VM-specific v1.3.0 opcodes simulation (verbatim)

NOTES:

- changed META - it can be used for MSIZE simulation
- setting ergs per pubdata is done by separate opcode now (not part of `near_call`)
- incrementing TX counter is done by separate opcode now (not part of `far_call`)

Our VM has some opcodes that are not expressible in Solidity, but we can simulate them on the Yul compiler level by using “verbatim_*” instruction.

For some simulations below we assume that there exist a hidden global pseudo-variable called `ACTIVE_PTR` for manipulations, since one can not easily load pointer value into Solidity’s variable.

Simulated opcode	Verbatim signature	Arg 1	Arg 2	Arg 3	Arg 4	Arg 5	Arg 6	Arg 7	Return value	LLVM implementation
`to_l1(is_first, in0, in1`	`verbatim_3i_0o("to_l1", ...)`	if_first (bool)	in0 (u256)	in1 (u256)					_	`@llvm.syncvm.tol1(i256 %in0, i256 %in1, i256 %is_first)`
`code_source`	`verbatim_0i_1o("code_source", ...)`								address	`@llvm.syncvm.context(i256 %param) ; param == 2` (see SyncVM.h)
`precompile(in0, ergs_to_burn, out0)`	`verbatim_2i_1o("precompile", ...)`	in0 (u256)	ergs_to_burn (u32)						out0	`@llvm.syncvm.precompile(i256 %in0, i256 %ergs)`
`meta`	`verbatim_0i_1o("meta", ...)`								u256 (tight packing of ``` pub struct VmMetaParameters { pub block_number: u64, pub block_timestamp: u64, pub ergs_per_pubdata_byte: u32, pub ergs_per_pubdata_byte_limit_in_block: u32, pub this_shard_id: u8, pub caller_shard_id: u8, pub code_shard_id: u8, } ```	`@llvm.syncvm.context(i256 %param) ; param == 3` (see SyncVM.h)
`mimic_call(to, abi_data, implicit r3 = who to mimic)`	`verbatim_3i_1o("mimic_call", ...)`	who_to_call	who_to_mimic	abi_data					It is a call, so it WILL mess up the registers and WILL use `r1-r4` for our standard ABI convention and `r5` for the extra `who_to_mimic` argument.	Runtime `*{i256, i1} __mimiccall(i256, i256, i256, *{i256, i1})`
`system_mimic_call(to, abi_data, implicit r3, r4, r5 = who to mimic)`	`verbatim_7i_1o("system_mimic_call", ...)`	who_to_call	who_to_mimic	abi_data	value_to_put_into_r3	value_to_put_into_r4	value_to_put_into_r5	value_to_put_into_r6	It is a call, so it WILL mess up the registers and WILL use `r1-r4` for our standard ABI convention and `r5` for the extra `who_to_mimic` argument.	Runtime `*{i256, i1} __system_mimiccall(i256, i256, i256, i256, i256, *{i256, i1})`
`mimic_call_byref`	`verbatim_2i_1o("mimic_call_byref", ...)`	who_to_call	who_to_mimic						It is a call, so it WILL mess up the registers and WILL use `r1-r4` for our standard ABI convention and `r5` for the extra `who_to_mimic` argument. Uses the active pointer.	Runtime `*{i256, i1} __mimiccall_byref(*i8 addrspace(3), i256, i256, *{i256, i1})`
`system_mimic_call_byref`	`verbatim_6i_1o("system_mimic_call_byref", ...)`	who_to_call	who_to_mimic	value_to_put_into_r3	value_to_put_into_r4	value_to_put_into_r5	value_to_put_into_r6		It is a call, so it WILL mess up the registers and WILL use `r1-r4` for our standard ABI convention and `r5` for the extra `who_to_mimic` argument. Uses the active pointer.	Runtime `*{i256, i1} __system_mimiccall_byref(*i8 addrspace(3), i256, i256, i256, i256, *{i256, i1})`
`raw_call`	`verbatim_4i_1o("raw[_<type>]_call", ...)` type = ‘’ | static | delegate	who_to_call	abi_data (CAN be with “to system = true”)	output_offset	output_length					Default wrapper for the corresponding call type. The ABI data is integer.
`raw_call_byref`	`verbatim_3i_1o("raw[_<type>]_call_byref", ...)` type = ‘’ | static | delegate	who_to_call	output_offset	output_length					Uses the active pointer.	Default wrapper for the corresponding call type. The ABI data is `*i8 addrspace(3)`.

`system_call`	`verbatim_6i_1o("system[_<type>]_call", ...)` type = '' | static | delegate	who_to_call	abi_data (MUST have "to system" set)	value_to_put_into_r3	value_to_put_into_r4	value_to_put_into_r5	value_to_put_into_r6			Runtime `*{i256, i1} __system_[type]call(i256, i256, i256, *{i256, i1})`
`system_call_byref`	`verbatim_5o_1o("system[_<type>]_call_byref", ...)` type = '' | static | delegate	who_to_call	value_to_put_into_r3	value_to_put_into_r4	value_to_put_into_r5	value_to_put_into_r6				Runtime `*{i256, i1} __system_[type]call_byref(*i8 addrspace(3), i256, i256, *{i256, i1})`
`set_context_u128`	`verbatim_1i_0o("set_context_u128", ...)`	value							Uses the active pointer.	
`set_pubdata_price`	`verbatim_1i_0o("set_pubdata_price", ...)`	price								
`increment_tx_counter`	`verbatim_0i_0o("increment_tx_counter", ...)`									
`event_initialize`	`verbatim_2i_0o("event_initialize", ...)`	in0 (u256)	in1 (u256)							
`event_write`	`verbatim_2i_0o("event_write", ...)`	in0 (u256)	in1 (u256)							
`load_calldata_into_active_ptr`	`verbatim_0i_0o("calldata_ptr_to_active", ...)`								loads value of `@calldataptr` (from `r1` into virtual `ACTIVE_PTR`)	
`load_returndata_into_active_ptr`	`verbatim_0i_0o("return_data_ptr_to_active", ...)`								loads value of the latest `@returndataptr` (from `r1` into virtual `ACTIVE_PTR`)	
`ptr_add_into_active`	`verbatim_1i_0o("active_ptr_add_assign", ...)`	offset							performs `ptr.add ACTIVE_PTR, in1, ACTIVE_PTR`	
`ptr_shrink_into_active`	`verbatim_1i_0o("active_ptr_shrink_assign", ...)`	offset							performs `ptr.shrink ACTIVE_PTR, in1, ACTIVE_PTR`	
`ptr_pack_into_active`	`verbatim_1i_0o("active_ptr_pack_assign", ...)`	data							performs `ptr.pack ACTIVE_PTR, in1, ACTIVE_PTR`	
`multiplication_high`	`verbatim_2i_1o("mul_high", ...)`	operand_1	operand_2						Returns the higher register (the overflown part)	
`get_global`	`verbatim_0i_1o("get_global::<name>", ...)` (`<name>` from the table below)	index							Pointers are loaded as INTEGERS!	value of the corresponding global. Note: it's largely to bind the "global" into solidity variable, and actual logic will be done with some other instruction
`throw`	`verbatim_i0_o0("throw", ...)`								Throws a local LLVM exception	

List of globals (zero-enumerated in the order below for purposes of `get_global`):

- `ptr_calldata` - one passed in `r1` on `far_call` to the callee (save in very first instructions on entry)
- `call_flags` - one passed in `r2` on `far_call` to the callee (save in very first instructions on entry)
- `extra_abi_data_{N}` - ones passed in `r3-r12` on `far_call` to the callee (save in very first instructions on entry), `0 <= N <= 9`
- `ptr_return_data` - one passed in `r1` on return from `far_call` back to the caller (save in very first instruction in the corresponding branch!)

Requirements for calling system contracts

By default, all system contracts at addresses 0x80XX require that the call was done via system call (i.e. `call_flags&2 != 0` .

Exceptions:

- BOOTLOADER_FORMAL address as the users need to be able to send money there.

Meaning of ABI params:

- MSG_VALUE_SIMULATOR: `extra_abi_data_1 = value || whether_the_call_is_system`, where || denotes the concatenation, value should occupy first 128 bits, while `whether_the_call_is_system` is a 1-bit flag that denotes whether the call should be a system call. `extra_abi_data_2` is the address of the callee.
- No meaning for the rest