Stack-based JOI Virtual Machine Specification

Version 2.0

Table of Contents

1. Memory Model

1.1 Overview

The JOI VM implements a *hybrid memory model* with stack-based and segment-based addressing. The memory space is divided into virtual segments that map to three physical segments: **Stack**, **Heap**, and **Pointer**.

1.2 Physical Memory Layout

Segment Type	Segment Name	Address Range	Size (bytes)
STACK	Local	8224 - 8735	512
STACK	Argument	8736 - 8767	32

STACK	Temp	8768 - 9279	512
STACK	ALU Stack	9280 - 10303	1024
HEAP	Heap	10304 - 11327	1024
POINTER	Pointer	11328-11839	512

---

2. Virtual Segments

2.1 Virtual Segment Types

- **Constant**: *Pseudo-segment for immediate values (not physically mapped)*
- **Local**: Function local variables storage
- **Argument**: Function parameter storage
- **Temp**: Compiler-generated temporary variables
- **Heap**: Dynamic memory allocation space
- **Static**: Static variable storage
- **Pointer**: Memory allocation metadata

2.2 Segment Mapping

Virtual segments map to physical memory as follows:

- **Local** → Stack
- **Argument** → Stack
- **Temp** → Stack
- **Static** → Heap
- **Pointer** → Pointer

---

3. Instruction Set Architecture

3.1 Stack Operations

- push <segment> <index> <type>: *Push value from segment to stack*

- `pop <segment> <index> <type>`: *Pop value from stack to segment*
- `add <type>`: *Add top two stack values*

3.2 Function Operations

- `function <name> <n_args> <return_type>`: *Function declaration*
- `call <name> <n_args>`: *Function call*
- `return [type]`: *Return from function*

3.3 Control Flow

- `label <label_name>`: *Define label*
- `goto <label_name>`: *Unconditional jump*
- `if-goto <label_name>`: *Conditional jump*

3.4 Libraries

- `lib <library name.jvm>`: *Include library*

3.5 Memory Access(Not fixed, could be changed Later)

- `alloc <count> <datatype>`: *Allocate heap memory for count number of variables of the mentioned datatype and return the triplet of (base address, count, datatype) in the stack*
- `delete`: *Free heap memory*

---

4. Type System

4.1 Supported Types

- **INT**: 32-bit signed integer
- **FLOAT**: 32-bit floating-point
- **BOOL**: Boolean value(Not done yet)
- **STRING:** (Not Tested yet)
- **PTR**: Triplet of INTs of the form BCD(base address, count, datatype).

4.2 Type Rules

- All stack operations must specify types.
- Type checking occurs at instruction execution.
- Implicit type conversion is not supported.

---

5. Function Calling Convention

5.1 Call Frame Layout

The call frame is organized as follows (from bottom to top):

1. Return address (-4 from LCL)
2. Previous ARG pointer (-16 from LCL)
3. Previous LCL value (-20 from LCL)
4. Previous TMP value (-12 from LCL)
5. Previous HEAP pointer (-8 from LCL)
6. Local variables (0 to n_locals * 4 from LCL)
7. Temporary variables (after locals)
8. Working stack (grows upward)

5.2 Function Call Sequence

1. Store the current ARG pointer in a temporary register.
2. Calculate and set a new ARG pointer as `(SP - (n_args + 1) * 4)`.
3. Push the calling context in the following order:
    - Previous LCL value
    - Previous ARG pointer
    - Previous TMP pointer
    - Previous HEAP pointer
4. Jump to the function label (storing the return address).

5.3 Function Entry Sequence

1. Store the return address.
2. Set a new LCL pointer to the current stack pointer.
3. Set a new TMP pointer `(LCL + n_locals * 4)`.
4. Allocate space for locals and temps.
5. Initialize the working stack pointer.

5.4 Function Return Sequence

1. Store the return value at `ARG[0]`.
2. Restore frame:
   - Restore HEAP pointer
   - Restore TMP pointer
   - Restore ARG pointer
   - Restore LCL pointer
3. Restore stack pointer to `ARG + 4`.
4. Jump to the return address.

5.5 Special Cases

- The **`joi` function** (entry point) has a simplified initialization and returns via the special `__END__` label.
- All functions must have valid return statements.
- Function validity is tracked during compilation.

---

6. Instruction Execution Model

6.1 Register Usage

- **x1**: Return address register
- **x2**: Stack pointer
- **x5, x6**: Temporary computation registers
- **x7**: ARG pointer preservation
- **x28**: Return jump register
- **x30**: Program termination

6.2 Memory Segment Management

Segment Pointers:

- **LCL**: Local variable base pointer
- **ARG**: Function arguments base pointer
- **TMP**: Temporary variable area pointer
- **HEAP**: Heap memory pointer

6.3 Stack Operations

- Stack grows upward (addresses increase).

- **Push**: Store value and increment SP by 4.
- **Pop**: Decrement SP by 4 and load value.
- Word-aligned operations (4-byte boundaries).

6.4 Function Context Preservation

1. **Context Save**:
   - Store current segment pointers (`LCL`, `ARG`, `TMP`, `HEAP`).
   - Preserve working registers.
   - Calculate new base pointers.
2. **Context Restore**:
   - Reload all segment pointers.
   - Restore stack frame.
   - Reset working stack.

6.5 Error Checking

- Function definition verification.
- Return statement validation.
- Link-time function resolution.
- Stack frame boundary validation.

6.6 Example Instruction Sequence

RISC-V Assembly

```
Unset
# Function Call
li x5, ARG_BASE           # Load ARG base address
lw x7, 0(x5)             # Store current ARG pointer
addi x5, x2, -12         # Calculate new ARG position
li x6, ARG_BASE          # Load ARG pointer location
sw x5, 0(x6)             # Set new ARG pointer
# [Push context...]
jal x1, function_name    # Jump to function

# Function Return
addi x2, x2, -4          # Adjust stack for return value
lw x5, 0(x2)             # Load return value
li x6, ARG_BASE          # Load ARG location
lw x6, 0(x6)             # Get ARG pointer
sw x5, 0(x6)             # Store return value at ARG[0]
```

```
# [Restore context...]
jalr x28, x5, 0         # Return to caller
```

---

7. Error Handling

7.1 Runtime Errors

- Stack overflow/underflow
- Type mismatch
- Invalid memory access
- Undefined function/label

7.2 Error Recovery

- Immediate execution halt
- Error code propagation
- Stack trace generation

---

8. Implementation Notes

8.1 Memory Management

- Stack frames are of fixed size.
- Heap allocation uses a *first-fit* algorithm.(TBD)
- Pointer segment tracks heap allocations.

8.2 Optimization Guidelines

- Constant folding is permitted.
- Dead code elimination allowed.
- Register allocation is recommended.

---

9. Linking Model

9.1 Overview

The JOI VM implements a *two-phase linking system*:

- **Library Linking**: Pre-processing of library dependencies
- **User Program Linking**: Resolution of function implementations across multiple source files

9.2 Library Linking

9.2.1 Library Declaration

- `lib <library_name>.jvm`

9.2.2 Library Resolution Process

- Libraries are resolved during initial preprocessing.
- Library paths are resolved relative to the `libraries/` directory.
- Library code is inserted at the point of declaration.
- Library declarations are stripped from the final code.

9.2.3 Library Loading Sequence

1. Read library file path
2. Load library content
3. Preprocess library code
4. Prepend processed library code to the main program
5. Remove library declaration line

9.3 User Program Linking

9.3.1 Function Resolution Rules

Functions can be defined in:

- Main program file
- Helper files
- Library files

Resolution Priority:

1. Main program definitions take precedence.
2. Helper file implementations are used for undefined functions.
3. Library implementations are used as fallback.

9.3.2 Linking Process

- **First Pass**:
  - Collect all function definitions from the main program.
  - Build a function table with main program implementations.
  - Mark functions as defined or undefined.
- **Second Pass**:
  - Process helper files.
  - Collect function implementations.
  - Store in helper function table.
- **Resolution Pass**:
  - For each undefined function in the main program:
    - Search helper function table.
    - Insert implementation if found.
    - Maintain the original function declaration position.
- **Validation Pass**:
  - Verify all functions have implementations.
  - Check for return statements.
  - Validate function signatures.

9.4 Error Handling

9.4.1 Library Errors

- Missing library file(done)
- Circular library dependencies(Ignored for now)
- Invalid library format(Compiler will take care)

9.4.2 Linking Errors

Linking error types:

1. **Undefined function**: "Function `{func_name}` is not defined"]
2. **Missing return**: "Function `{func_name}` lacks a valid return statement"
3. **Multiple definitions**: "Function `{func_name}` is defined multiple times"
4. **Signature mismatch**: "Function `{func_name}` implementation doesn't match declaration"

(For now, 1,2 and 3 have been implemented)

9.5 Example Linking Scenario

9.5.1 Source Files

```
Unset
main.jvm
lib math.jvm
function add 2 INT
function joi
    [joi implementation]
    return

helper.jvm
function add 2 INT
    push argument 0 INT
    push argument 1 INT
    add INT
    return INT
```

9.5.2 Linking Result

Processed Output

```
Unset
[math.jvm contents]

function add 2 INT
    push argument 0 INT
    push argument 1 INT
    add INT
    return INT

function joi
    [joi implementation]
    return
```

9.6 Implementation Details

9.6.1 Helper Function Processing

```
helper_functions = {
    'function_name': 'implementation_code'
}
```

9.6.2 Main Function Tracking

```
main_functions = {
    'function_name': bool  # True if has return statement
}
```

9.6.3 Code Generation

- **Process main file line by line**
  - For function declarations:
    - Check if the function exists in `helper_functions`.
    - If exists and is not in `main_functions`, insert helper implementation.
    - Otherwise, keep the original declaration.
- **Maintain line ordering for non-function code**