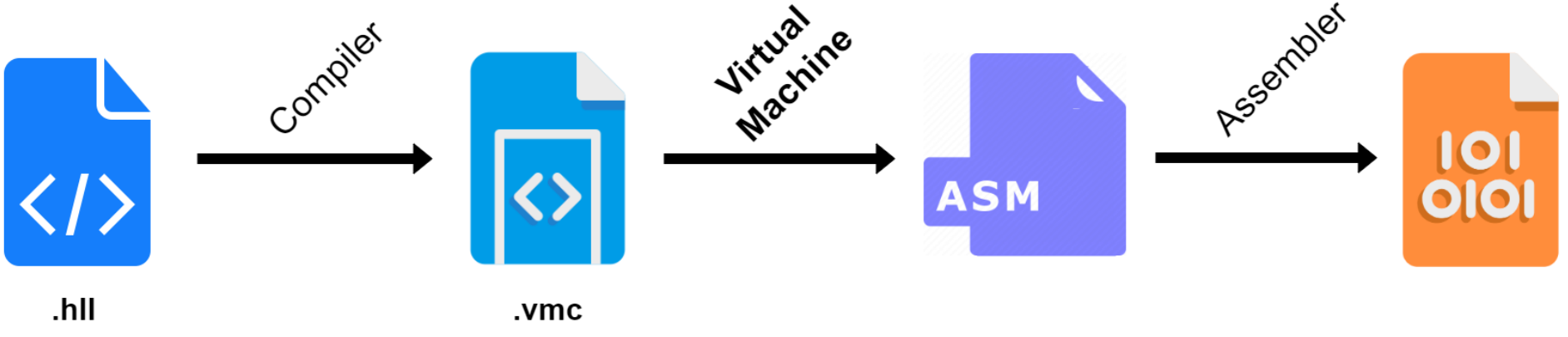
- Also known as Process VM
- Simulates only a runtime environment
- For example, converting Java bytecode to machine code in the JRE
- Examples: JVM, Dalvik, CLR

SO, WHAT SHOULD A LANGUAGE VM DO?

SO, WHAT SHOULD A LANGUAGE VM DO?



SO, WHAT SHOULD A LANGUAGE VM DO?

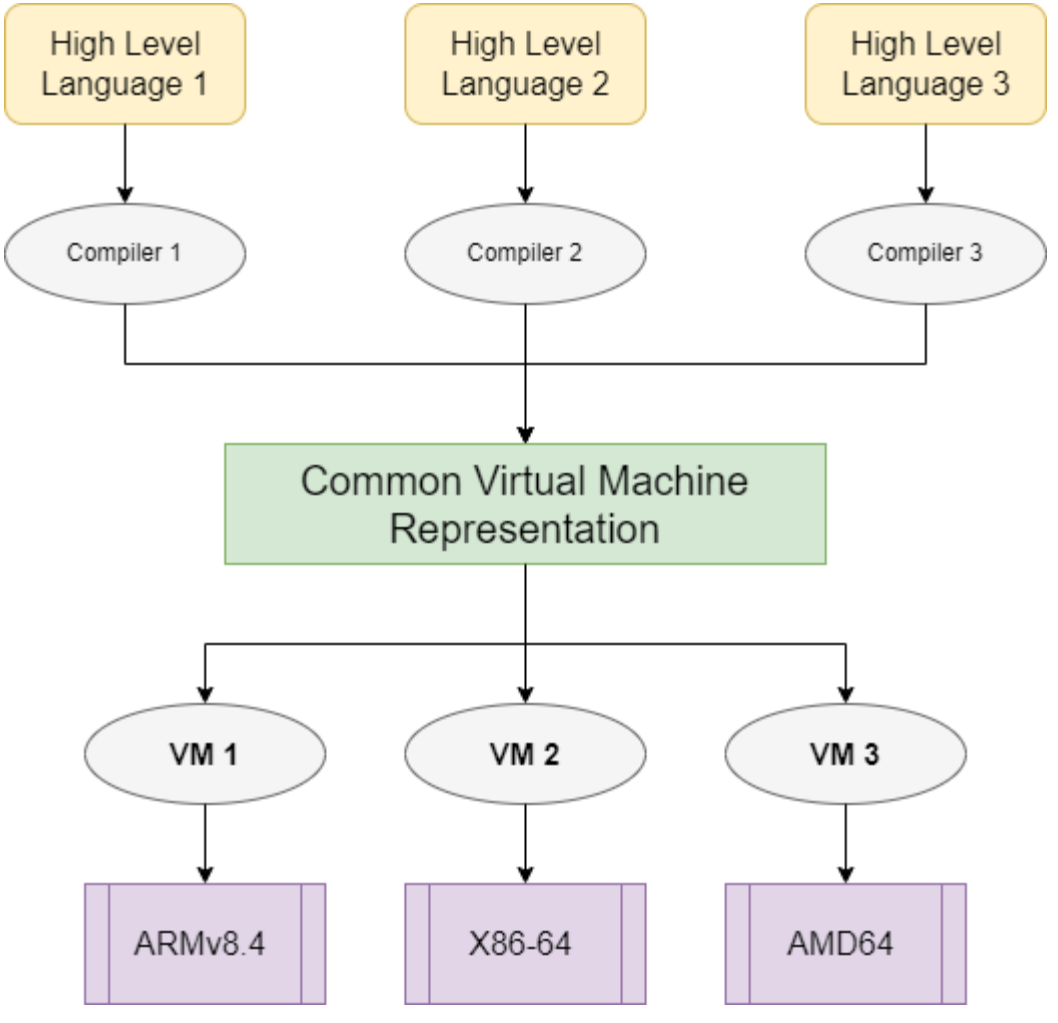


OTHER EXECUTION MODELS:

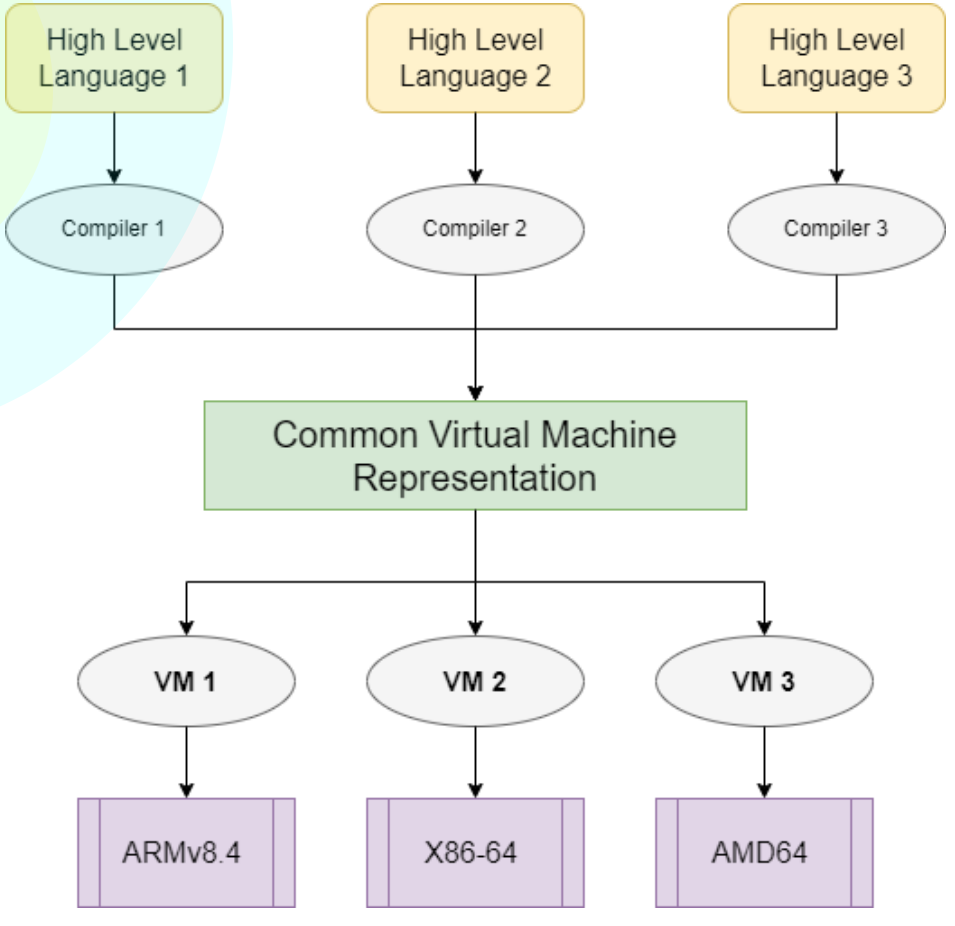
1. **Interpreted Execution:** VM directly executes `.vmc` in its environment without a specific translation. For example, Dalvik VM.
2. **Tiered Execution:** VM uses a mix of translation and direct interpetation of `.vmc`. For example, JVM.

BUT WHAT IS THE USE OF A VM?

BUT WHAT IS THE USE OF A VM?



BUT WHAT IS THE USE OF A VM?



- 9 monolithic compilers vs 3 frontend and 3 backend.
- Increased portability across multiple hardware due to a common VM code representation.
- Separation of concerns and increased modularity.
- Reduced costs of development overall.

ARCHITECTURES OF VM

1. **Stack Machine Model** - JVM, CLR
2. **Register Machine Model** - Dalvik, Lua VM
3. **SSA Machine Model** - LLVM
4. **Accumulator Machine Model**
5. **Register-Memory Machine Model**
6. ...

THE STACK MODEL

THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

is translated to →

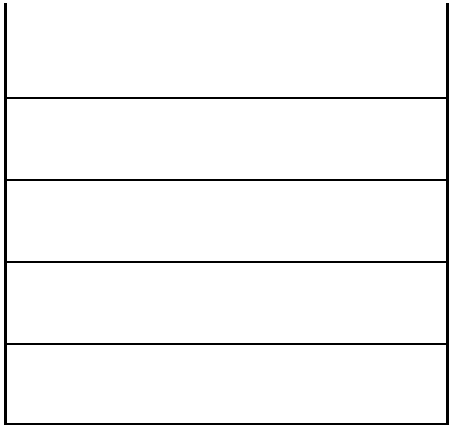
```
push 2
push x
sub
push y
push 5
add
mult
pop a
```

THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

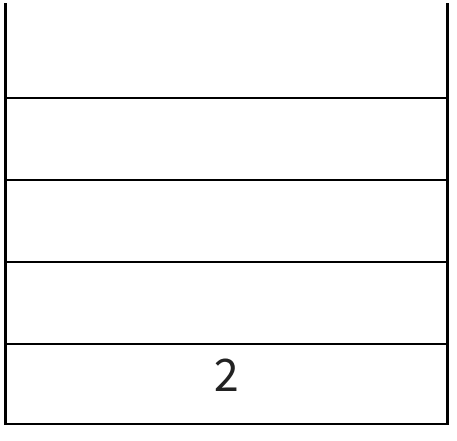


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

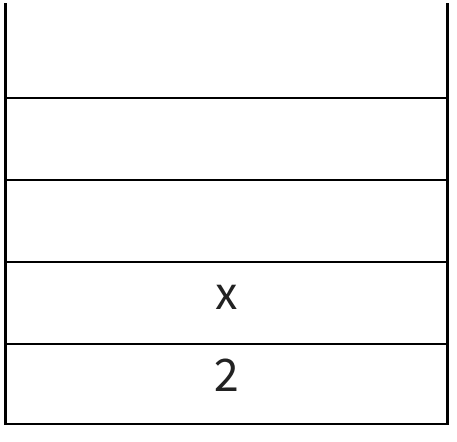


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

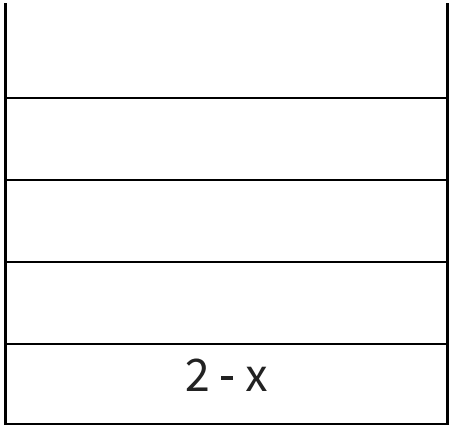


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

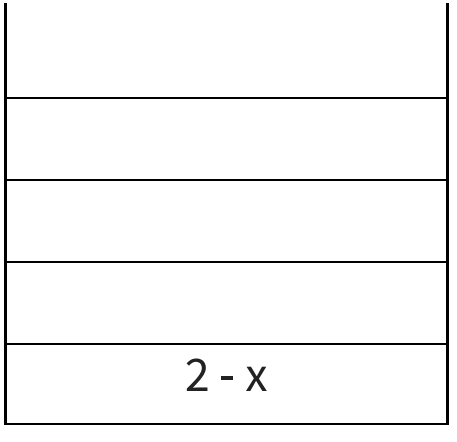


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

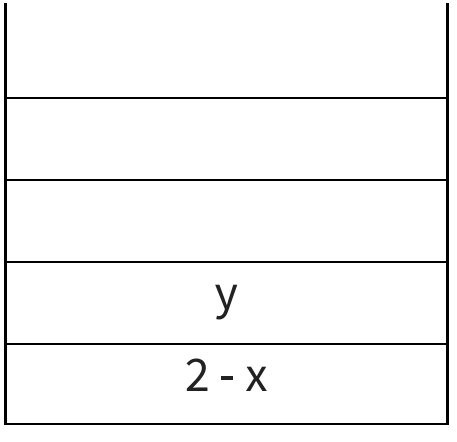


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

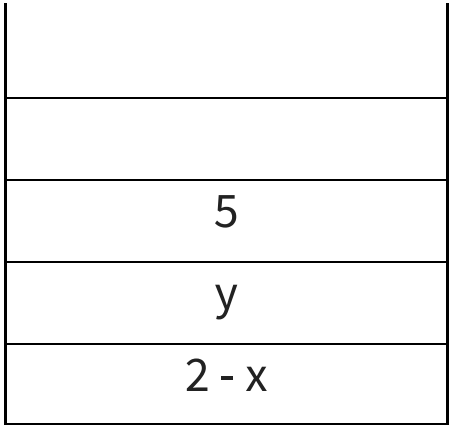


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

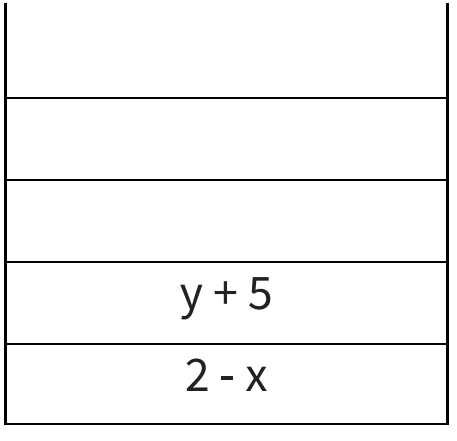


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

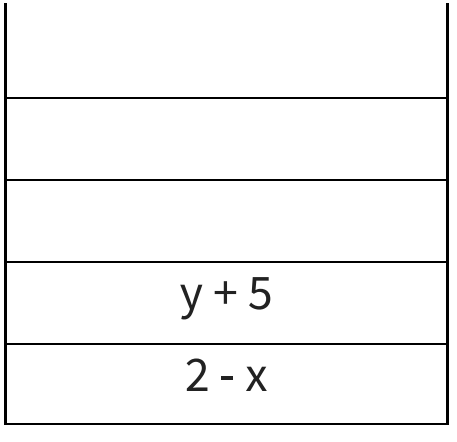


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

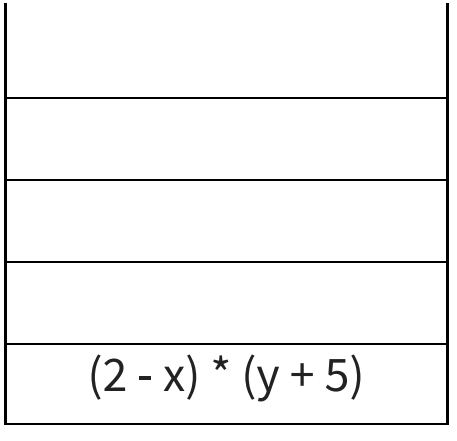


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```

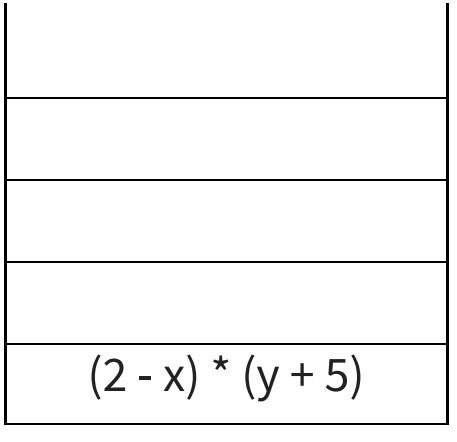


THE STACK MODEL

DESIGNING THE VM CODE:

$$a = (2 - x) \times (y + 5)$$

```
push 2
push x
  sub
push y
push 5
  add
 mult
 pop a
```



THE STACK MODEL

VIRTUAL MEMORY SEGMENTS:

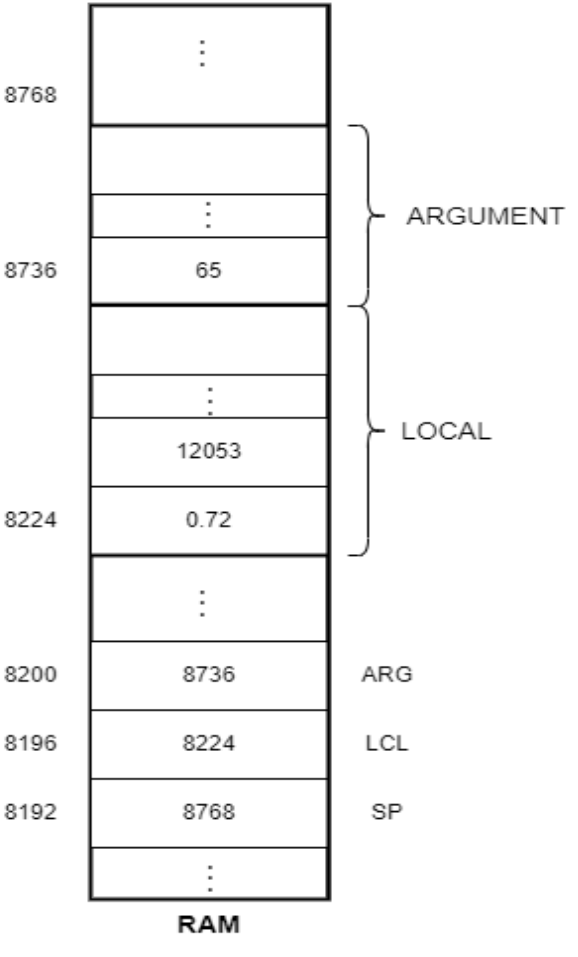
Segment	Purpose	Comments
`argument`	Stores the function's arguments.	Allocated dynamically by the VM implementation when the function is entered.
`local`	Stores the function's local variables.	Allocated dynamically by the VM implementation and initialized to 0's when the function is entered.
`static`	Stores static variables shared by all functions in the same `.vm` file.	Allocated by the VM imp. for each `.vm` file; shared by all functions in the `.vm` file.
`constant`	Pseudo-segment that holds all the constants in the range 0 . . . 32767.	Emulated by the VM implementation; Seen by all the functions in the program.
`this` `that`	General-purpose segments. Can be made to correspond to different areas in the heap. Serve various programming needs.	Any VM function can use these segments to manipulate selected areas on the heap.
`pointer`	A two-entry segment that holds the base addresses of the `this` and `that` segments.	Any VM function can set `pointer` 0 (or 1) to some address; this has the effect of aligning the `this` (or `that`) segment to the heap area beginning in that address.
`temp`	Fixed eight-entry segment that holds temporary variables for general use.	May be used by any VM function for any purpose. Shared by all functions in the program.

Name	*Usage*
`SP`	Stack pointer: points to the next topmost location in the stack;
`LCL`	Points to the base of the current VM function's `local` segment;
`ARG`	Points to the base of the current VM function's `argument` segment;
`THIS`	Points to the base of the current `this` segment (within the heap);
`THAT`	Points to the base of the current `that` segment (within the heap);
	Holds the contents of the `temp` segment;
	Can be used by the VM implementation as general-purpose registers.

THE STACK MODEL

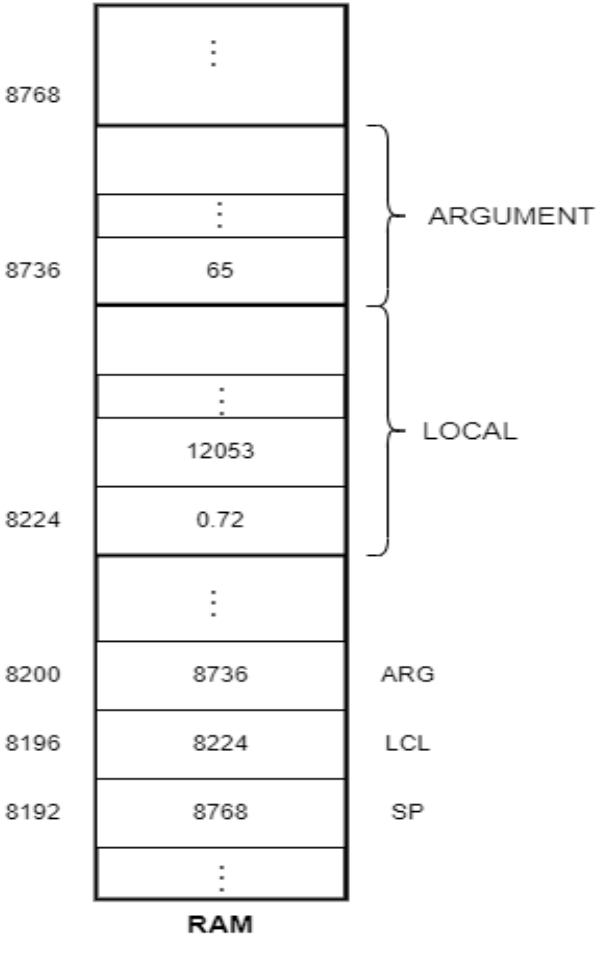
VIRTUAL MEMORY SEGMENTS:

Segment	Purpose	Comments
`argument`	Stores the function's arguments.	Allocated dynamically by the VM implementation when the function is entered.
`local`	Stores the function's local variables.	Allocated dynamically by the VM implementation and initialized to 0's when the function is entered.
`static`	Stores static variables shared by all functions in the same `.vm` file.	Allocated by the VM imp. for each `.vm` file; shared by all functions in the `.vm` file.
`constant`	Pseudo-segment that holds all the constants in the range 0 . . . 32767.	Emulated by the VM implementation; Seen by all the functions in the program.
`this` `that`	General-purpose segments. Can be made to correspond to different areas in the heap. Serve various programming needs.	Any VM function can use these segments to manipulate selected areas on the heap.
`pointer`	A two-entry segment that holds the base addresses of the `this` and `that` segments.	Any VM function can set `pointer` 0 (or 1) to some address; this has the effect of aligning the `this` (or `that`) segment to the heap area beginning in that address.
`temp`	Fixed eight-entry segment that holds temporary variables for general use.	May be used by any VM function for any purpose. Shared by all functions in the program.



THE STACK MODEL

EXAMPLE:



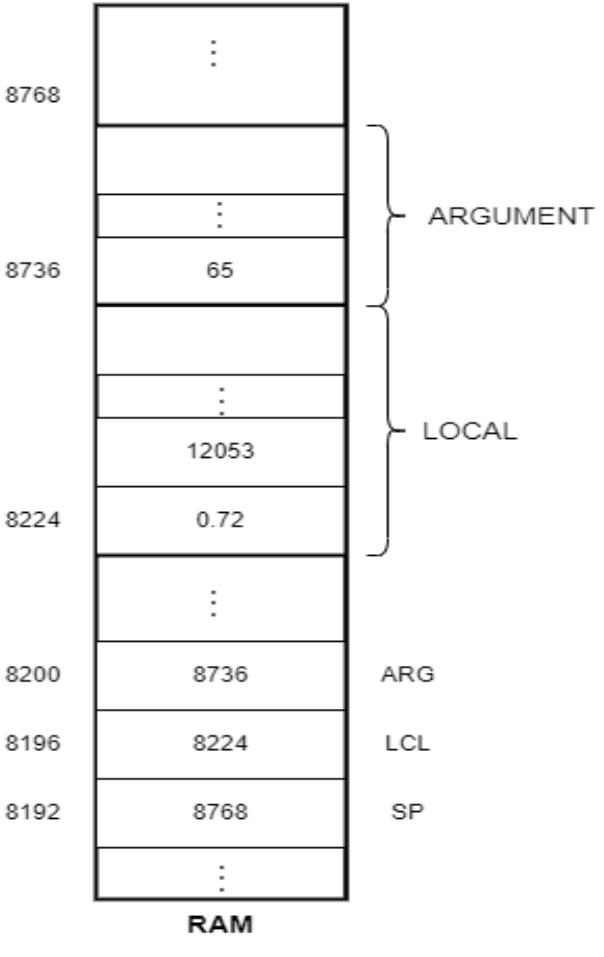

```
1 function main
2
3 push argument 0
4 push constant 42
5
6 add
7
8 pop local 1
```

.vmc to .asm →

```
 1 main:
 2 li x6, 8200   # x6=82
 3 lw x6, 0(x6)  # x6=87
 4 lw x6, 0(x6)  # x6=65
 5 sw x6, 0(x2)
 6 addi x2, x2, 4
 7
 8 li x7, 42
 9 sw x7, 0(x2)
10 addi x2, x2, 4
11
12 addi x2, x2, -4
13 lw x6, 0(x2)  # x6=42
14 addi x2, x2, -4
15 lw x7, 0(x2)  # x7=65
16 add x6, x6, x7
17 sw x6, 0(x2)
18 addi x2, x2, 4
19
```

THE STACK MODEL

EXAMPLE:



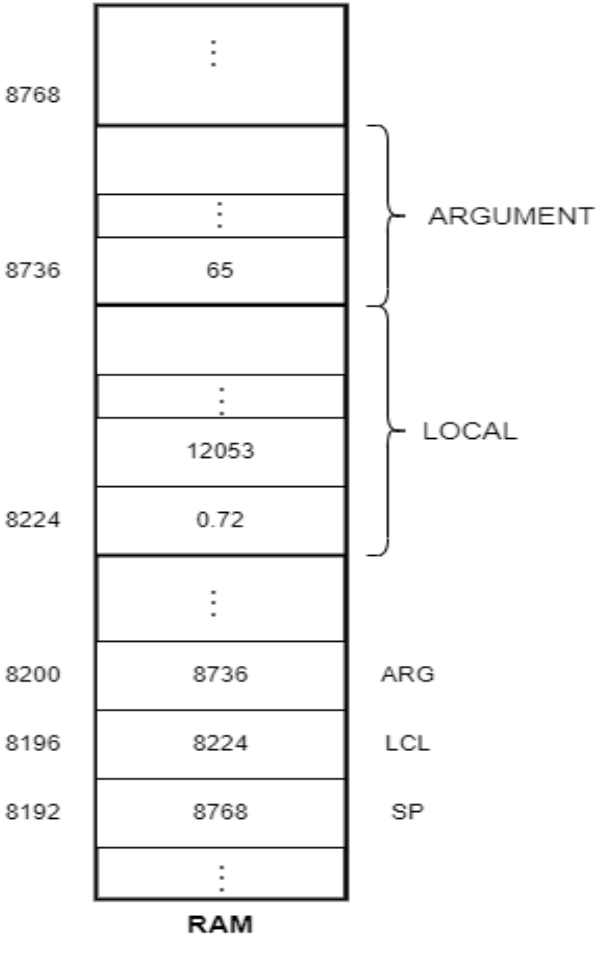

```
1  function main
2
3  push argument 0
4  push constant 42
5
6  add
7
8  pop local 1
```

.vmc to .asm →

```
 1  main:
 2  li x6, 8200    # x6=82
 3  lw x6, 0(x6)   # x6=87
 4  lw x6, 0(x6)   # x6=65
 5  sw x6, 0(x2)
 6  addi x2, x2, 4
 7
 8  li x7, 42
 9  sw x7, 0(x2)
10  addi x2, x2, 4
11
12  addi x2, x2, -4
13  lw x6, 0(x2)   # x6=42
14  addi x2, x2, -4
15  lw x7, 0(x2)   # x7=65
16  add x6, x6, x7
17  sw x6, 0(x2)
18  addi x2, x2, 4
19
```

THE STACK MODEL

EXAMPLE:



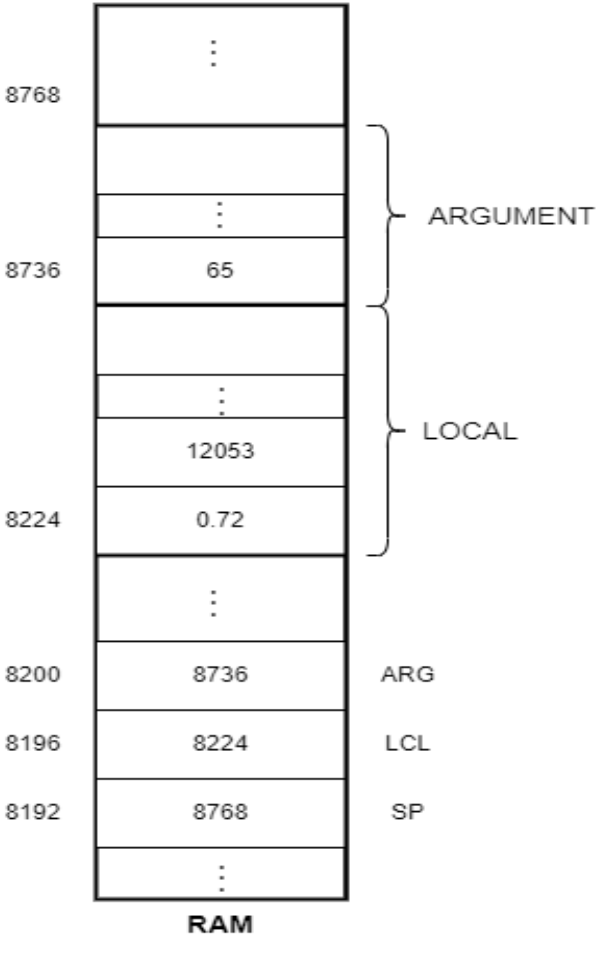

```
1  function main
2
3  push argument 0
4  push constant 42
5
6  add
7
8  pop local 1
```

.vmc to .asm →

```
 1  main:
 2  li x6, 8200    # x6=82
 3  lw x6, 0(x6)   # x6=87
 4  lw x6, 0(x6)   # x6=65
 5  sw x6, 0(x2)
 6  addi x2, x2, 4
 7
 8  li x7, 42
 9  sw x7, 0(x2)
10  addi x2, x2, 4
11
12  addi x2, x2, -4
13  lw x6, 0(x2)   # x6=42
14  addi x2, x2, -4
15  lw x7, 0(x2)   # x7=65
16  add x6, x6, x7
17  sw x6, 0(x2)
18  addi x2, x2, 4
19
```

THE STACK MODEL

EXAMPLE:



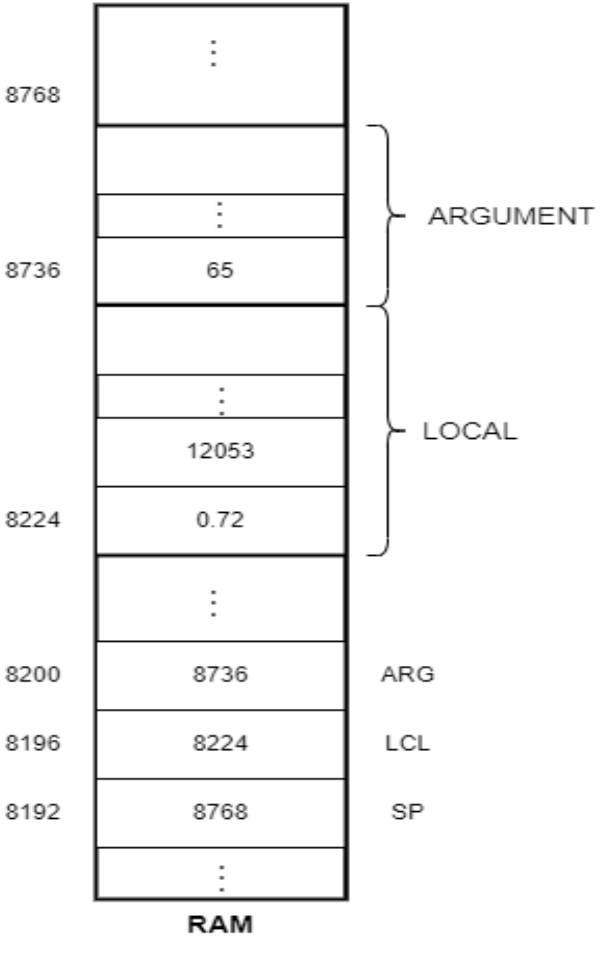

```
1  function main
2
3  push argument 0
4  push constant 42
5
6  add
7
8  pop local 1
```

.vmc to .asm →

```
 6  addi x2, x2, 4
 7
 8  li x7, 42
 9  sw x7, 0(x2)
10  addi x2, x2, 4
11
12  addi x2, x2, -4
13  lw x6, 0(x2)   # x6=42
14  addi x2, x2, -4
15  lw x7, 0(x2)   # x7=65
16  add x6, x6, x7
17  sw x6, 0(x2)
18  addi x2, x2, 4
19
20  addi x2, x2, -4
21  lw x6, 0(x2)   # x6=10
22  li x7, 8196
23  lw x7, 0(x7)   # x7=82
24  addi x7, x7, 4
```

THE STACK MODEL

EXAMPLE:



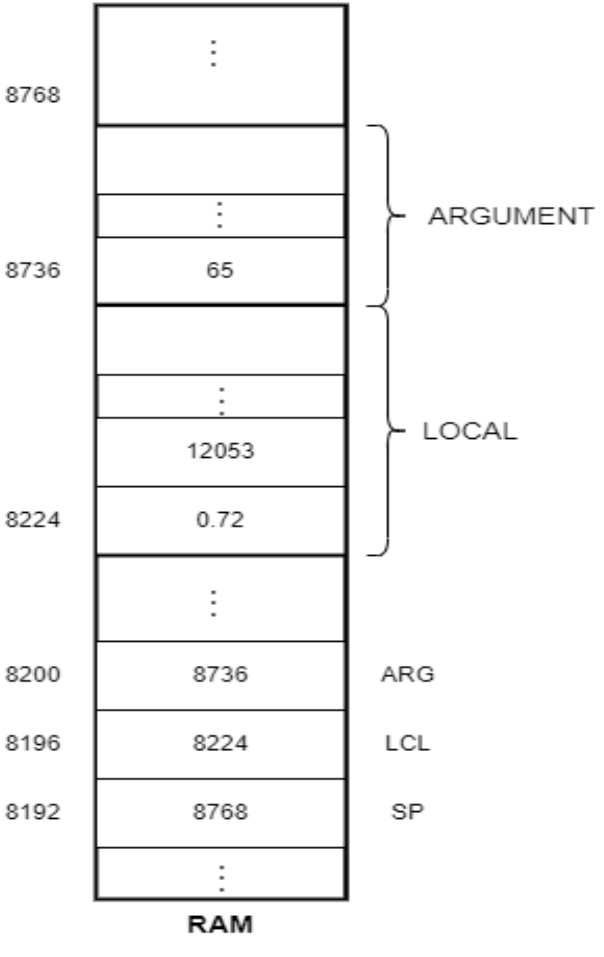

```
1  function main
2
3  push argument 0
4  push constant 42
5
6  add
7
8  pop local 1
```

.vmc to .asm →

```
 7
 8  li x7, 42
 9  sw x7, 0(x2)
10  addi x2, x2, 4
11
12  addi x2, x2, -4
13  lw x6, 0(x2)  # x6=42
14  addi x2, x2, -4
15  lw x7, 0(x2)  # x7=65
16  add x6, x6, x7
17  sw x6, 0(x2)
18  addi x2, x2, 4
19
20  addi x2, x2, -4
21  lw x6, 0(x2)  # x6=10
22  li x7, 8196
23  lw x7, 0(x7)  # x7=82
24  addi x7, x7, 4
25  sw x6, 0(x7)
```

THE STACK MODEL

EXAMPLE:




```
1  function main
2
3  push argument 0
4  push constant 42
5
6  add
7
8  pop local 1
```

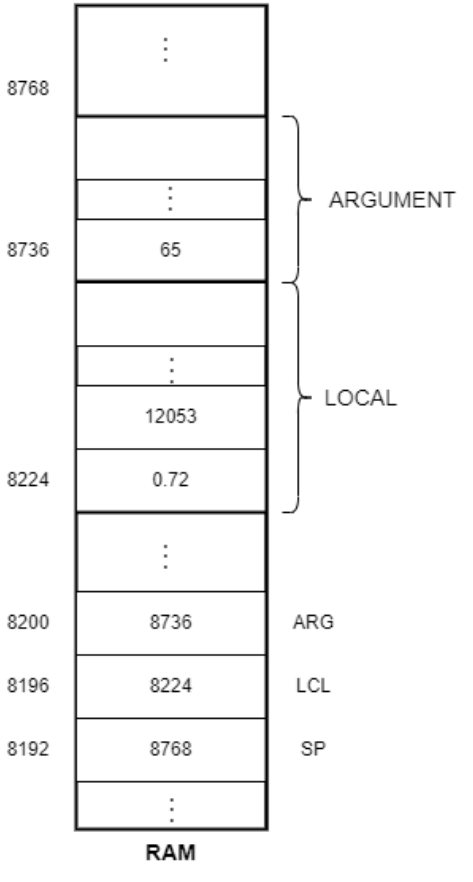
.vmc to .asm →

```
 1  main:
 2  li x6, 8200    # x6=82
 3  lw x6, 0(x6)   # x6=87
 4  lw x6, 0(x6)   # x6=65
 5  sw x6, 0(x2)
 6  addi x2, x2, 4
 7
 8  li x7, 42
 9  sw x7, 0(x2)
10  addi x2, x2, 4
11
12  addi x2, x2, -4
13  lw x6, 0(x2)   # x6=42
14  addi x2, x2, -4
15  lw x7, 0(x2)   # x7=65
16  add x6, x6, x7
17  sw x6, 0(x2)
18  addi x2, x2, 4
19
```

PROJECT TIPS AND EXPERIENCE

PROJECT TIPS AND EXPERIENCE

- Do enough research before finalizing idea and implementation.
- Establish clear communication guidelines between different teams as early as possible. For VM:
  - Compiler team for VM machine model and language
  - Assembler team for assembly language and additional custom operators, if any
  - Processor team for starting location of memory for VM segments, list of all operators supported, floating point support, size of memory, byte addressable/word addressable, etc.
- Maintain a documentation each by compiler, VM, assembly team and processor team. Communicate regularly.
- Complete your individual team's work in advance. Integrating into a single pipeline **will** reveal inconsistencies and errors.
- Optimize translated code in compiler and VM stages.
- Try some new features - OS scheduling, support for objects and heap.
- Enjoy the process. There is a lot to learn - implementing all the major CS concepts together.



PROJECT TIPS AND EXPERIENCE

RESOURCES:

1. The Elements of Computing Systems: Building a Modern Computer from First Principles by Noam Nisan, Shimon Schocken.
2. Building a modern computer from first principles playlists.

---

3. JVM and the tiered compilation process.
4. How to build a VM (Interpreted execution model) by T Parr.
5. RARS simulator for RISV-V.

---

6. Our github repository for VM.

THANK YOU

AND ALL THE BEST!