VM 8-Segment Translation Cheat Sheet

1,422 words · ~6 min read

---

*VCA-CSA-101 cross-chapter quick-reference handout. Anchor: §7.6 + Ch 8 §8.5-§8.7. *

Purpose: complete reference for the eight memory segments the Virtus VM exposes and the RV32I-Lite assembly each `push X i` / `pop X i` translates to. Print and pin during Lab 7.1 (stack arithmetic), Lab 7.2 (segment translator), Lab 7.3 (capstone end-to-end), Lab 8.1-8.4 (function-call protocol). Every memory access in any `.vm` source file the student writes lands on one of these eight segments; every emission pattern is one of the three translation strategies catalogued below.

At a glance

Property	Value
Total segments	**8** (`constant`, `argument`, `local`, `static`, `this`, `that`, `pointer`, `temp`)
Translation strategies	**3** (runtime base register / direct-address mapping / translation-time symbol)
Stack growth	**Upward** (Nand2Tetris convention; CSA-201 reconciles to real-RISC-V's downward)
`sp` register	`x2` per RV32I ABI; points at next-free slot
Primitive cost	Every `push`/`pop` ≈ 2 RV32I-Lite instructions
Typical bloat	~5-7 RV32I-Lite instructions per memory-touching VM command

---

The 8 segments. Overview table

Segment	Translation strategy	Lifetime	Per-instance?	Lab introduced
`constant`	No memory; immediate value	Compile-time	-	7.1
`argument`	Runtime base register (`ARG`)	Per-call	yes	7.2, 8.5
`local`	Runtime base register (`LCL`)	Per-call	yes	7.2, 8.5
`this`	Runtime base register (`THIS`)	Per-call object follow-through	yes	7.2
`that`	Runtime base register (`THAT`)	Per-call object follow-through	yes	7.2
`pointer`	Direct-address mapping → `THIS` / `THAT` registers	Per-program-shared	no	7.2
`temp`	Direct-address mapping → 8 fixed slots	Per-program-shared scratch	no	7.2
`static`	Translation-time symbol → linker-resolved `.data` slot	Per-source-file globals	no	7.2

The three lifetime classes drive the three translation strategies:

- **Per-call** (`argument`, `local`, `this`, `that`) → runtime base register held at fixed memory location; the *callee's* call-protocol prologue writes the per-call value into the base-register slot.
- **Per-program-shared** (`pointer`, `temp`) → fixed memory addresses; same for every call.
- **Per-source-file globals** (`static`) → translation-time symbol; the linker resolves to a `.data` slot the assembler reserved.

Memory map: where each segment lives at runtime

```
Virtus Console runtime memory layout (Ch 12 §12.2):

0x00000000 ─── OS .text (16 KiB)               ── (immutable; FPGA-init)
0x00004000 ─── OS .data + .bss (16 KiB)
       ├─ 0x00004400  Memory._free_head
       ├─ 0x00004??? Screen state / Keyboard state / Sound state
       └─ ...
0x00008000 ─── Application .text (32 KiB)      ── (immutable; FPGA-init)
0x00010000 ─── Application heap (16 KiB)       ── managed by Memory.alloc/dealloc
0x00010000 ─── ALSO the VM segment-base region:
       ├─ 0x00010000  LCL_addr     ← `local` segment base pointer
       ├─ 0x00010004  ARG_addr     ← `argument` segment base pointer
       ├─ 0x00010008  THIS_addr    ← `this` segment base pointer
       ├─ 0x0001000C  THAT_addr    ← `that` segment base pointer
       ├─ 0x00010010  temp[0]      ← fixed `temp` slot 0
       ├─ 0x00010014  temp[1]
       ├─ 0x00010018  temp[2]
       ├─ ...
       └─ 0x0001002C  temp[7]      ← fixed `temp` slot 7
0x00010030 ─── stack (grows upward)            ── sp initialised here
0x00014000 ─── application data heap continues
0x80000000 ─── HDMI framebuffer (peripheral)
0x80010000 ─── VCP shared memory (peripheral)
0x80020000 ─── GPIO/GamePad decoder (peripheral; DS2 protocol)
0x80030000 ─── system control (peripheral)
```

`static` storage lives in `.data` at addresses the linker resolves at link time; per-file naming convention `static.<basename>.<i>` keeps each file's statics distinct (see §static naming below).

Stack discipline

Every memory-touching VM command bottoms out on the stack, accessed via `sp` (= `x2`).

```
sp always points at the NEXT FREE SLOT (one past the topmost occupied word).

push:  M[sp] = value;  sp += 4
pop:   sp -= 4;        value = M[sp]
```

Push / pop primitives in RV32I-Lite (the building block for every emission pattern):

```
# push (value in t0 → top of stack)
sw   t0, 0(sp)
addi sp, sp, 4

# pop (top of stack → t0)
addi sp, sp, -4
lw   t0, 0(sp)
```

2 instructions per primitive. Every translation pattern below is `<address arithmetic> + <push or pop primitive>`.

Strategy 1: Runtime base register

Used by `argument`, `local`, `this`, `that`. **The base address is itself stored at a fixed memory location** (per the memory map above); the translator emits a load-then-add-offset-then-load/store pattern.

`push <segment> i`. Read from segment

```
# push &lt;segment&gt; i  (segment ∈ {local, argument, this, that})
lw   t1, 0(&lt;SEGMENT&gt;_addr)    # t1 = base address of segment
addi t1, t1, 4*i              # t1 = address of segment[i]
lw   t0, 0(t1)                # t0 = M[base + 4*i] = segment[i]
sw   t0, 0(sp)                # push t0
addi sp, sp, 4
```

5 instructions plus whatever the assembler needs to materialise `<SEGMENT>_addr` (typically one extra `lw` via `.data`-resident pointer indirection because RV32I-Lite has no `lui`).

`pop <segment> i`. Write to segment

```
# pop &lt;segment&gt; i  (segment ∈ {local, argument, this, that})
addi sp, sp, -4               # decrement sp first
lw   t0, 0(sp)                # t0 = top-of-stack value
lw   t1, 0(&lt;SEGMENT&gt;_addr)    # t1 = base address
addi t1, t1, 4*i              # t1 = address of segment[i]
sw   t0, 0(t1)                # M[base + 4*i] = t0
```

5 instructions. The ordering (pop first, then compute address) matters: it uses 2 temp registers (`t0` for value, `t1` for address) instead of 3, which fits the chapter's two-temp convention.

Mapping table

VM segment	Base register location	Set by
`local`	`LCL_addr` at `0x00010000`	Caller's `call` protocol (Ch 8 §8.6)
`argument`	`ARG_addr` at `0x00010004`	Caller's `call` protocol
`this`	`THIS_addr` at `0x00010008`	Method prologue (Ch 8 §8.5) or `pop pointer 0`
`that`	`THAT_addr` at `0x0001000C`	`pop pointer 1` from VM source

Strategy 2: Direct-address mapping

Used by `pointer` and `temp`. **The segment's address range is fixed at translation time**; the translator emits a direct-address load.

`pointer` segment. Exposes THIS/THAT registers themselves

The `pointer` segment is a **two-slot segment** that aliases the THIS and THAT registers:

- `pointer 0` ↔ `THIS`
- `pointer 1` ↔ `THAT`

```
# push pointer 0
lw   t0, 0(THIS_addr)         # t0 = current THIS
sw   t0, 0(sp)
addi sp, sp, 4

# pop pointer 0  (sets THIS register)
addi sp, sp, -4
lw   t0, 0(sp)
sw   t0, 0(THIS_addr)
```

3 instructions per **`pop pointer 0`**, the simplest of all segments because no base+offset arithmetic.

`pop pointer 0` is the canonical idiom for *establishing* *`this`* in a method body:

```
 push argument 0            # push the receiver (caller's `this`)
pop  pointer 0             # set THIS to the receiver
```

Indices outside `0..1` are illegal and must be diagnosed by the translator.

`temp` segment, 8 fixed scratch slots

The `temp` segment is an **eight-slot scratch region** at fixed addresses:

- `temp 0` → `0x00010010`
- `temp 1` → `0x00010014`
- …
- `temp 7` → `0x0001002C`

```
# push temp i
lw   t0, (0x00010010 + 4*i)(x0)   # absolute address; assembler synthesises if needed
sw   t0, 0(sp)
addi sp, sp, 4

# pop temp i
addi sp, sp, -4
lw   t0, 0(sp)
sw   t0, (0x00010010 + 4*i)(x0)
```

3 instructions for either direction. No indirection.

Use cases: compiler-emitted intermediate values that need to outlive a single VM expression but do not deserve a `local` slot. Indices outside `0..7` are illegal.

Strategy 3: Translation-time symbol (`static`)

The `static` segment holds **per-source-file globals**. The translator emits a symbolic reference; the linker resolves it to a `.data` slot at link time.

Per-file naming convention (central)

> A reference `static i` in source file `<basename>.vm` translates to the assembly symbol `static.<basename>.<i>`.

Source	Assembly symbol
`Foo.vm`'s `static 0`	`static.Foo.0`
`Foo.vm`'s `static 1`	`static.Foo.1`
`Bar.vm`'s `static 0`	`static.Bar.0`
`Bar.vm`'s `static 1`	`static.Bar.1`

`Foo.Bar`'s `static 0` and `Foo.Baz`'s `static 0` are *different* memory cells, the same way C `static` globals are file-scoped. Two `.vm` files declaring `static 0` do **not** alias.

Translation pattern

```
# push static i  (in file Foo.vm)
la   t1, static.Foo.<i>       # load address of the static slot
lw   t0, 0(t1)                # t0 = M[static.Foo.<i>]
sw   t0, 0(sp)
addi sp, sp, 4

# pop static i
addi sp, sp, -4
lw   t0, 0(sp)
la   t1, static.Foo.<i>
sw   t0, 0(t1)
```

`la` (load-address) expands to a `.data`-resident pointer indirection (RV32I-Lite has no `lui`); the linker resolves the address.

`.data` reservation (emitted at top of generated `.s` file)

The translator scans the source for `max(static_index)` and reserves `max_index + 1` slots:

```
.section .data
.global static.Foo.0
.global static.Foo.1
.global static.Foo.2
static.Foo.0: .word 0
static.Foo.1: .word 0
static.Foo.2: .word 0
.section .text
```

Lab 7.2's harness checks that the reservation count matches the maximum static-index used.

`constant`. Not memory, just a literal

`constant` is the **simplest segment** and the entry point for every numeric literal in source.

```
# push constant n  (n in [-2048, +2047] - 12-bit signed immediate)
addi t0, x0, n
sw   t0, 0(sp)
addi sp, sp, 4

# push constant n  (n outside [-2048, +2047])
li   t0, n                   # assembler chooses expansion (.data indirection on RV32I-Lite)
sw   t0, 0(sp)
addi sp, sp, 4
```

3 instructions for in-range; 4 (or more) for out-of-range depending on `li` expansion.

`pop constant n` **is illegal** and must be diagnosed by the translator. *`constant` is read-only by definition. There is no memory cell named `constant[N]` to write to.*

Quick-emit table, every push/pop pattern at a glance

Command	Instructions	Notes
`push constant n`	3 (small) / ~4 (large)	Immediate; no memory access for small `n`
`push local i`	5	Runtime base; `LCL_addr`
`pop local i`	5	Runtime base; `LCL_addr`
`push argument i`	5	Runtime base; `ARG_addr`
`pop argument i`	5	Runtime base; `ARG_addr`
`push this i`	5	Runtime base; `THIS_addr`
`pop this i`	5	Runtime base; `THIS_addr`
`push that i`	5	Runtime base; `THAT_addr`
`pop that i`	5	Runtime base; `THAT_addr`
`push pointer {0,1}`	3	Direct address; aliases THIS/THAT
`pop pointer {0,1}`	3	Direct address; aliases THIS/THAT
`push temp i` (`i ∈ 0..7`)	3	Direct address; fixed `0x00010010 + 4i`
`pop temp i`	3	Same
`push static i`	5	Symbol-resolved at link time; `.data` slot
`pop static i`	5	Same

Worked translation example (§7.8 from prose)

VM source. Three lines:

```
push constant 7
push local 2
add
```

Translates to **17 RV32I-Lite instructions**:

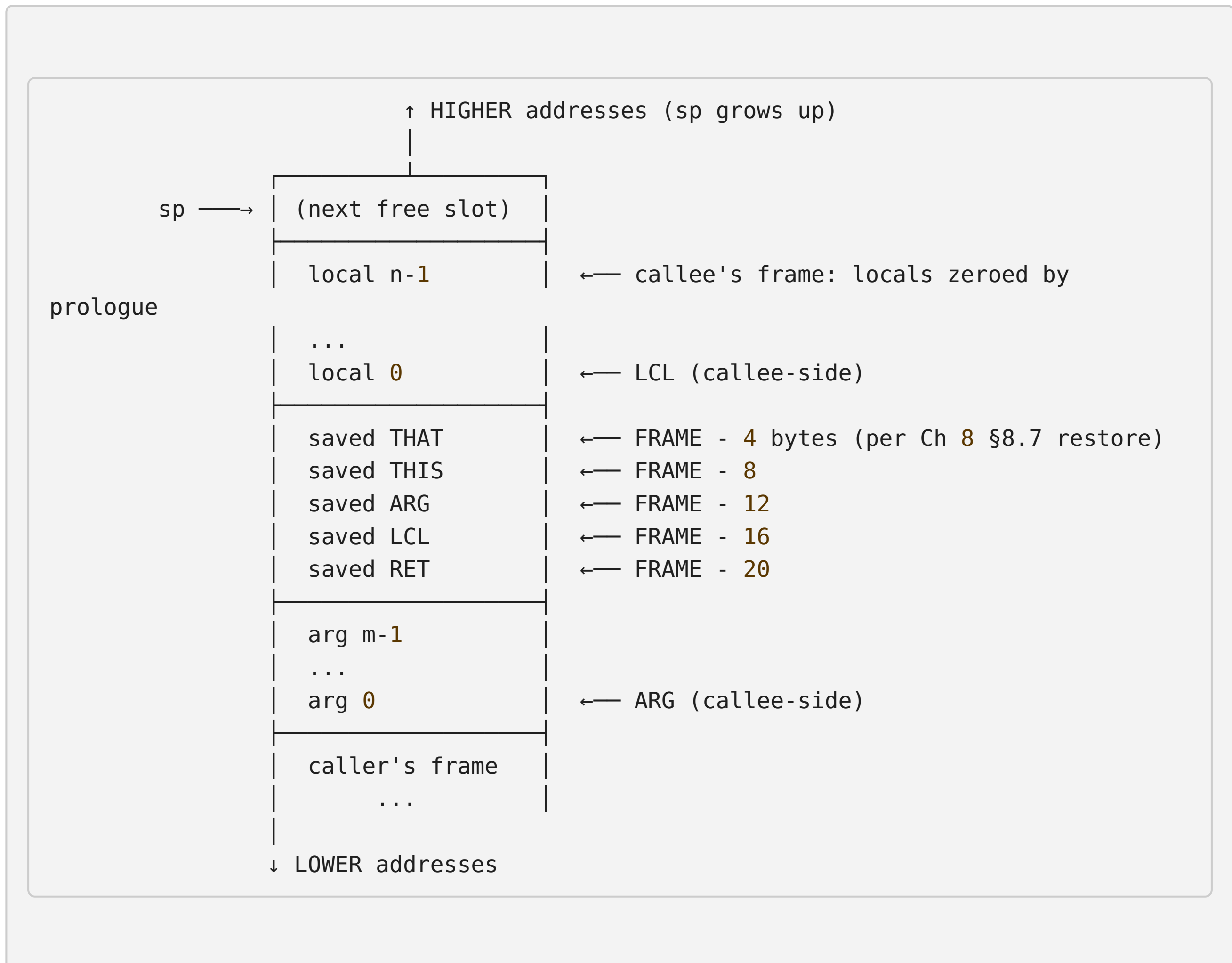
```
# push constant 7
addi t0, x0, 7             ; 1
sw   t0, 0(sp)             ; 2
addi sp, sp, 4             ; 3

# push local 2
lw   t1, 0(LCL_addr)       ; 4   (plus assembler indirection - count separately)
addi t1, t1, 8             ; 5   (offset = 4 * 2 = 8 bytes)
lw   t0, 0(t1)             ; 6
sw   t0, 0(sp)             ; 7
addi sp, sp, 4             ; 8

# add
addi sp, sp, -4            ; 9   (pop b)
lw   t0, 0(sp)             ; 10
addi sp, sp, -4            ; 11  (pop a)
lw   t1, 0(sp)             ; 12
add  t0, t1, t0            ; 13
sw   t0, 0(sp)             ; 14  (push a+b)
addi sp, sp, 4             ; 15
```

~17 instructions for 3 VM commands. Hand-written equivalent (`local[2] + 7`) is ~5 instructions plus address materialisation. **The 12-instruction overhead is the cost of the stack-machine abstraction** (Ch 11 §11.9 quantifies the cost across a 50-line program at 64×; CSA-201's optimisation track recovers most of it via register allocation).

Calling-convention diagram (Ch 8 §8.6 + §8.7)

> *(Added 2026-04-29 per audit . Cross-references Ch 8 §8.6.2 (register convention) + Ch 6a §6a.5.4-§6a.5.5 (linker prologue + memory layout).)*

The Virtus VM call protocol pushes **5 saved-state words** per call onto the stack: `RET` (return-address label) + caller's `LCL` + caller's `ARG` + caller's `THIS` + caller's `THAT`. Plus arguments. Plus locals (zeroed by callee prologue). The shape on the stack at the moment the callee starts executing:



`FRAME = LCL` is what `Ch 8 §8.7` step 1 saves into `t6` before any restore writes back to `LCL_addr`, without saving FRAME first, the saved-state-region offsets walk into the wrong memory.

Caller's emission (`call f m` per Ch 8 §8.6)

```
1. push return-address label   ← `la t0, Caller$ret.N; sw t0, 0(sp); addi sp, sp,
4`
2. push current LCL            ← `lw t0, LCL_addr; sw t0, 0(sp); addi sp, sp, 4`
3. push current ARG            ← same shape
4. push current THIS           ← same shape
5. push current THAT           ← same shape
6. ARG = SP - (5+m)*4          ← `addi t0, sp, -(5+m)*4; sw t0, ARG_addr`
7. LCL = SP                    ← `sw sp, LCL_addr`
8. transfer control            ← `la t0, f; jalr x0, t0, 0`
9. <return label>              ← Caller$ret.N: (callee returns here)
```

Register convention applied: `t0` is caller-clobbered (used freely as scratch); `sp` and `gp` are preserved (callee won't touch); arguments go on the stack (not in registers). Per cross-chapter-rv32i-lite-encoding-card.md "Register convention" section.

Callee's emission (`function f n` prologue + `return` epilogue per Ch 8 §8.5 + §8.7)

```
function f n:                   ← entry label
  for i in 0..n-1:              ← zero each local slot
    sw x0, 0(sp); addi sp, sp, 4

return:                         ← epilogue (9 numbered steps; ordering matters)
  1. FRAME = LCL                 ← `lw t6, LCL_addr` - save before any restore
writes
  2. RET = M[FRAME-20]           ← saved return address (before frame teardown)
  3. M[ARG] = pop                ← place return value at caller-visible slot
  4. SP = ARG + 4                ← caller's stack tops up at the return-value slot
  5-8. restore THAT/THIS/ARG/LCL from FRAME-4/-8/-12/-16
  9. jalr x0, RET, 0             ← jump back to caller's return label
```

Ordering constraint: step 3 + step 4 use the *callee-side* ARG; steps 5-8 restore caller-side values. Step 1 must run before any of 5-8 (which overwrite `LCL_addr`). This is the most-central student-trap in Lab 8.2.

Cross-cuts to Ch 6a's runtime-image partition

- **`la t0, Caller$ret.N`** lowers (per Ch 6a §6a.4.6) to `lw t0, gp_offset(gp)` against a la-ptr-table slot at `data_mem[gp+0x40..0x3FF]` that the linker prologue (Ch 6a §6a.5.4) populated at boot. The `t0`-loaded value is the absolute address of `Caller$ret.N` in instr_mem (`0x1200`-region per cross-chapter-instr-mem-layout.md).
- **`la t0, f`** for cross-section calls lowers the same way; the la-ptr-table slot's value is `f`'s resolved address.
- **`gp` itself** is initialized by the synth-time bootstrap at instr_mem `0x000`-`0x01F` to point at `0x00010000` (the segment-pointer-region base; per existing memory map above). The bootstrap zeroes `gp+0..0x10` (LCL/ARG/THIS/THAT slots); the linker prologue at instr_mem `0x200`-`0x11FF` populates `gp+0x40..0x3FF`.

Lifetime + segment-base setup table

When does each segment's base get set? Who sets it?

Segment	Base address held at	Set by	When
`local`	`LCL_addr` (`0x00010000`)	Caller's `call` protocol; sets `LCL = SP` after pushing saved-state	Ch 8 §8.6 emits `sw sp, LCL_addr` after the 5 caller-pushes
`argument`	`ARG_addr` (`0x00010004`)	Caller's `call` protocol; sets `ARG = SP - (5+m)*4`	Ch 8 §8.6
`this`	`THIS_addr` (`0x00010008`)	(a) Method prologue: `push argument 0; pop pointer 0`. (b) Constructor prologue: `push constant N; call Memory.alloc 1; pop pointer 0`. (c) Source-level `pop pointer 0`.	Ch 8 §8.5 (method/constructor); Ch 11 §11.5 (compiler emission)
`that`	`THAT_addr` (`0x0001000C`)	Source-level `pop pointer 1` only	Compiler emits during array-base setup
`pointer`	(aliases THIS/THAT)	Source-level `pop pointer i`	(no separate setup)
`temp`	(fixed addresses)	(no setup needed)	(always available)
`static`	(linker-resolved)	(linker fills `.data` at link time; initial values from `.word 0` directives)	Ch 6a

---

Lab grading hooks

Lab 7.2's harness exercises the eight-segment translation explicitly:

Test class	What it checks
`push constant N` for N in `[-2048, 0, 2047]` and outside	Immediate range + `li` fallback
`push local i; pop local i` round-trip	Runtime base register read/write
`pop pointer 0` then `push this i`	THIS register update via pointer-segment alias
`push temp 7; pop temp 7`	Fixed-address segment ends
`push temp 8`	Diagnoses out-of-range; **must reject**
`pop constant N`	Diagnoses illegal; **must reject**
`push pointer 2`	Diagnoses out-of-range pointer; **must reject**
`Foo.vm static 0` vs `Bar.vm static 0`	Per-file static-naming yields distinct symbols at link time
Negative indices on any segment	Diagnoses illegal; **must reject**

Where to read more

- **Ch 7** *VM I*. Full segment overview; §7.6 (the conceptual heart) + §7.6.1-§7.6.5 (per-segment translation patterns) + §7.7 (`static` per-file naming).
- **Ch 8** *VM II*. Function-call protocol; §8.5 (`function f n` prologue), §8.6 (`call f m` caller-side), §8.7 (`return` callee-side restore), these are what set / save / restore the runtime base registers.
- **Ch 6a** *Static Linker*. `R_VIRTUS_32` resolution that lets `la t1, static.Foo.<i>` work across files.
- **Ch 11 §11.9**. Quantitative bloat reckoning; the 64× source-to-RV32I-Lite expansion that the per-segment 5-instruction emissions accumulate to.
- **Ch 12 §12.2**, Virtus Console memory map (the absolute addresses the segment-base table cites).
- **Findings §7.1** + **§16**, the canonical RV32I-Lite + 8-segment specification.

© Virtus Cyber Academy. Generated 2026-05-08.