Library EraVM.Introduction

This is the specification of the instruction set of EraVM 1.4.1, a language virtual machine for zkSync Era. It describes the virtual machine's architecture, instruction syntax and semantics, and some elements of system protocol.

Table of contents

- `MemoryForwarding` : a mechanism to pass memory slices between contracts.
- `Events` : different types of events emitted when EraVM is operating.
- `KernelMode` : privileged mode of execution for system contracts allowing a restricted part of the instructions set.
- `StaticMode` : mode of execution with limited effects on persistent state.
- `SmallStep` : formal semantics of instruction execution cycle, in small-step operational style
- `Panics` : a mechanism of signaling irrecoverable errors and a list of situations where they occur.

- **Instruction set semantic**
  - Stack manipulation

    SpAdd : `SpAddDefinition` : forward stack pointer.
    SpSub : `SpSubDefinition` : rewind stack pointer.

  - Arithmetic

    Add : `AddDefinition` :
    Sub : `SubDefinition` :
    Mul : `MulDefinition` :
    Div : `DivDefinition` :

  - Bitwise logical

    And : `AndDefinition` :
    Or : `OrDefinition` :
    Xor : `XorDefinition` :

  - Bitwise shifts

    Shl : `ShlDefinition` : left logical shift.
    Shr : `ShrDefinition` : right logical shift.
    Rol : `RolDefinition` : left circular shift.
    Ror : `RorDefinition` : right circular shift.

  - Control flow

    Nop : `NopDefinition` : do nothing.

    Jump : `JumpDefinition` : jump to code (conditional jumps are implemented through `Predication`).

    NearCall : `NearCallDefinition` : call a function in the same contract.

    NearRet : `NearRetDefinition` : normal return from near call to the call site, return unspend ergs.

    NearRevert : `NearRevertDefinition` : return from near call due to a recoverable error, return unspend ergs, roll back storage/events, execute exception handler.

NearRetTo : `NearRetToDefinition` : Like `NearRet` but returns to explicit label.

NearRevertTo : `NearRevertToDefinition` : like `NearRevert` but executes code at label instead of exception handler.

Panic : `PanicDefinition` : trigger panic.

NearPanicTo : `NearPanicToDefinition` : trigger panic and return to label.

Farcall : `FarcallDefinition` : call a contract

FarRet : `FarRetDefinition` : return from farcall.

FarRevert : `FarRevertDefinition` : like `FarRet` but roll back storage/events and execute exception handler.

- Operations with heaps

LoadPtr : `LoadPtrDefinition` : load word by a fat pointer.
LoadPtrInc : `LoadPtrIncDefinition` : like `LoadPtr`, additionally return an incremented pointer.
Load : `LoadDefinition` : load word from heap by a fat pointer.
LoadInc : `LoadIncDefinition` : like `Load`, additionally return offset+32.
Store : `StoreDefinition` : store word to heap by an offset.
StoreInc : `StoreIncDefinition` : like `Store`, additionally return offset+32.

- Operations with pointers

PtrAdd : `PtrAddDefinition` : increment pointer.
PtrSub : `PtrSubDefinition` : decrement pointer.
PtrShrink : `PtrShrinkDefinition` : restrict the range of addresses that a pointer can reference.
PtrPack : `PtrPackDefinition` : put data in the unused high 128 bits of a pointer.

- Operations with storage

SStore : `SStoreDefinition` : store value at a key in storage of the current contract.
SLoad : `SLoadDefinition` : load value by key from storage of the current contract.

- Events and precompiles

OpEvent : `OpEventDefinition` : emit an event.

ToL1 : `ToL1Definition` : emit a message to L1.

PrecompileCall : `PrecompileCallDefinition` : call an extension of VM specific to currently executing system contract.

Context : `ContextDefinition` : access some parts of VM state such as currently executed contract or stack pointer.

- **ABI** (memory layouts of compound data structures serialized to 256-bit words.)
  - `Coder` : general definitions of encoding, decoding, composition and required properties.
  - `NearCallABI`

- `FatPointerABI`
- `FarCallABI`
- `FarRetABI`
- `MetaParametersABI` : layout of result of `ContextMeta`.
- `PrecompileParametersABI` : layout of result of `ContextMeta`.

- **Instruction encoding**
  - `EncodingTools` : binary encoding of different parts of instruction layout.
  - `encode_opcode` : encoding source/destination, addressing modes, and all modifiers as a single 11-bit opcode value.
  - `MachEncoder` : the main assembly instruction encoder definition.

- **Elements of protocol**
  - `Bootloader` : a system contract in charge of block construction.
  - `Precompiles` : extensions of virtual machine.
  - `VersionedHash` : used to fetch the contract code.
  - `Decommitter`

Library EraVM.Glossary

```
Section Glossary.
```

Notations

- Definitional equality is denoted with $A{:=}B$. Its meaning is: A can be substituted with B.
- Ranges of numbers are denoted as follows:

  $$[start,limit){:=}\{n|start{\leq}n{<}limit\}$$

  In other words, start of the range is included, and limit is excluded.
- Accessing subranges of a binary representation of a number is denoted with $\{low,high\}$. For example, this denotes a binary number obtained by taking bits from 128-th inclusive to to 256-th exclusive of the value op:

  $$op\{128,256\}$$

- Concatenation of sequences of binary numbers is denoted with $\#\#$

  For example, the following denotes concatenating bit representations of the numbers a and b:

$$a\#\#b$$

Glossary

- **ABI** – application binary interface. See `ABI`.
- **Active external frame** – the closest external frame to the top of call stack. For example, in a callstack `(InternalCall (InternalCall (ExternalCall (InternalCall …))))` the active external frame will be the third frame in stack. See `active_extframe`.
- **Auxheap** – one of two heap variants (heap and auxheap), mostly used by system contracts. Executing one of far call instructions creates a new external frame and allocates pages for code, constants, data stack, heap and aux heap.
- **Burning ergs** – setting ergs to zero in topmost call stack frame, external or internal. See `Ergs`.
- **Callstack** – a stack of context information. Executing one of near call instructions pushes a frame of type `InternalCall` to the callstack, executing one of far call instructions pushes a frame of type `ExternalCall`. See `CallStack`.
- **Checkpoint** – see `state_checkpoint`. Not to confuse with EraVM snapshot.
- **Code page** – a read only page filled with `instruction_predicated`. Created by far calls, filled with code obtained from `Decommitter`. See `code_page`.
- **Const page** – a read only page filled with constant data. Created by far calls, filled with constants obtained from `Decommitter`. Can be implemented by putting constants to code pages instead. See `const_page`.
- **Context instructions** – instructions implemented as variants of `context` machine instruction:
  - `OpContextCaller`
  - `OpContextCodeAddress`
  - `OpContextErgsLeft`
  - `OpContextGetContextU128`
  - `OpContextIncrementTxNumber`
  - `OpContextMeta`
  - `OpContextSetContextU128`
  - `OpContextSetErgsPerPubdataByte`
  - `OpContextSp`
  - `OpContextThis`
- **Running instance of a contract, or a function** – a set of memory pages, call stack frames, and other resources associated with a running contract or function. They are distinct per function/ contract call.
- **Current contract** – contract currently being executed. See `ecf_this_address` and `active_extframe`.
- **Current function** – the most recent of functions currently being executed.
- **Data page** – one of types of memory pages. See `data_page`.
- **Data stack** – a stack implicitly operated upon by instructions using address modes like `RelSpPop` or `RelSP`. Located on pages of type `stack_page`. Every contract instance has its own stack; functions invoked by `OpNearCall` share the stack with their caller.
- **Decommitter** – a module responsible for storing contract code and providing it upon query.
- **Exception handler** – a `code_address` of a piece of code associated to a call stack frame. This code will be executed if the corresponding function reverts or panics.
- **External/internal frame, function/contract frame** –
- **Far call** – execution of one of the following instructions: `OpFarCall`, `OpMimicCall`, `OpDelegateCall`.

- **Fat pointer** – a full 128-bit `data_page` pointer encoding page id, a span of addresses from some starting address and with a specified length, and an offset inside this span. See `fat_ptr`.
- **GPR, general purpose register** – one of 16 registers `r0`–`r15` containing primitive values. Register `r0` is a constant register, can not be written to and is read-only.
- **Heap** – one of two heap variants (heap and auxheap), mostly used by system contracts. Executing one of far call instructions creates a new external frame and allocates pages for code, constants, data stack, heap and aux heap.
- **Heap pointer** – a pair of address in heap and a limit; it is associated with a span $[0;limit)$. See `heap_ptr`.
- **Heap variant** – either Heap or Auxheap, depending on the context.
- **Integer value** – a `primitive_value` with a reset pointer tag. See `IntValue`.
- **L1** – level-1, refers to the main Ethereum blockchain, also known as the Ethereum Mainnet. Used to distinguish from scaling solutions or Layer 2 solutions that aim to improve the scalability and throughput of the Ethereum blockchain.
- **Log instructions** – variations of `log` machine instruction: `OpSLoad`, `OpSStore`, `OpEvent`, `OpToL1Message`, `OpPrecompileCall`.
- **Memory forwarding** – a mechanism of sharing a read-only fragment ofmemory between contracts. The memory fragment is created from `span` and described by `fat_ptr`. This fragment can be narrowed or shrunk as a result of far call or executing `OpPtrShrink`.
- **Machine instruction** – a low-level machine instruction with a fixed format. The high-level `instruction_predicated` is encoded to machine instructions.
- **Memory growth** – a process, where an access to a heap variant beyond its bound leads to increasing the bound and payment.
- **Narrowing a fat pointer** – subtract a given number from its length and add it to its start; it is guaranteed to not overflow. See `fat_ptr_narrow`. Used when passing a fat pointer to a far call, or returning it from a contract.
- **Near call** – calling a function that belongs to the same contract.
- **Operand** – data or the address that is operated upon by the instruction. It represents the input or output values used by the instruction to perform a specific operation or computation. See `InstructionArguments`.
- **Page** – see `page_type`.
- **Pointer value** – a `primitive_value` with a set pointer tag. May contain a fat pointer. See `PtrValue`.
- **Precompile call** – an invokation of `OpPrecompileCall`. Precompiles are extensions of EraVM bound to one of the system contracts. When this contract executes an instruction `OpPrecompileCall`, EraVM executes an algorithm specific to this contract. See `Precompiles`.
- **Precompile processor** – a module responsible for encoding the algorithms of precompile calls and executing them.
- **Primitive value** – a tagged word. See `primitive_value`.
- **Shrinking a fat pointer** – subtract a given number from its length; it is guaranteed to not overflow. See `fat_ptr_shrink`. Triggered by `OpPtrShrink` instruction.
- **Slice** – see `slice`.
- **Span** – see `span`.
- **Stack page** – a type of pages. See `stack_page`.
- **System contracts** – contracts with addresses from 0 to `KERNEL_MODE_MAXADDR_LIMIT`. They are executed in `KernelMode`.
- **Topmost callstack frame** – the last frame pushed to call stack.
- **Word** – 256-bit unsigned untagged integer value.
- **Address resolution** – a matching between instruction operands and locations using the supported address modes. See `resolve`.
- **Base cost** – the fixed cost of executing instruction, in ergs. Some instructions imply additional costs, e.g. far calls may require paying for code decommitment.
- **Bootloader** – a system contract written in YUL in charge of block construction.
- **Cell** – alias to “word”. Often used to distinguish between values themselves and the memory locations holding them.
- **Depot** – all storages in all shards. See `depot`.

- **Ergs** – resource spent on actions such as executing instructions. See `Ergs`.
- **Instruction predicate** – see `Predication`.
- **Flag** – see `flag_state`.
- **Fully qualified address** – see `fqa_key`.
- **Instruction** – low-level command or operation that is executed by a virtual machine to perform a specific task. Instructions supported by EraVm are described by the type `instruction_predicated`.
- **Kernel mode** – a mode of execution for system contracts opening access to full instruction set, containing instructions potentially harmful to the global state
  e.g. `OpContextIncrementTxNumber`. See `KernelMode`.
- **Key** – a 256-bit address of a cell in storage.
- **Location** – see `loc`.
- **Panic** – irrecoverable error. Handled by formally executing `OpPanic`.
- **Revert** – execution of `OpRevert`, usually as a consequence of recoverable error.
- **Malformed transaction** – a transaction rejected by the bootloader. Handling it is the responsibility of the server that controls EraVM.
- **Predicate** – see `Predication`.
- **Predication** – see `Predication`.
- **Program counter** – the `code_address` of the next instruction to be executed. See `cf_pc`.
- **Rollback** – restoration of the `gs_revertable` portion of `state` as a result of revert or panic. The state is saved at every near or far call. See also `state_checkpoint`.
- **Shard** – a collection of `storage`s. See `shard`.
- **Snapshot** – a copy of the full state of EraVM. Server makes a snapshot of VM state every time a transaction is successfully completed. When the bootloader encounters a malformed transaction, it fails, and the server restarts EraVM from the most recent snapshot, skipping this transaction.
- **Server** – a program that launches EraVM and controls it. Feeds the transactions to the bootloader, provides decommitter and other external modules, restarts EraVM from the latest snapshot in case of malformed transactions.
- **Stack pointer** – the `stack_address` where the next element will be pushed. It is the address of the (top of the stack + 1).
- **Static mode** – see `StaticMode`.
- **Storage** – see `storage`.
- **Total cost** – a sum of `base_cost` of an instruction and all its additional costs.
- **Versioned hash** – a key used to retrieve the contract code from decommitter. See `versioned_hash` and `Decommitter`.
- **VM** – the same as EraVM, the abstract virtual machine that this document specifies.

```
End Glossary.
```

Library EraVM.Core

```
Require Common.
Import Common ZArith Types.

Section Parameters.
```

EraVM architecture overview

EraVM is a 256-bit register-based language machine with two stacks and dedicated memory for code, data, stack and constants.

```
  Definition word_bits: nat := 256.

  Definition word: Type := BITS word_bits.
```

word0 is a word with a zero value.

```
  Definition word0: word := fromZ 0%Z.
```

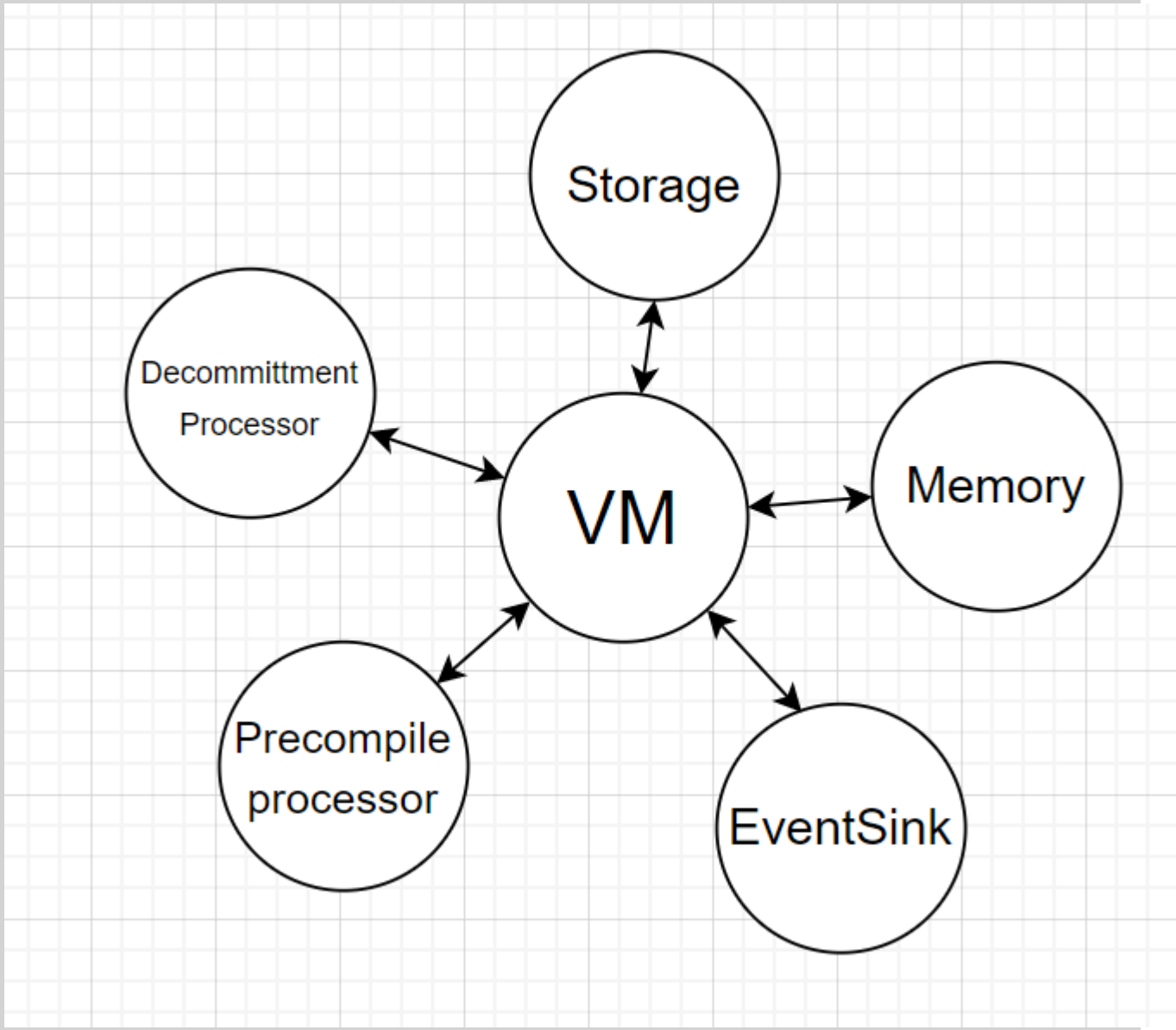
Helpers

```
  Definition bytes_in_word : nat := word_bits/bits_in_byte.
  Definition z_bytes_in_word : Z := Z.of_nat bytes_in_word.

  Definition word_to_bytes (w:u256) : tuple.tuple_of 32 u8 := @bitsToBytes 32
w.
  Definition bytes_to_word (bs: tuple.tuple_of 32 u8) : word := @bytesToBits _
bs.

End Parameters.

Section ArchOverview.
```



- **Memory** provides access to transient memory pages. See `Memory`.
- **Storage** provides access to persistent storage with two shards, each shard maps 2^{160} contracts, to a key-value storage. See `Depot`.
- **EventSink** collects events and messages to L1. See `Events`.
- **Precompile processor** executes system contract-specific extensions to the EraVM instruction set, called precompiles, e.g. `keccak256`, `sha256`, and so on. See `Precompiles`.
- **Decommitment processor** stores and decommits the code of contracts. See `Decommitter`.

Functions and contracts

In EraVM, contracts are the coarse-grained execution units. Contracts may have multiple **functions** in them; a contract may execute its functions by using **near call** instruction `OpNearCall`. A contract may also call other contracts by using **far call** instructions `OpFarCall`, `OpMimicCall`, or

`OpDelegateCall`.

A **running instance** of a function or a contract is a piece of VM runtime state associated with the current execution of a function or a contract, as described by `callstack`.

EraVM allows recursive execution of functions and contracts. For example, a contract A calls a function f_1 which calls a function f_2 which calls a contract B, which calls a function g_1, which calls A again …

$$A \to f_1 \to f_2 \to B \to g_1 \to A \to \dots$$

During the execution of g_1, **running instances** of A, f_1, f_2 keep existing, waiting for the control to return to them.

Executing a contract allocates memory pages dedicated to it (see `alloc_pages_extframe`). In a running instance of a contract, this contract's functions share these memory pages.

Contracts have more contract-specific state associated to them than functions (compare `InternalCall` and `ExternalCall`). However, running instances of both functions and contracts have their own exception handlers, program counters, stack pointers and allocated ergs (see `callstack_common`).

Execution state

EraVM's functionality is to sequentially execute instructions.

The main components of EraVM's execution state are:

- 256-bit tagged general-purpose registers R1–R15 and a reserved constant register R0 holding 0. See `regs_state`.
- Three flags: OverFlow/Less-Than, EQuals, Greater-Than. See `flags_state`.
- Data stack containing tagged words. It is located on a dedicated `stack_page`.
- Callstack, holding callframes, which contain such information as the program counter, data stack pointer, ergs for the current function/contract instance, current contract's address, and so on. See `CallStack`.
- Frames in callstack; can be internal (belong to a function, near called) frames or external frames (belong to a contract, far called, richer state).
- Read-only pages for constants and code, one per contract stack frame.

Instructions

The type `asm_instruction` describes the supported instructions.

All instructions are `predicated`: they contain an explicit condition related to the current flags state. If the condition is satisfied, they are executed, otherwise they are skipped (but their `basic_cost` is still paid).

Instructions accept data and return results in various formats:

- The formats of instruction operands are described in `Addressing`.
- The address resolution to locations in memory/registers is described in `resolve`

- Reading and writing to memory is described in `MemoryOps`.

Modes

EraVM has two modes which can be independently turned on and off.

1. Kernel mode/User Mode

First `KERNEL_MODE_MAXADDR_LIMIT` contracts are marked as **system contracts**. EraVM executes them in kernel mode, allowing an access to a richer instruction set, containing instructions potentially harmful to the global state e.g. `OpContextIncrementTxNumber`. See `KernelMode`.

2. Static mode/Non-static mode

   Intuitively, executing code in static mode aims at limiting its effects on the global state, similar to executing pure functions. Globally visible actions like emitting events or writing to storage are forbidden in static mode. See `StaticMode`.

Ergs

Instructions and some other actions should be paid for with a resource called **ergs**, similar to Ethereum's gas. See the overview in `Ergs`.

Operation

The VM is started by a server that controls it and feeds the transactions to the `Bootloader`.

Context of EraVM

When the server needs to build a new batch, it starts an instance of EraVM.

EraVM accepts three parameters:

1. Bootloader's `versioned_hash`. It is used to fetch the bootloader code from `Decommitter`.
2. Default code hash `DEFAULT_AA_VHASH`. It is used to fetch the default code from `Decommitter` in case of a far call to a contract without any associated code.
3. A boolean flag `is_porter_available`, to determine the number of shards (two if zkPorter is available, one otherwise).

Bootloader is a contract written in YUL in charge of block construction. See `Bootloader`.

EraVM retrives the code of bootloader from `Decommitter` and proceeds with sequential execution of instructions on the bootloader's code page.

Failures and rollbacks

There are three types of behaviors triggered by execution failures.

1. Skipping a malformed transaction. It is a mechanism implemented by the server, external to EraVM. Server makes a snapshot of EraVM state after completing every transaction. When the bootloader encounters a malformed transaction, it fails, and the server restarts EraVM from the most recent snapshot, skipping this transaction.

   This behavior is specific to bootloader; the contract code has no ways of invoking it.

2. Revert is triggered by the contract code explicitly by executing `OpRevert`. EraVM saves its persistent state to `state_checkpoint` on every near or far call. If the contract code identifies a recoverable error, it may execute `OpRevert`. Then EraVM rolls the storage and event queues back to the last `state_checkpoint` and executes the exception handler. See `rollback`.

3. Panic is triggered either explicitly by executing `OpPanic`/`OpNearPanicTo`, or internally when some execution invariants are violated. For example, attempting to execute in user mode an instruction, which is exclusive to kernel mode, results in panic.

   On panic, the persistent state of EraVM is rolled back in the same way as on revert. See `rollback`.

```
End ArchOverview.
Definition timestamp := nat.
Definition tx_num := u16.
```

Library EraVM.PrimitiveValue

```
Require Types.

Section PrimitiveValue.
```

Primitive values

Primitive value is a tagged word. The tag `is_ptr` is a boolean.

- If `is_ptr` is set, it is guaranteed that the lowest 128-bits of the word contain a serialized `fat_ptr`. Such values can be used as operands in instructions that require pointer arguments, for example, `OpPtrAdd`. The other half (128 most significant bits) may hold meaningful data, e.g. when forming a value according to FarCall ABI using `Assembly.OpPtrPack` instruction.
- If `is_ptr` is cleared, there are no guarantees to its value. It may contain an integer, a representation of a `heap_ptr`, a representation of a `span` of addresses, etc.

```
Context {word:Type}.
Inductive primitive_value :=
  mk_pv {
      is_ptr: bool;
      value: word;
    }.
```

Only `Registers` and `stack_pages` hold primitive values; other types of memory, including storage and heap pages, contain non-tagged data.

Function `clear_pointer_tag` clears the pointer tag of a primitive value.

```
Definition clear_pointer_tag (pv:primitive_value): primitive_value :=
  match pv with | mk_pv _ value ⇒ mk_pv false value end.

End PrimitiveValue.
```

For brevity, a primitive value is called **a pointer value** if its tag is set, and **integer value** otherwise.

```
Notation IntValue := (mk_pv false).
Notation PtrValue := (mk_pv true).

Definition pv0 := IntValue Types.zero256.
```

Library EraVM.State

```
From RecordUpdate Require Import RecordSet.
Require ABI Core Decommitter GPR Ergs Event History CallStack TransientMemory
VMPanic.

Import ssreflect ssrfun ssrbool eqtype ssreflect.tuple zmodp.
Import RecordSetNotations.

Import Core Flags ZArith ABI Common GPR Ergs Event CallStack History MemoryBase
memory.Depot Decommitter Predication PrimitiveValue TransientMemory VMPanic.

Section StateDefinitions.

Definition exception_handler := code_address.
```

helpers

```
Definition instruction_invalid : predicated Assembly.asm_instruction := invalid
Assembly.OpInvalid.
Definition decommitter := decommitter instruction_invalid.
Definition code_page := code_page instruction_invalid.
Definition memory := @memory code_page const_page data_page stack_page.
Definition query := @query [ eqType of contract_address] [eqType of
PrecompileParametersABI.params].
Definition event := @event [ eqType of contract_address].
Definition page_has_id := @page_has_id code_page const_page data_page
stack_page.
```

EraVM state

EraVM employs a `state` that comprises the following components:

1. The `global_state` contains:

- current price of publishing one byte of **pubdata** to L1 `gs_current_ergs_per_pubdata_byte`.
- transaction number in the current block `gs_tx_number_in_block`
- decommitter `gs_contracts`
- a revertable part `state_checkpoint`. It houses the **depot** state, embodying all contracts storages across all shards, as well as two queues for events and L1 messages.

  Launching a contract (far call) or a function (near call) defines a checkpoint. If a contract or a function reverts or panics, the state rolls back to the latest snapshot (see `rollback`).

  The rollback may be implemented in any efficient way conforming to this behavior.

```
Record state_checkpoint := {
    gs_depot: depot;
    gs_events: @history [eqType of query];
    gs_l1_msgs: @history [eqType of event];
  }.

Record global_state :=
  mk_gstate {
    gs_current_ergs_per_pubdata_byte: ergs;
    gs_tx_number_in_block: tx_num;
    gs_contracts: decommitter;
    gs_revertable:> state_checkpoint;
    }.

Inductive rollback checkpoint: global_state → global_state → Prop :=
| rb_apply: ∀ e tx ccs _cp,
  rollback checkpoint (mk_gstate e tx ccs _cp) (mk_gstate e tx ccs checkpoint).
```

2. The `transient_state` contains:
   - flags `gs_flags`: boolean values representing some characteristics of the computation results. See `Flags`.
   - general purpose registers `gs_regs`: 15 mutable tagged words (primitive values) `r1`–`r15`, and a reserved read-only zero valued `r0`. See `Registers`.
   - all memory `gs_pages` allocated by VM, including code pages, data stack pages, data pages for heap variants etc. See `memory`.
   - `gs_callstack`, where each currently running contract and function has a stack frame. Note, that program counter, data stack pointer, and the remaining ergs allocated for the current function's run are parts of a stack frame. See `Callstack`.

- 128-bit register `gs_context_u128`. Its usage is detailed below.

helpers

```
Definition callstack := @callstack state_checkpoint.
Record transient_state :=
  mk_transient_state {
      gs_flags : flags_state;
      gs_regs: regs_state;
      gs_pages: memory;
      gs_callstack: callstack;
      gs_context_u128: u128;
      gs_status: status;
    }.

Record state :=
  mk_state {
      gs_transient :> transient_state;
      gs_global :> global_state;
    }.
```

Context register

The 128-bit context value occurs in two places in EraVM:

- In the mutable `gs_context_u128` register, forming a part of the EraVM state `state`.
- In the read-only `ecf_context_u128_value` of each external call stack frame `callstack_external`.

These values are distinct: the value in callstack is a snapshot of the register `gs_context_u128` in a moment of a far call.

The typical usage of the context value is as follows:

1. Set the value of `gs_context_u128` to C by executing the instruction `OpContextSetContextU128`.
2. Launch a contract using one of the far call instructions. This action pushes a new `callstack_external` frame F onto the `gs_callstack`. The value of the F's field `ecf_context_u128_value` is equal to C. In addition, far calls reset `gs_context_u128` to 0.
3. Retrieve the context value by executing the instruction `OpContextGetContextU128` to use it.
4. On contract code completion, the `gs_context_u128` is reset to zero by either `OpFarRet`, `OpFarRevert`, or `OpPanic`.

Note that setting the context register `gs_context_u128` is forbidden in `StaticMode`. See `forbidden_static`.

Context is used to simulate `msg.value`, a Solidity construction standing for the amount of wei sent in a

transaction. The system contract `MsgValueSimulator` is responsible for ensuring that whenever this context value is set to C, there are indeed C wei transferred to the callee.

Helpers

```
Inductive global_state_new_depot: depot → global_state → global_state → Prop :=
| gsnd_apply: ∀ current_ergs_per_pubdata_byte tx codes d evs l1s d',
  global_state_new_depot d'
  {|
    gs_current_ergs_per_pubdata_byte := current_ergs_per_pubdata_byte;
    gs_tx_number_in_block := tx;
    gs_contracts := codes;
    gs_revertable := {| gs_depot := d; gs_events := evs; gs_l1_msgs := l1s |} ;
  |}
  {|
    gs_current_ergs_per_pubdata_byte := current_ergs_per_pubdata_byte;
    gs_tx_number_in_block := tx;
    gs_contracts := codes;
    gs_revertable := {| gs_depot := d'; gs_events := evs; gs_l1_msgs := l1s |}
;
  |}.

Inductive emit_event e: global_state → global_state → Prop :=
| ee_apply: ∀ current_ergs_per_pubdata_byte tx codes d evs l1s d',
  emit_event e
  {|
    gs_current_ergs_per_pubdata_byte := current_ergs_per_pubdata_byte;
    gs_tx_number_in_block := tx;
    gs_contracts := codes;
    gs_revertable := {| gs_depot := d; gs_events := evs; gs_l1_msgs := l1s |} ;
  |}
  {|
    gs_current_ergs_per_pubdata_byte := current_ergs_per_pubdata_byte;
    gs_tx_number_in_block := tx;
    gs_contracts := codes;
    gs_revertable := {| gs_depot := d'; gs_events := e::evs; gs_l1_msgs := l1s
|} ;
  |}.

Inductive emit_l1_msg e: global_state → global_state → Prop :=
| eel1_apply: ∀ current_ergs_per_pubdata_byte tx codes d evs l1s d',
  emit_l1_msg e
  {|
    gs_current_ergs_per_pubdata_byte := current_ergs_per_pubdata_byte;
    gs_tx_number_in_block := tx;
    gs_contracts := codes;
    gs_revertable := {| gs_depot := d; gs_events := evs; gs_l1_msgs := l1s |} ;
  |}
  {|
    gs_current_ergs_per_pubdata_byte := current_ergs_per_pubdata_byte;
    gs_tx_number_in_block := tx;
    gs_contracts := codes;
    gs_revertable := {| gs_depot := d'; gs_events := evs; gs_l1_msgs := e::l1s
```

```
|} ;
  |}.

Inductive tx_inc : tx_num → tx_num → Prop := | txi_apply: ∀ n m, uinc_of n =
(false, m) → tx_inc n m.

Inductive global_state_increment_tx tx_mod: global_state → global_state → Prop
:=
| gsit_apply: ∀ current_ergs_per_pubdata_byte tx new_tx codes rev ,
  tx_mod tx new_tx →
  global_state_increment_tx tx_mod
  {|
    gs_current_ergs_per_pubdata_byte := current_ergs_per_pubdata_byte;
    gs_tx_number_in_block := tx;
    gs_contracts := codes;
    gs_revertable := rev;
  |}
  {|
    gs_current_ergs_per_pubdata_byte := current_ergs_per_pubdata_byte;
    gs_tx_number_in_block := new_tx;
    gs_contracts := codes;
    gs_revertable := rev;
  |}.

End StateDefinitions.

Definition heap_variant_page_id (page_type: data_page_type)
  : callstack → page_id :=
  match page_type with
  | Heap ⇒ @active_heap_id state_checkpoint
  | AuxHeap ⇒ @active_auxheap_id state_checkpoint
  end.

Definition heap_variant_page (page_type: data_page_type) (cs:callstack)
(mem:memory) :=
    match page_type with
    | Heap ⇒ active_heappage
    | AuxHeap ⇒ active_auxheappage
    end (page_has_id mem) cs .

Definition heap_variant_id (page_type: data_page_type)
  : callstack → page_id :=
  match page_type with
  | Heap ⇒ @active_heap_id
  | AuxHeap ⇒ @active_auxheap_id
  end state_checkpoint.
```

Library EraVM.GPR

```
From RecordUpdate Require Import RecordSet.
Require Core List PrimitiveValue.

Import Core PrimitiveValue.
```

```
Section Registers.
  Import RecordSetNotations.
  Context (pv := @primitive_value Core.word).
```

GPR (General Purpose Registers)

EraVM has 15 mutable general purpose registers R1, R2, …, R15. They hold `primitive_value word`, so they are tagged 256-bit words. The tag is set when the register contains a fat pointer in its 128 least significant bits (it may contain other useful data in topmost 128-bits; this is used e.g. for encoding parameters of `FarCall`).

Additionally, EraVM has one read-only, **constant register** R0 which evaluates to `IntValue 0`, that is, an untagged integer 0.

```
  Inductive reg_name : Set :=
    R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 | R11 | R12 | R13
  | R14 | R15.

  Record regs_state := mk_regs {
                           r1 : pv;
                           r2 : pv;
                           r3 : pv;
                           r4 : pv;
                           r5 : pv;
                           r6 : pv;
                           r7 : pv;
                           r8 : pv;
                           r9 : pv;
                           r10 : pv;
                           r11 : pv;
                           r12 : pv;
                           r13 : pv;
                           r14 : pv;
                           r15 : pv;
                         }.
```

Function `fetch_gpr` loads a value from register.

```
Definition fetch_gpr (rs:regs_state) (r:reg_name) : pv :=
  match r with
  | R0 ⇒ IntValue word0
  | R1 ⇒ r1 rs
  | R2 ⇒ r2 rs
  | R3 ⇒ r3 rs
  | R4 ⇒ r4 rs
  | R5 ⇒ r5 rs
  | R6 ⇒ r6 rs
  | R7 ⇒ r7 rs
```

```
    | R8 ⇒ r8 rs
    | R9 ⇒ r9 rs
    | R10 ⇒ r10 rs
    | R11 ⇒ r11 rs
    | R12 ⇒ r12 rs
    | R13 ⇒ r13 rs
    | R14 ⇒ r14 rs
    | R15 ⇒ r15 rs
    end.
```

helpers

```
  #[export] Instance etaGPRs : Settable _ := settable! mk_regs < r1; r2; r3;
r4; r5; r6; r7; r8; r9; r10; r11; r12; r13; r14; r15 >.
```

Predicate `store_gpr` stores value to a general purpose register. Storing to `R0` is ignored.

```
  Definition store_gpr (rs: regs_state) (name: reg_name) (pv: primitive_value)
: regs_state :=
    match name with
    | R0 ⇒ rs
    | R1 ⇒ rs <| r1 := pv|>
    | R2 ⇒ rs <| r2 := pv|>
    | R3 ⇒ rs <| r3 := pv|>
    | R4 ⇒ rs <| r4 := pv|>
    | R5 ⇒ rs <| r5 := pv|>
    | R6 ⇒ rs <| r6 := pv|>
    | R7 ⇒ rs <| r7 := pv|>
    | R8 ⇒ rs <| r8 := pv|>
    | R9 ⇒ rs <| r9 := pv|>
    | R10 ⇒rs <| r10 := pv|>
    | R11 ⇒rs <| r11 := pv|>
    | R12 ⇒rs <| r12 := pv|>
    | R13 ⇒rs <| r13 := pv|>
    | R14 ⇒rs <| r14 := pv|>
    | R15 ⇒rs <| r15 := pv|>
    end.

  Definition reg_map f (rs:regs_state) : regs_state :=
    match rs with
    | mk_regs r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 ⇒
        ( mk_regs (f r1) (f r2) (f r3) (f r4) (f r5) (f r6) (f r7) (f r8) (f
r9) (f r10) (f r11) (f r12) (f r13) (f r14) (f r15))
    end.
End Registers.
```

Library EraVM.Flags

```
Require Bool.
Import Bool.

Section Flags.
```

Flags

Flags are boolean values, which reflect certain characteristics of the result of the last instruction.

It is commonly said that a flag is **set** if its value is `true`, and a flag is **cleared** if its value is false.

In EraVM there are three flags:

1. `OF_LT` : overflow or less than;

```
Inductive OF_LT := Set_OF_LT | Clear_OF_LT.
```

2. `EQ` : equals;

```
Inductive EQ := Set_EQ | Clear_EQ.
```

3. `GT` : greater than.

```
Inductive GT := Set_GT | Clear_GT.
```

More specifically:

1. `OF_LT` is set in the following cases:
   - Overflow
   - Underflow
   - Division by zero
   - Panic, then when the exception handler starts execution, `OF_LT` is set.
2. `EQ` is set when the result of a binary operation is zero.
3. `GT` is set when the first operand of a binary operation was greater than the second.

Helpers

```
Definition OF_LT_to_bool (f:OF_LT) := if f then true else false.
Definition EQ_to_bool (f:EQ) := if f then true else false.
Definition GT_to_bool (f:GT) := if f then true else false.

#[reversible]
  Coercion OF_LT_to_bool : OF_LT >-> bool.
#[reversible]
  Coercion EQ_to_bool : EQ >-> bool.
#[reversible]
  Coercion GT_to_bool : GT >-> bool.

Definition EQ_of_bool (f:bool) := if f then Set_EQ else Clear_EQ.
```

```
Definition OF_LT_of_bool (f:bool) := if f then Set_OF_LT else Clear_OF_LT.
Definition GT_of_bool (f:bool) := if f then Set_GT else Clear_GT.
```

State of three flags `flags_state` is stored in the global `state` in the field `gs_xstate` in the field `gs_flags`.

```
Record flags_state := mk_fs {
                        fs_OF_LT: OF_LT;
                        fs_EQ: EQ;
                        fs_GT: GT;
                      }.
```

Usage

Flags are used to control the execution flow. See `Predication`.

Helpers

```
Definition set_overflow fs := match fs with
                              | mk_fs _ EQ GT ⇒ mk_fs Set_OF_LT EQ GT
                              end.

  Definition bflags (OF EQ GT: bool) : flags_state :=
    {|
      fs_OF_LT := OF_LT_of_bool OF;
      fs_EQ := EQ_of_bool EQ;
      fs_GT := GT_of_bool GT
    |}.

  Definition flags_clear : flags_state := mk_fs Clear_OF_LT Clear_EQ Clear_GT.
End Flags.
```

Library EraVM.Predication

```
Require Common Flags.

Import Common Flags.
Import ssrbool.

Section Predication.
```

Predication

Every instruction on the `code_page` is predicated, meaning it is augmented with a `predicate`. A predicate describes a condition on flags; if this condition is satisfied, then the instruction is executed; otherwise, EraVM skips the instruction.

When an instruction is skipped, its base cost is still paid.

```
Inductive predicate : Set :=
| IfAlways | IfGT | IfEQ | IfLT | IfGE | IfLE | IfNotEQ | IfGTOrLT.

Definition predicate_holds (c:predicate) (fs:flags_state) : bool :=
  match c, fs with
  | IfAlways, _
  | IfGT, mk_fs _ _ Set_GT
  | IfEQ, mk_fs _ Set_EQ _
  | IfLT, mk_fs Set_OF_LT _ _
  | IfGE, mk_fs _ Set_EQ _
  | IfGE, mk_fs _ _ Set_GT
  | IfLE, mk_fs _ Set_EQ _
  | IfLE, mk_fs Set_OF_LT _ _
  | IfNotEQ, mk_fs _ Clear_EQ _
  | IfGTOrLT,mk_fs Set_OF_LT _ _
  | IfGTOrLT,mk_fs     Set_GT ⇒ true
  | _,_ ⇒ false
  end.

Inductive predicate_spec: predicate → flags_state → Prop
  :=
| ac_Always: ∀ fs,
    predicate_spec IfAlways fs

| ac_GT: ∀ of_lt eq,
    predicate_spec IfGT (mk_fs of_lt eq Set_GT)

| ac_EQ: ∀ of_lt gt,
    predicate_spec IfEQ (mk_fs of_lt Set_EQ gt)

| ac_LT: ∀ eq gt,
    predicate_spec IfLT (mk_fs Set_OF_LT eq gt)

| ac_GE1: ∀ fs,
    predicate_spec IfEQ fs →
    predicate_spec IfGE fs

| ac_GE2: ∀ fs,
    predicate_spec IfGT fs →
    predicate_spec IfGE fs

| ac_LE1: ∀ fs,
    predicate_spec IfLT fs →
    predicate_spec IfLE fs
```

```
  | ac_LE2: ∀ fs,
      predicate_spec IfEQ fs →
      predicate_spec IfLE fs

  | ac_NotEQ: ∀ of_lt gt,
      predicate_spec IfNotEQ (mk_fs of_lt Clear_EQ gt)

  | ac_IfGTOrLT1: ∀ fs,
      predicate_spec IfGT fs→
      predicate_spec IfGTOrLT fs

  | ac_IfGTOrLT2: ∀ fs,
      predicate_spec IfLT fs→
      predicate_spec IfGTOrLT fs
  .
```

proofs

```
  Hint Constructors predicate_spec:flags.

  Theorem predicate_holds_spec :
    ∀ c fs, predicate_holds c fs = true ↔ predicate_spec c fs.
  Proof.
    split; destruct c, fs as [[] [] []];
      simpl in *; try solve [auto with flags| inversion 1; inversion H0].
  Qed.

  Theorem predicate_activated_dec: ∀ ec flags, decidable (predicate_spec ec
flags).
  Proof.
    intros ec flags.
    unfold decidable.
    destruct ec, flags as [[][][]]; solve [left;constructor| right;inversion 1
| auto with flags | right; inversion 1; subst; inversion H0].
  Defined.

  Record predicated (instruction:Type): Type :=
    Ins {
        ins_spec: instruction;
        ins_cond: predicate;
      }.
```

Invalid instruction. It is a default value on code_pages. See code_page.

```
  Definition invalid {I} (ins:I) : predicated I :=
    {|
      ins_spec := ins;
      ins_cond:= IfAlways
    |}.

End Predication.
```

Library EraVM.TransientMemory

```
Require PrimitiveValue memory.Pages memory.PageTypes Pointer.
Import PrimitiveValue memory.Pages memory.PageTypes Pointer.

Export memory.Pages memory.PageTypes.
```

Informal overview

All memory available to the contract code can be divided into transient and persistent memory.

- **Transient memory** exists to enable computations and does not persist between VM, like the main memory of personal computers.
- **Persistent memory** exists as a storage of untagged 256-bit `word`s shared between the network participants.

Contract code uses transient memory to perform computations and uses the storage to publish its results

Persistent memory

The global persistent data structure is the `depot`. It holds untagged 256-bit words.

Depot is split in two `shard`s: one corresponds to rollup, another to porter (see `shard_exists`).

Each shard is a map from a `contract_address` (160 bit, might be extended in future to up to 256 bits) to contract `storage`.

Each contract `storage` is a linear mapping from 2^{256} **keys** to 256-bit untagged words.

To address a word in any contract's storage, it is sufficient to know:

- shard
- contract address
- key

See `fqa_key`.

Contract has one independent storage per shard.

One shard is selected as currently active in `state`.

Contract code is global and shared between shards.

Transient memory

Transient memory consists of **pages** (`page_type`) holding data or code. Each page holds 2^{32} bytes; all bytes are initialized to zero at genesis.

New pages are allocated implicitly when the contract execution starts; calling another contract allocates more pages. Pages persist for as long as they are referenced from the live code.

Pages hold one of:

- data: 2^{32} byte-addressable data for heap or auxheap; bounded, so reading or writing outside bounds leads to a paid growth of available portion.
- code: instruction-addressable, read-only;
- constants: 2^{16} word-addressable, read-only;
- stack: 2^{16} words, word-addressable, tagged words. When the execution of a contract starts, the initial value of stack pointer is `INITIAL_SP_ON_FAR_CALL`.

The following describes all types of memory formally and with greater detail.

The definition `vm_page` collects the specific types of pages used by EraVM semantic.

```
Definition stack_address := stack_address pv0.
Definition stack_address_bits := stack_address_bits pv0.
Definition stack_page := stack_page pv0.
Definition stack_page_params := stack_page_params pv0.

#[global]
  Canonical vm_page {instr} (inv:instr) type : page := @mk_page (code_page inv)
const_page data_page stack_page type.
#[global]
  Canonical vm_mem {instr} (inv:instr) type : memory := @mk_pages (code_page
inv) const_page data_page stack_page type.
```

Library EraVM.memory.Depot

```
Require Core PrimitiveValue MemoryBase.

Import Common Core MemoryBase BinInt PrimitiveValue.

Section Depot.
```

Storage of a contract

A **storage** is a persistent linear mapping from 2^{256} addresses to words.

Therefore, given a storage, each word storage is addressed through a 256-bit address.

In storage, individual bytes inside a word can not be addressed directly: a load or a store happen on a word level.

Both word address in storage and word width are 256 bits.

All words in a storage are zero-initialized on genesis. Therefore, reading storage word prior to writing yields zero. It is a well-defined behavior.

```
Definition storage_params := {|
                               addressable_block := word;
                               address_bits := 256;
                               default_value := word0;
                               writable := true
                             |}.

Definition storage_address := BITS (address_bits storage_params).
Definition storage: Type := mem parameterized storage_params.
```

Storage start blanks.

```
Definition storage_empty : storage := empty storage_params.
```

Storage does not hold contract code, it is a responsibility of decommittment processor `decommitter`.

Storage is a part of a `shard`, which is a part of `depot`.

One storage is selected as an active storage, it is the storage corresponding to the `current_shard` and `current_contract`.

Use the instruction `OpSStore` to write to the active storage, `OpSLoad` to read from the active storage.

Instructions

Instruction `OpSLoad` implements reading from storage; instruction `OpSStore` implements writing to storage.

Memory model

Storage has a sequentially-consistent, strong memory model. All writes are atomic and immediately visible; reads are guaranteed to return the last value written.

Shards and contracts

A **contract** is uniquely identified by its 160-bit address `contract_address`. In future, the address could be seamlessly extended to up to 256 bits.

A **shard** is a mapping of contract addresses to storages.

Therefore, every contract is associated with as many storages as there are shards.

```
Definition shard_params := {|
                            addressable_block := storage;
                            address_bits := 160;
                            default_value := storage_empty;
                            writable := true
                           |}.
Definition contract_address := address shard_params.
Definition contract_address_bits := address_bits shard_params.
Definition shard := mem_parameterized shard_params.
```

Contracts are also associated with code. The association is global per depot and implemented by `Decommitter`. Therefore, the contract code is the same for all shards, but the storages of a contract in different shards differ.

Unlike in Ethereum, there is only type of accounts capable of both transacting coins and executing contracts.

Contracts with addresses in range from 0 (inclusive) to `KERNEL_MODE_MAXADDR` (exclusive) are **system contracts**; they are allowed to execute all instructions.

Depot

Depot is a collection of shards. Depot is the global storage of storages of all contracts.

```
Definition depot_params :=
  {|
    addressable_block := shard;
    address_bits := 8;
    default_value := empty _;
    writable := true
```

```
  |}.

Definition depot := mem_parameterized depot_params.

Definition shard_id := BITS (address_bits depot_params).
```

There are currently two shards: one for zkRollup, one for zkPorter.

```
Inductive shard_exists: shard_id → Prop :=
| se_rollup: shard_exists (fromZ 0)
| se_porter: shard_exists (fromZ 1)
.
```

The **fully qualified address** of a word in depot is a triple:

$$(\texttt{shard\_id}, \texttt{contract\_id}, \texttt{key})$$

It is formalized by `fqa_key`.

```
Record fqa_storage := mk_fqa_storage {
                        k_shard: shard_id;
                        k_contract: contract_address;
                      }.
  Record fqa_key := mk_fqa_key {
                      k_storage :> fqa_storage;
                      k_key: storage_address
                    }.

  Inductive storage_get (d: depot): fqa_storage → storage → Prop :=
  | sg_apply: ∀ storage shard s c st,
      shard_exists s →
      MemoryBase.load_result depot_params s d shard →
      MemoryBase.load_result shard_params c shard storage →
      storage_get d (mk_fqa_storage s c) st .

  Inductive storage_read (d: depot): fqa_key → word → Prop :=
  | sr_apply: ∀ storage sk c w,
      storage_get d sk storage →
      storage_read d (mk_fqa_key sk c) w.

  Inductive storage_write (d: depot): fqa_key → word → depot → Prop :=
  | sw_apply: ∀ storage shard sk s c k w shard' depot' storage',
      storage_get d sk storage →
      MemoryBase.store_result storage_params k storage w storage' →
      MemoryBase.store_result shard_params c shard storage' shard' →
      MemoryBase.store_result depot_params s d shard' depot' →
      storage_write d (mk_fqa_key sk k) w depot'.

End Depot.
```

Library EraVM.memory.Pages

```
Require Common MemoryBase.
Import Common MemoryBase.

Section Pages.
```

Main memory (transient)

Memory structure

Contract execution routinely uses **main memory** to hold instructions, stack, heaps, and constants.

When the execution of a contract starts, new pages are allocated for:

- contract code: `code_page`, fetched by decommitter; see `Decommitter` and `FarCallDefinitions`);
- data stack: `stack_page`;
- heap: `data_page`;
- aux_heap: `data_page`;
- constants: `const_page`, implementation may chose to allocate code and constants on the same page.

Therefore, the types of pages are: data pages, stack pages, constant data pages, and code pages.

```
Context {code_page const_page data_page stack_page: Type}.

Inductive page_type :=
| DataPage : data_page → page_type
| StackPage : stack_page → page_type
| ConstPage : const_page → page_type
| CodePage : code_page → page_type
.

Record page := mk_page {
                   type:> page_type
                 }.
```

Memory is a collection of pages `memory`, each page is attributed a unique identifier `page_id`. Pages persist for as long as they can be read by some code; in presence of `FatPointer` the page lifetime may exceed the lifetime of the frame that created it.

```
Definition page_id := nat.
Definition pages := list (page_id × page).
Record memory := mk_pages {
```

```
                  mem_pages:> pages;
                }.

Inductive page_has_id : memory → page_id → page → Prop :=
| mpid_select : ∀ mm id page,
    List.In (id, page) mm →
    page_has_id (mk_pages mm) id page.
```

The set of identifiers has a complete linear order, ordering the pages by the time of creation. The ability to distinguish older pages from newer is necessary to prevent returning fat pointers to pages from older frames. See e.g. `step_RetExt_ForwardFatPointer`.

```
Section Order.
  Definition page_ordering := Nat.leb.
  Definition page_eq x y := page_ordering x y && page_ordering y x.
  Definition page_neq x y := negb (page_eq x y).
  Definition page_older (id1 id2: page_id) : bool :=
    page_ordering id1 id2.
End Order.
```

Predicate `page_replace` describes a relation between two memories `m1` and `m2`, where `m2` is a copy of `m1` but a page with it `id` is replaced by another page `p`.

```
  Inductive page_replace_list (id:page_id) (p:page): pages → pages → Prop :=
  | mm_replace_base: ∀ oldpage newpage tail,
      page_replace_list id p ((id, oldpage)::tail) ((id,newpage)::tail)
  | mm_replace_ind: ∀ oldpage not_id tail tail',
      page_replace_list id p tail tail' →
      id ≠ not_id →
      page_replace_list id p ((not_id,oldpage)::tail)
((not_id,oldpage)::tail').

  Inductive page_replace (id:page_id) (p:page): memory → memory → Prop :=
  | prl_apply: ∀ ls ls',
      page_replace_list id p ls ls' →
      page_replace id p (mk_pages ls) (mk_pages ls').
```

Function `page_alloc` creates a new page in memory.

```
  Definition page_alloc (p:page) (m: memory) : memory :=
    let new_id := length (mem_pages m) in
    match m with
    | mk_pages mem_pages ⇒ mk_pages (cons (new_id, p) mem_pages)
    end.

End Pages.
```

Library EraVM.memory.PageTypes

```
Require List Core MemoryBase.
Import Common Core MemoryBase List.

Section Memory.
```

Data pages

A **data page** contains individually addressable bytes. Each data page holds 2^{32} bytes.

```
  Definition data_page_params := {|
                                  addressable_block := u8;
                                  address_bits := 32;
                                  default_value := zero8;
                                  writable := true
                                |}.
  Definition mem_address := address data_page_params.
  Definition mem_address_zero: mem_address := zero32.
  Definition mem_address_bits := data_page_params.(address_bits).
  Definition data_page := mem_parameterized data_page_params.
```

There are currently two types of data pages:

- `Heap`
- `AuxHeap`.

  We call both of them **heap variants** for brevity, as in almost all cases they are handled in a similar way.

```
  Inductive data_page_type := Heap | AuxHeap.
  Definition page_bound := prod data_page_type mem_address.

  Local Open Scope ZMod_scope.
  Definition growth (current_bound: mem_address) (query_bound: mem_address)
    : mem_address :=
    if query_bound < current_bound then zero32 else
      snd (query_bound - current_bound)
     .
```

Note: only selected few instructions can access data pages:

- `OpLoad`/`OpLoadInc`
- `OpStore`/`OpStoreInc`
- `OpLoadPointer`/`OpLoadPointerInc`

Every byte on data pages has an address, but these instructions read or store 32-byte words.

Fat pointers `fat_ptr` define slices of data pages and allow passing them between contracts.

Stack pages

A **stack page** contains 2^{16} primitive values (see `primitive_value`). Reminder: primitive values are tagged words.

```
Context {primitive_value: Type} (pv0: primitive_value).
Definition stack_page_params := {|
                                 addressable_block := primitive_value;
                                 address_bits := 16;
                                 default_value := pv0;
                                 writable := true
                               |}.

Definition stack_address := address stack_page_params.
Definition stack_address_bits := stack_page_params.(address_bits).
Definition stack_address_zero: stack_address := zero16.

Definition stack_page := mem_parameterized stack_page_params.
```

Data stack in EraVM

There are two stacks in EraVM: call stack to support the execution of functions and contracts, and data stack to facilitate computations. This section details the data stack.

At each moment of execution, one stack page is active; it is associated with the topmost of external frames, which belongs to the contract currently being executed. See `active_extframe`, its field `ecf_mem_ctx` and subfield `ctx_stack_page_id`.

Each contract instance has an independent stack on a separate page. Instead of zero, stack pointer on new stack pages start with a value `INITIAL_SP_ON_FAR_CALL`. Stack addresses in range [0; `INITIAL_SP_ON_FAR_CALL`) can be used as a scratch space.

Topmost frame of the callstack, whether internal or external, contains the stack pointer (SP) value `cf_sp`; which points to the address after the topmost element of the stack. That means that the topmost element of the stack is located in word number $(SP-1)$ on the associated stack page.

Data stack grows towards greater addresses. In other words, pushing to stack increases stack pointer value, and popping elements decreases stack pointer.

Const pages

A **const page** contains 2^{16} non tagged `word`s. They are not writable.

Implementation may put constants and code on the same pages. In this case, the bytes on the same virtual page can be addressed in two ways:

- For instructions `OpJump` and `OpNearCall`, the code addressing will be used: consecutive addresses correspond to 8-bytes instructions.
- For instructions like `OpAdd` with `CodeAddr` addressing mode, the const data addressing will be used: consecutive addresses correspond to 32-bytes words.

For example, `OpJump 0`, `OpJump 1`, `OpJump 2`, `OpJump 3` will refer to the first four instructions on the page; their binary representations, put together, can be fetched by addressing the const page's 0-th word e.g. `OpAdd (CodeAddr R0 zero_16 ) (Reg R0) (Reg R1)`.

```
Definition const_address_bits := 16.

Definition const_page_params := {|
                                  addressable_block := word;
                                  address_bits := const_address_bits;
                                  default_value := word0;
                                  writable := false
                                |}.
Definition const_address := address const_page_params.
Definition const_page := mem_parameterized const_page_params.
```

Code pages

A **code page** contains 2^{16} instructions. They are not writable.

On genesis, code pages are filled as follows:

- The contract code is places starting from the address 0.
- The rest is filled with a value guaranteed to decode as `invalid` instruction.

Const pages can coincide with code pages.

```
  Context {ins_type: Type} (invalid_ins: ins_type).

  (* Implementation note: code_address should be address
code_page_params but we don't want to introduce
     dependency between code_address and instruction type *)
  Definition code_address_bits := 16.
  Definition code_address := BITS code_address_bits.
  Definition code_address_zero: stack_address := zero16.

  Definition code_page_params := {|
```

```
                                  addressable_block := ins_type;
                                  address_bits := code_address_bits;
                                  default_value := invalid_ins;
                                  writable := false
                                |}.
  Record code_page := mk_code_page {
                        cp_insns:> mem_parameterized code_page_params;
                      }.
  Definition code_length := code_address.

End Memory.
```

Library EraVM.Pointer

```
From RecordUpdate Require Import RecordSet.

Require memory.Pages memory.PageTypes.

Import Arith Core Common MemoryBase PageTypes Pages RecordSetNotations BinInt
zmodp.
Import ssreflect.tuple ssreflect.eqtype.

Section PointerDefinitions.
  Context (bytes_in_word := u32_of z_bytes_in_word).

  Open Scope ZMod_scope.
```

This library describes concepts related to `data_page`s: spans, heap pointers, and fat pointers.

Span

A **span** is a range of addresses from `s_start` inclusive to `s_start + s_length` exclusive. It is not bound to a specific page.

```
Record span :=
  mk_span {
      s_start : mem_address;
      s_length: mem_address;
    }.

Definition span_empty := mk_span zero32 zero32.

Inductive span_limit: span → mem_address → Prop :=
  | sl_apply:∀ start length limit,
    (false, limit) = start + length →
    span_limit (mk_span start length) limit.

Inductive bound_of_span : span → data_page_type → page_bound → Prop :=
  | qos_apply: ∀ s limit hv,
```

```
      span_limit s limit →
      bound_of_span s hv (hv, limit).
```

Usage

Passing a span in `ForwardNewHeapPointer` as an argument to far calls or far returns creates a **fat pointer**. The required encoding is described by `FarRetABI` or `FarCallABI`. See `FarCallDefinitions` and `FarRet`.

See also: `Slices`.

In EraVM, heap variants have a bound, stored in call stack frames (see `ctx_heap_bound` of `ecf_mem_ctx`). If memory is accessed beyond this bound, the bound is adjusted and the growth (difference between bounds) is paid in ergs. Definition `span_induced_growth` computes how many bytes should the user pay for.

```
Inductive span_induced_growth: span → mem_address → mem_address → Prop :=
| gb_bytes: ∀ start length query_bound current_bound diff,
    start + length = (false,query_bound) →
    diff = growth current_bound query_bound →
    span_induced_growth (mk_span start length) current_bound diff.

Section HeapPointer.
```

Heap pointer

A **heap pointer** is an absolute address `hp_addr` on some data page. Heap pointers are used in instructions:

- `OpLoad`/`OpLoadInc`
- `OpStore`/`OpStoreInc`

```
Record heap_ptr :=
  mk_hptr
    {
      hp_addr: mem_address;
    }.
Definition heap_ptr_empty : heap_ptr := mk_hptr zero32.

Inductive hp_resolves_to : heap_ptr → mem_address → Prop :=
| tpr_apply: ∀ addr,
    hp_resolves_to (mk_hptr addr) addr.
```

Incrementing a heap pointer with `hp_inc` increases its offset by 32, the size of a word in bytes. This is used by instructions `OpLoadInc` and `OpStoreInc`.

```
Inductive hp_inc : heap_ptr → heap_ptr → Prop :=
|fpi_apply :
  ∀ ofs ofs',
    ofs + bytes_in_word = (false,ofs') →
    hp_inc (mk_hptr ofs) (mk_hptr ofs' ).

Definition hp_inc_OF (hp: heap_ptr) : option heap_ptr :=
  match hp with
  | mk_hptr ofs ⇒
      match ofs + bytes_in_word with
      | (false, ofs') ⇒ Some (mk_hptr ofs')
      | _ ⇒ None
      end
  end.
```

Usage

Heap pointers are used by the following instructions:

- `OpLoad`/`OpLoadInc`
- `OpStore`/`OpStoreInc`

The layout of a heap pointer in a 256-bit word is described by `decode_heap_ptr`.

Relation to fat pointers

Heap pointers are bearing a similarity to fat pointers. They can be thought about as fat pointers where:

- the span starts at 0;
- an offset is the address;
- length is the limit, and
- page ID is ignored.

Heap pointers are not necessarily tagged and do not receive any special treatment from the VM's side.

```
End HeapPointer.
```

Definition `free_ptr` is auxiliary and is only used to facilitate the definition of `fat_ptr`.

```
Record free_ptr :=
  mk_ptr {
      p_span :> span;
      p_offset: mem_address;
    }.

Definition fresh_ptr (s:span) : free_ptr :=
  mk_ptr s zero32.

Definition validate_in_bounds (p:free_ptr) : bool := p.(p_offset) <
```

```
p.(s_length) .

  Inductive ptr_resolves_to : free_ptr → mem_address → Prop :=
  | upr_apply: ∀ addr start ofs length,
      (ofs < length) = true →
      (false, addr) = start + ofs →
      ptr_resolves_to (mk_ptr (mk_span start length) ofs) addr.

  Inductive word_upper_bound : heap_ptr → mem_address → Prop :=
  | qbu_apply : ∀ start upper_bound,
      let bytes_in_word := fromNat Core.bytes_in_word in
      start + bytes_in_word = (false,upper_bound) →
      word_upper_bound (mk_hptr start) upper_bound.

  Section FatPointer.
```

Fat pointer

A **fat pointer** defines an address inside a `span` of a specific data memory page `fp_page`. We may denote is as a 4-tuple: $(page, start, length, offset)$. These four components are enough to unambiguously identify a **`slice`** of a data memory page.

```
    Record fat_ptr :=
      mk_fat_ptr {
        fp_page: page_id;
        fp_ptr :> free_ptr;
      }.
```

Usage

- Fat pointers are used to pass read-only spans of `data_page`s between contracts.
- Fat pointers are created from spans by `OpFarCall`/`OpMimicCall`/`OpDelegateCall` or through `OpFarRet`/`OpFarRevert`.
  - Far calls accept a serialized instance of `FarCall.params` in a register. If it contains a span in `fwd_memory`, then when the contract starts executing, `r1` is assigned the fat pointer obtained from the slice.
  - Far returns accept a serialized instance of `FarRet.fwd_memory` in a register. If it contains a span in `fwd_memory`, then after return `r1` is assigned the fat pointer obtained from the slice.
- An existing fat pointer is passed by using `ForwardFatPointer`.

Pointers may be created only by far calls (`OpFarCall`, `OpMimicCall`, `OpDelegateCall`) or far returns (`OpFarRet`, `OpFarRevert`).

```
Record validation_exception :=
  mk_ptr_validation_exception
    {
      ptr_deref_beyond_heap_range : bool;
      ptr_bad_span: bool;
    }.

Definition no_exceptions : validation_exception
  := mk_ptr_validation_exception false false.

Inductive fat_ptr_nullable :=
| NullPtr
| NotNullPtr (ptr: fat_ptr)
.
```

A fat pointer $(page, start, length, offset)$ is **invalid** if:

- $start_{\mathbb{N}} + length_{\mathbb{N}} \geq 2^{32}$, or
- $offset > length$.

By a subscript $\mathbb{N}$ we denote that we explicitly interpret $start$ and $length$ as natural numbers, and addition on natural numbers does not overflow.

If $offset = length$, the span of the pointer is empty, but it is still valid.

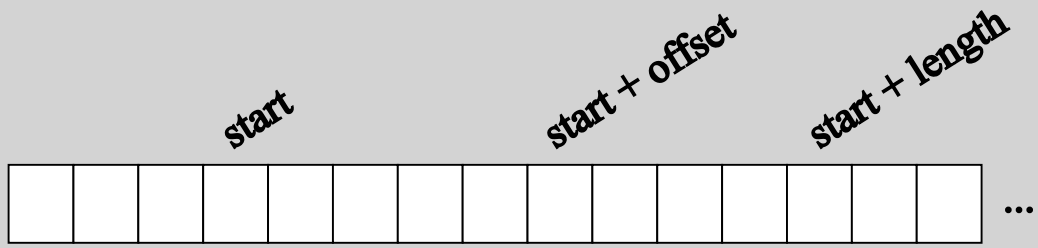
```
    Import ssrbool.
    Definition validate (p:free_ptr) : validation_exception :=
      match p with
      | mk_ptr (mk_span start len) ofs =>
          {|
            ptr_deref_beyond_heap_range := fst (start + len);
            ptr_bad_span := ofs > len ;
          |}
      end.

    Definition is_trivial (p:free_ptr) := (p.(s_length) == zero32) &&
(p.(p_offset) == zero32).
```

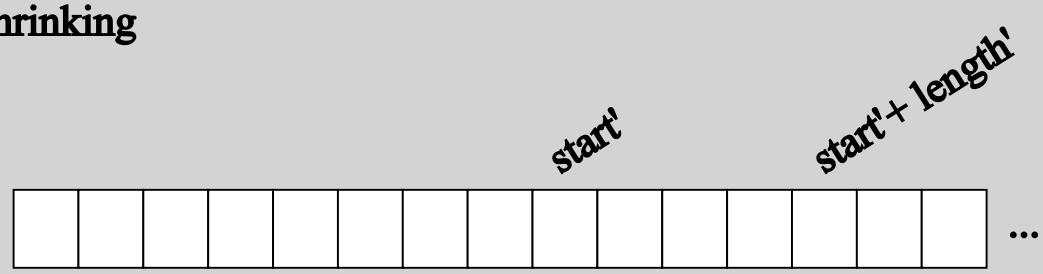
Narrowing a fat pointer moves its $start$ to where $start + offset$ was, and sets $offset$ to zero:

$$(page, start, length, offset) \mapsto (page, start + offset, length - offset, 0)$$

Before shrinking



After shrinking



A fat pointer is automatically narrowed in two situations:

- by far calls with forwarding mode `ForwardFatPointer`;
- by returns from far calls with forwarding mode `ForwardFatPointer`.

Attention: **shrinking** and **narrowing** far pointers are different. See `fat_ptr_shrink` and `fat_ptr_narrow`.

```
    Inductive free_ptr_narrow: free_ptr → free_ptr → Prop :=
    | tps_narrow: ∀ start start' length length' ofs,
        validate (mk_ptr (mk_span start length) ofs) = no_exceptions →
        start + ofs = (false, start') →
        length - ofs = (false, length')→
        free_ptr_narrow (mk_ptr (mk_span start length) ofs) (mk_ptr (mk_span
start' length') zero32).

    Definition fat_ptr_narrow := fat_ptr_liftP free_ptr_narrow.
```

Shrinking a fat pointer with `fat_ptr_shrink` subtracts a given number `diff` from its length; it is guaranteed to not overflow.

Shrinking may result in a pointer with $offset > length$, but such pointer will not resolve to a memory location.

```
    Inductive free_ptr_shrink (diff: mem_address) : free_ptr → free_ptr → Prop
```

```
:=
    | tptl_apply: ∀ start ofs length length',
        length - diff = (false, length') →
        free_ptr_shrink diff (mk_ptr (mk_span start length) ofs) (mk_ptr
(mk_span start length') ofs).

    Definition free_ptr_shrink_OF (diff: mem_address) : free_ptr → option
free_ptr :=
      fun fp ⇒ match fp with
              | mk_ptr (mk_span start length) p_offset ⇒
                  match length - diff with
                  | (false, length') ⇒ Some (mk_ptr (mk_span start length')
p_offset)
                  | _ ⇒ None
                  end
              end.

    Definition fat_ptr_shrink diff := fat_ptr_liftP (free_ptr_shrink diff).
    Definition fat_ptr_shrink_OF diff := fat_ptr_opt_map (free_ptr_shrink_OF
diff).
```

Incrementing a fat pointer with `fp_inc` increases its offset by 32, the size of a word in bytes. This is used by the instruction `OpLoadPointerInc`.

```
    Definition ptr_inc_OF (p: free_ptr) : option free_ptr :=
      match p with
      | mk_ptr s ofs ⇒
        match ofs + bytes_in_word with
        | (false, ofs') ⇒ Some (mk_ptr s ofs')
        | _ ⇒ None
        end
      end.

    Inductive ptr_inc : free_ptr → free_ptr → Prop :=
    |fpf_apply :
      ∀ ofs ofs' s,
        ofs + bytes_in_word = (false, ofs') →
        ptr_inc (mk_ptr s ofs) (mk_ptr s ofs').

    Definition fat_ptr_inc := fat_ptr_liftP ptr_inc.
    Definition fat_ptr_inc_OF := fat_ptr_opt_map ptr_inc_OF.
  End FatPointer.

End PointerDefinitions.
```

Library EraVM.Slice

```
From RecordUpdate Require Import RecordSet.

Require Pointer TransientMemory lib.PMap_ext.
```

```
Section Slices.
Import Bool Core Common MemoryBase RecordSetNotations Pointer TransientMemory
PMap_ext.
Open Scope ZMod_scope.
```

Slice

Data slice is a virtual memory page holding a read-only fragment of a `data_page`.

Accesses through a fat pointer should be in bounds of its span. However, loads by fat pointer return words, not individual bytes, so it is important to cut off parts of the memory page outside the pointer's span.

For example, suppose P is a fat pointer $(page, 0, 33, 10)$. Reading 32-byte `word` yields bytes from the offset 10-th to 42-th (excluded). However, the span of P is $[0, 33)$ so the bytes from 33-th to 42-th are outside of this span. EraVM treats the bytes outside P's span as if they were zeros.

More generally, suppose $P := (page, start, length, offset)$ is a fat pointer. Accesses through `OpLoadPtr` and `OpLoadPtrInc` return 32-byte words starting at an address $start + offset$.

However, a 32-byte `word` spans across addresses in range $[start + offset, start + offset + 32)$ and therefore can surpass the upper bound $start + length$ if $length - offset \leq 32$.

Reading past $start + offset$ yields zero bytes. In other words, attempting to read a word that spans across the pointer bound $start + offset$ will return zero bytes for the addresses $[start + length, start + offset + 32)$.

```
Definition data_page_slice_params := data_page_params <| writable := false |>.
Definition data_slice := mem_parameterized data_page_slice_params.

Definition do_slice_page (from_incl to_excl: mem_address) (m:data_page) :
data_slice :=
  let from_key := MemoryBase.addr_to_key _ from_incl in
  let to_key := MemoryBase.addr_to_key _ to_excl in
  let contents := pmap_slice m from_key to_key in
  mk_mem data_page_slice_params contents.
```

Predicate `slice_page` describes a slice visible to a fat pointer.

```
Inductive slice_page (m:data_page) : span → data_slice → Prop :=
| sfp_apply:
  ∀ start length upper_bound readonly_slice,
    start + length = (false, upper_bound) →
    do_slice_page start upper_bound m = readonly_slice →
    slice_page m (mk_span start length) readonly_slice.
```

```
End Slices.
```

Library EraVM.CallStack

```
From RecordUpdate Require Import RecordSet.
Import RecordSetNotations.
Require Common Ergs KernelMode memory.Depot TransientMemory MemoryContext.

Import Common Ergs KernelMode memory.Depot TransientMemory MemoryContext List
ListNotations.

Section Callstack.

  Context (CALLSTACK_LIMIT : nat).
  Context {state_checkpoint: Type} {ins: Type} (ins_invalid: ins).

  Definition exception_handler := code_address.
```

Call stack

EraVM operates with two stacks:

1. call stack, which supports function and contract execution
2. data stack, aiding in computations.

Data stack has a role similar to the stack in JVM and other language virtual machines. Call stack may be implemented in any way, but it is separated from data stack. This section covers the call stack in detail.

A **stack frame**, or **call frame**, is a data structure representing a fragment of current execution environment associated with a running instance of a contract or a function.

There are two types of stack frames:

- External frames `ExternalCall`: created on far calls (by instructions `OpFarCall`, `OpMimicCall`, `OpDelegateCall`).
- Internal frames `InternalCall`: created by near calls (by instruction `OpNearCall`).

Each external frame is **associated** with a contract address. It means that it was created when the associated contract address was far called. Naturally, each contract may be called many times recursively, therefore at each moment of time any contract may have multiple external frames associated to it.

`callstack` is a stack of a maximum of `CALLSTACK_LIMIT` stack frames. It is unrelated to the `stack_page` which holds data stack.

There is one callstack per execution, stored in `state` in `gs_callstack` field.

Internal call frames hold the following information:

- `cf_exception_handler_location`: a `code_address` of an exception handler. On revert or panic VM destroys the topmost frame and jumps to this handler.
- `cf_sp`: current **data stack** pointer. The topmost element in data stack is located at `cf_sp-1` in the currently `active_stackpage`.
- `cf_ergs_remaining`: ergs allocated for the current function or contract. It decreases as VM spends ergs for its operation.
- `cf_saved_checkpoint` : a snapshot of the state for a rollback. In case of panic or revert, the state of storage and event queues will be restored.

```
  Record callstack_common := mk_cf {
                               cf_exception_handler_location:
exception_handler;
                               cf_sp: stack_address;
                               cf_pc: code_address;
                               cf_ergs_remaining: ergs;
                               cf_saved_checkpoint: state_checkpoint;
                             }.
```

External call frames hold the same information as internal. Additionally, they hold:

- Three associated contract addresses:
    1. `ecf_this_address` : the stack frame was created when this contract was called.
    2. `ecf_msg_sender` : the stack frame was created when this contract invoked one of far call instructions.
    3. `ecf_code_address` : which contract owns the code associated with the stack frame. It is not always the same contract as `ecf_this_address`.
- `ecf_mem_ctx` : current `mem_ctx` holding ids of active stack, heap variants, code, const pages and bounds of data pages.
- boolean `ecf_is_static` : true if the code associated with this frame is being executed in static mode.
- `ecf_context_u128_value` : captured value of `gs_context_u128`. It represents a snapshot of the value of global register `gs_context_u128` in a moment of far call to the current contract.
- `ecf_shards` : shards associated with `ecf_this_address`, `ecf_msg_sender` and `ecf_code_address`.

```
Record active_shards := mk_shards {
                          shard_this: shard_id;
                          shard_caller: shard_id;
                          shard_code: shard_id;
                        }.


Record associated_contracts :=
  mk_assoc_contracts {
      ecf_this_address: contract_address;
      ecf_msg_sender: contract_address;
```

```
        ecf_code_address: contract_address;
      }.

  Record callstack_external :=
    mk_extcf {
        ecf_associated:> associated_contracts;
        ecf_mem_ctx: mem_ctx;
        ecf_is_static: bool; (* forbids any write-
like "log" and so state modifications, event emissions, etc *)
        ecf_context_u128_value: u128;
        ecf_shards:> active_shards;
        ecf_common :> callstack_common
      }.


  Inductive callstack :=
  | InternalCall (_: callstack_common) (tail: callstack): callstack
  | ExternalCall (_: callstack_external) (tail: option callstack): callstack.

  Fixpoint callstack_depth cf :=
    (match cf with
     | InternalCall x tail => 1 + callstack_depth tail
     | ExternalCall x (Some tail)=> 1 + callstack_depth tail
     | ExternalCall x None => 1
     end)%nat.
```

Operation

When the server starts forming a new block, it starts a new instance of VM to execute the code called `Bootloader`. Bootloader is a contract with an address `BOOTLOADER_SYSTEM_CONTRACT_ADDRESS`. To support its execution, a corresponding `ExternalCall` frame with is pushed to the call stack.

Handling each transaction requires executing `OpFarCall`, which pushes another call frame to the callstack.

As the transaction is executed, call stack changes as follows:

- `OpNearCall` pushes an `InternalCall` frame to callstack.
- `OpFarCall`, `OpDelegateCall`, or `OpMimicCall` push an `ExternalCall` frame to callstack.
- `OpNearRet`, `OpNearRetTo`, `OpNearRevert`, `OpNearRevertTo`, `OpPanic`, `OpFarRet`, `OpFarRevert` pop a frame from callstack.

Attempting to have more than `CALLSTACK_LIMIT` elements in callstack will force the VM into panic.

Panics are equivalent to executing `OpPanic`, so they pop up the topmost stack frame and pass the control to the exception handler, specified in the popped frame.

Executing any instruction I changes the topmost frame:

1. `cf_pc` is incremented, unless I is `OpJump` .
2. `cf_sp` may be modified if I affects the data stack pointer through addressing modes

RelSpPop or RelSpPop.

3. cf_ergs_remaining is decreased by the **total cost** of I. Total cost is a sum of base_cost and additional costs, described by the small step predicates like step_jump.

```
  Definition stack_overflow (xstack:callstack) : bool :=
    Nat.ltb CALLSTACK_LIMIT (callstack_depth xstack).

  Definition cfc (ef: callstack) : callstack_common :=
    match ef with
    | InternalCall x _ ⇒ x
    | ExternalCall x _ ⇒ x
    end.

  Definition cfc_map (f:callstack_common→callstack_common) (ef: callstack) :
callstack :=
    match ef with
    | InternalCall x tail ⇒ InternalCall (f x) tail
    | ExternalCall x tail ⇒ ExternalCall (x <| ecf_common ::= f |>) tail
    end.

  Section ErgsManagement.

    Open Scope ZMod_scope.

    Definition ergs_remaining (ef:callstack) : ergs := (cfc
ef).(cf_ergs_remaining).
    Definition ergs_map (f: ergs→ergs) (ef:callstack) : callstack
      := cfc_map (fun x ⇒ x <| cf_ergs_remaining ::= f |>) ef.
    Definition ergs_set newergs := ergs_map (fun _ ⇒ newergs).

    Inductive ergs_return: ergs → callstack → callstack → Prop :=
    | er_return: ∀ delta new_ergs ef ef',
        delta + ergs_remaining ef = (false, new_ergs) →
        ef' = ergs_set new_ergs ef →
        ergs_return delta ef ef'.

    Inductive ergs_return_caller_and_drop : callstack → callstack → Prop
      :=
    |erc_internal: ∀ caller new_caller cf,
        ergs_return (ergs_remaining (InternalCall cf caller)) caller
          new_caller →
        ergs_return_caller_and_drop (InternalCall cf caller) new_caller
    |erc_external: ∀ caller new_caller cf,
        ergs_return (ergs_remaining (ExternalCall cf (Some caller))) caller
          new_caller →
        ergs_return_caller_and_drop (ExternalCall cf (Some caller)) new_caller.

    Definition ergs_reset := ergs_set zero32.

    Definition affordable (ef: callstack) (e:ergs): bool :=
      match ergs_remaining ef - e with
      | (false, paid) ⇒ true
      | (true, overflowed) ⇒ false
```

```
    end.

  Inductive pay : ergs → callstack → callstack → Prop :=
  | pay_ergs : ∀ e ef paid,
      ergs_remaining ef - e = (false, paid) →
      pay e ef (ergs_set paid ef).
End ErgsManagement.

Section SP.
```

Fetching value of the stack pointer itself.

```
  Definition sp_get (cf: callstack) : stack_address :=
    (cfc cf).(cf_sp).

  Definition sp_map_extcall (f:stack_address→stack_address) ef :=
    (ef <| ecf_common ::= fun cf ⇒ cf <| cf_sp ::= f |> |>).

  Inductive sp_map_extcall_spec f: callstack_external → callstack_external →
Prop :=
  | sme_apply: ∀ a d e g h eh sp pc ss ergs,
      sp_map_extcall_spec f (mk_extcf a d e g h (mk_cf eh sp pc ergs ss))
        (mk_extcf a d e g h (mk_cf eh (f sp) pc ergs ss)).

  Theorem sp_map_extcall_correct:
    ∀ f ef, sp_map_extcall_spec f ef (sp_map_extcall f ef).

  Definition sp_map (f:stack_address→stack_address) ef : callstack :=
    match ef with
    | InternalCall x tail ⇒ InternalCall (x <| cf_sp ::= f |>) tail
    | ExternalCall x tail ⇒ ExternalCall (sp_map_extcall f x) tail
    end.

  Definition sp_update new_sp := sp_map (fun _ ⇒ new_sp).

  Inductive sp_map_spec f : callstack → callstack → Prop :=
  | usp_ext:
    ∀ ecf ecf' tail,
      sp_map_extcall_spec f ecf ecf' →
      sp_map_spec f (ExternalCall ecf tail) (ExternalCall ecf' tail)
  | usp_int:
    ∀ eh sp pc ergs tail ss,
      sp_map_spec f (InternalCall (mk_cf eh sp pc ergs ss ) tail)
(InternalCall (mk_cf eh (f sp) pc ergs ss) tail).

  Theorem sp_map_spec_correct f:
    ∀ ef, sp_map_spec f ef (sp_map f ef).

End SP.

Section PC.
  Definition pc_get (ef: callstack) : code_address :=
    match ef with
    | InternalCall cf _ ⇒ cf.(cf_pc)
    | ExternalCall ef tail ⇒ ef.(ecf_common).(cf_pc)
```

```
      end.

    Definition pc_map f ef :=
      match ef with
      | InternalCall x tail ⇒ InternalCall (x <| cf_pc ::= f |>) tail
      | ExternalCall x tail ⇒ ExternalCall (x <| ecf_common ::= fun cf ⇒ cf <|
cf_pc ::= f |> |>) tail
      end.

    Definition pc_set new := pc_map (fun _ ⇒ new).

    Inductive pc_map_cfc_spec f : callstack_common → callstack_common
                                → Prop :=
    | upc_map:
      ∀ ehl sp ergs pc pc' ss,
        f pc = pc' →
        pc_map_cfc_spec f (mk_cf ehl sp pc ergs ss) (mk_cf ehl sp pc' ergs ss).

    Inductive pc_map_extcall_spec f: callstack_external → callstack_external
                                   → Prop :=
    | upe_update:
      ∀ cf cf' ac memory is_static context_u128_value cc,
        pc_map_cfc_spec f cf cf' →
        pc_map_extcall_spec f
          (mk_extcf ac memory is_static
             context_u128_value cc cf)
          (mk_extcf ac memory is_static
             context_u128_value cc cf')
    .

    Inductive pc_map_spec f : callstack → callstack → Prop :=
    | upc_ext:
      ∀ ecf ecf' tail ,
        pc_map_extcall_spec f ecf ecf' →
        pc_map_spec f (ExternalCall ecf tail) (ExternalCall ecf' tail)
    | upc_int:
      ∀ cf cf' tail,
        pc_map_cfc_spec f cf cf' →
        pc_map_spec f (InternalCall cf tail) (InternalCall cf' tail).

    Theorem pc_map_correct:
      ∀ ef f, pc_map_spec f ef (pc_map f ef).

End PC.

Section TopmostExternalFrame.

    Fixpoint active_extframe (ef : callstack) : callstack_external :=
      match ef with
      | InternalCall _ tail ⇒ active_extframe tail
      | ExternalCall x tail ⇒ x
      end.

    Inductive active_extframe_spec : callstack → callstack_external → Prop :=
    | te_Top: ∀ x t, active_extframe_spec (ExternalCall x t) x
    | te_Deeper: ∀ c t f,
        active_extframe_spec t f → active_extframe_spec (InternalCall c t) f
```

```
.
Theorem active_extframe_correct:
  ∀ ef, active_extframe_spec ef (active_extframe ef).

Fixpoint change_active_extframe f (ef:callstack) : callstack :=
  match ef with
  | InternalCall x tail ⇒ InternalCall x (change_active_extframe f tail)
  | ExternalCall x tail ⇒ ExternalCall (f x) tail
  end.

Inductive change_active_extframe_spec f : callstack → callstack → Prop :=
| ct_base: ∀ cf t,
    change_active_extframe_spec f (ExternalCall cf t) (ExternalCall (f cf)
t)
| ct_ind: ∀ cf t t',
    change_active_extframe_spec f t t' →
    change_active_extframe_spec f (InternalCall cf t) (InternalCall cf t')
.

Lemma change_active_extframe_correct : ∀ f ef,
    change_active_extframe_spec f ef (change_active_extframe f ef).

Definition update_memory_context (ctx:mem_ctx): callstack → callstack :=
  change_active_extframe (fun ef ⇒ ef <| ecf_mem_ctx := ctx |> ).

Definition revert_state (cs:callstack) : state_checkpoint :=
  match cs with
  | InternalCall x tail ⇒ x.(cf_saved_checkpoint)
  | ExternalCall x tail ⇒ x.(ecf_common).(cf_saved_checkpoint)
  end .

  Definition current_shard xstack : shard_id := (active_extframe
xstack).(ecf_shards).(shard_this).

  Definition current_contract xstack : contract_address := ecf_this_address
(active_extframe xstack).

End TopmostExternalFrame.

Section ActiveMemory.

  Section ActivePageId.

    Context (ef:callstack) (active_extframe := active_extframe ef).

    Definition get_mem_ctx: mem_ctx := active_extframe.(ecf_mem_ctx).

    Definition active_code_id: page_id := get_mem_ctx.(ctx_code_page_id).

    Definition active_stack_id: page_id := get_mem_ctx.(ctx_stack_page_id).

    Definition active_const_id: page_id := get_mem_ctx.(ctx_const_page_id).

    Definition active_heap_id : page_id := get_mem_ctx.(ctx_heap_page_id).

    Definition active_auxheap_id : page_id :=
get_mem_ctx.(ctx_auxheap_page_id).
```

```
      Definition heap_bound := get_mem_ctx.(ctx_heap_bound).

      Definition auxheap_bound := get_mem_ctx.(ctx_auxheap_bound).

      Definition heap_variant_bound (page_type:data_page_type): mem_address :=
        match page_type with
        | Heap ⇒ heap_bound
        | AuxHeap ⇒ auxheap_bound
        end.

      Definition heap_variant_page_id (page_type: data_page_type)
        : page_id :=
        match page_type with
        | Heap ⇒ active_heap_id
        | AuxHeap ⇒ active_auxheap_id
        end.
    End ActivePageId.

    Section ActivePages.

      Context {code_page const_page data_page stack_page} (page_has_id: page_id
→ @page code_page const_page data_page stack_page → Prop).

      Definition active_exception_handler (ef: callstack) : exception_handler
:=
        (cfc ef).(cf_exception_handler_location).

      Context (ef: callstack) (page_id := fun i ⇒ page_has_id (i ef)).

      Inductive active_codepage : code_page → Prop :=
      | ap_active_code: ∀ codepage,
          page_id active_code_id (mk_page (CodePage codepage)) →
          active_codepage codepage.

      Inductive active_constpage : const_page → Prop :=
      | ap_active_const: ∀ constpage,
          page_id active_const_id (mk_page (ConstPage constpage)) →
          active_constpage constpage.

      Inductive active_stackpage : stack_page → Prop :=
      | ap_active_stack: ∀ stackpage,
          page_id active_stack_id (mk_page (StackPage stackpage)) →
          active_stackpage stackpage.

      Inductive active_heappage : data_page → Prop :=
      | ap_active_heap: ∀ p,
          page_id active_heap_id (mk_page (DataPage p)) →
          active_heappage p.

      Inductive active_auxheappage : data_page → Prop :=
      | ap_active_auxheap: ∀ p,
          page_id active_auxheap_id (mk_page (DataPage p)) →
          active_auxheappage p.
    End ActivePages.

End ActiveMemory.
```

```
  Definition in_kernel_mode (ef:callstack) : bool :=
    let ef := active_extframe ef in
    addr_is_kernel ef.(ecf_this_address).

End Callstack.
```

Library EraVM.MemoryContext

```
From RecordUpdate Require Import RecordSet.

Require memory.Pages memory.PageTypes.

Import memory.Pages memory.PageTypes.

Section MemoryContext.
  Import RecordSetNotations.
  Import seq Arith.

  Open Scope ZMod_scope.
```

Memory context

Creation of an external frame leads to allocation of pages for code, constant data, stack, and heap variants (see `alloc_pages_extframe`).

Memory context is a collection of pages associated with a contract's frame, plus the bounds for heap variants. It is stored in `ecf_mem_ctx` field of `ExternalCall` frame.

```
Record mem_ctx :=
  mk_mem_ctx
    {
      ctx_code_page_id: page_id;
      ctx_const_page_id: page_id;
      ctx_stack_page_id: page_id;
      ctx_heap_page_id: page_id;
      ctx_auxheap_page_id: page_id;
      ctx_heap_bound: mem_address;
      ctx_auxheap_bound: mem_address;
    }.
```

The exact values of identifiers of pages in `mem_ctx` are not guaranteed, neither is their order.

However, pages are ordered according to the order of creation (see `page_older`).

```
Definition list_mem_ctx (ap:mem_ctx) : list page_id :=
  match ap with
  | mk_mem_ctx code_id const_id stack_id heap_id auxheap_id _ _ ⇒
      [:: code_id; const_id;stack_id;heap_id;auxheap_id]
  end.

Definition page_older (id: page_id) (mps: mem_ctx) : bool :=
  List.forallb (page_older id) (list_mem_ctx mps).
```

Function `is_active_page` returns `true` if memory page `id` belongs to the context `c`.

```
Definition is_active_page (c:mem_ctx) (id: page_id) : bool :=
  List.existsb (page_eq id) (list_mem_ctx c).
```

If an instruction addresses a heap variant outside of its bounds, the bound of this heap variant is adjusted to include the used address. Predicates `grow_heap_page`, `grow_auxheap_page`, `grow_heap_variant` are relating memory contexts where a heap variant is grown.

WARNING: KNOWN DIVERGENCE (in versions prior to v1.4.1)

In earlier implementations, if heap/auxheap was grown inside a near call, the parent's heap/auxheap bound may be restored after `ret` as if no growth happened. Since v 1.4.1 the implementation conforms to the spec.

```
  Inductive grow_heap_page: mem_address → mem_ctx → mem_ctx → Prop :=
  | gp_heap: ∀ ap new_bound diff,
      ap.(ctx_heap_bound) + diff = (false,new_bound) →
      grow_heap_page diff ap (ap <| ctx_heap_bound := new_bound |>).

  Inductive grow_auxheap_page : mem_address → mem_ctx → mem_ctx → Prop :=
  | gp_auxheap: ∀ ap new_bound diff,
      ap.(ctx_auxheap_bound) + diff = (false,new_bound) →
      grow_auxheap_page diff ap (ap <| ctx_auxheap_bound := new_bound |>).

  Inductive grow_heap_variant: data_page_type → mem_address → mem_ctx → mem_ctx
→ Prop :=
  | ghv_heap: ∀ diff ap ap',
      grow_heap_page diff ap ap' →
      grow_heap_variant Heap diff ap ap'
  | ghv_auxheap: ∀ diff ap ap',
      grow_auxheap_page diff ap ap' →
      grow_heap_variant AuxHeap diff ap ap'.

End MemoryContext.
```

Library EraVM.PointerErasure

```
Require Common Pointer.
Import Common Pointer.
```

Pointer Erasure (since v1.4.1)

Recall that `fat_ptr` contain four fields:

```
Record fat_ptr_layout := mk_fat_ptr_layout {
                           length: u32;
                           start: u32;
                           page: u32;
                           offset: u32;
                         }.
```

In kernel mode, these fields are observable by all instructions, even by those that work with integers.

In user mode, `fat_ptr` are opaque: if an instruction expects an integer (with a cleared `is_ptr` tag), and it is provided a pointer instead (with a set `is_ptr` tag), the pointer value is downcast to the integer before the instruction is able to observe it.

The downcasting zeroes the fields `page` and `start`.

```
Section PointerErasure.
  Context (is_kernel_mode: bool).

  Definition span_erase (s:span) : span :=
    if is_kernel_mode then s else
      match s with
      | mk_span start len => mk_span # 0 len
      end
  .

  Definition free_ptr_erase (fp: free_ptr) : free_ptr :=
    if is_kernel_mode then fp else
      match fp with
      | mk_ptr s ofs => mk_ptr (span_erase s) ofs
      end
  .

  Definition fat_ptr_erase (fp:fat_ptr) : fat_ptr:=
    if is_kernel_mode then fp else
      match fp with
      | mk_fat_ptr page ptr => mk_fat_ptr 0%nat (free_ptr_erase ptr)
      end
  .

  Definition fat_ptr_nullable_erase (fp:fat_ptr_nullable) : fat_ptr_nullable :=
    if is_kernel_mode then fp else
```

```
      match fp with
      | NullPtr ⇒ NullPtr
      | NotNullPtr fp ⇒ NotNullPtr (fat_ptr_erase fp)
      end.

End PointerErasure.
```

Library EraVM.Binding

```
From RecordUpdate Require Import RecordSet.
Import RecordSetNotations.
Require
  ABI
    Addressing
    CallStack
    Core
    Coder
    GPR
    isa.CoreSet
    Pointer
    PointerErasure
    PrimitiveValue
    State.
Import
  ABI
    FatPointerABI
    Addressing
    Addressing.Coercions
    CallStack
    Coder
    Core
    GPR
    isa.CoreSet
    MemoryOps
    Pointer
    PointerErasure
    PrimitiveValue
    State
    Types.

Section OperandBinding.
```

Binding operands in core instructions

The instructions from `CoreInstructionSet` are parameterized with an instance of `descr`, specifying the exact types of their operands. This allows for `instruction` definition to be reused:

- the `instruction decoded` are instructions decoded from `Assembly.asm_instruction`. They contain descriptions of operand sources and destinations, e.g. register or various memory locations.
- the `instruction bound` are instructions that contain the values fetched, stored, and with decoding/encoding already performed. It allows to bind these values equationally inside the spec clauses such as `step_add`.

The rest of this section is technical; refer to `bind_fat_ptr` and similar predicates to see how the compound instruction arguments are encoded and decoded for small step relations, e.g. `step_ptradd` for `OpPtrAdd`.

```
  #[global]
    Canonical Structure bound: descr :=
  {|
    src_pv := @primitive_value Core.word;
    src_fat_ptr := option (@primitive_value (u128 × fat_ptr_nullable));
    src_heap_ptr := option (@primitive_value (u224 × heap_ptr));
    src_farcall_params := option (@primitive_value FarCallABI.params);
    src_nearcall_params := option (@primitive_value (u224 ×
NearCallABI.params));
    src_ret_params := option (@primitive_value FarRetABI.params);
    src_precompile_params := option (@primitive_value
PrecompileParametersABI.params);
    dest_pv := @primitive_value Core.word;
    dest_heap_ptr := option (@primitive_value (u224 × heap_ptr) );
    dest_fat_ptr := option (@primitive_value (u128 × fat_ptr_nullable) );
    dest_meta_params := option (@primitive_value (MetaParametersABI.params ));
  |}
  .
```

Knowing the call stack, memory pages and registers are enough to bind any value appearing in `CoreInstructionSet`; additionally, kernel mode affects it: pointer erasure only happens in user mode.

```
  Record binding_state := mk_bind_st {
                            bs_is_kernel_mode: bool;
                            bs_cs: State.callstack;
                            bs_regs: regs_state;
                            bs_mem: State.memory;
                          }.

  Inductive bind_any_src: binding_state → binding_state → in_any →
@primitive_value word → Prop :=
  | bind_any_src_apply: ∀ regs (mem:State.memory) (cs cs':State.callstack)
(op:in_any) (v:@primitive_value word) (is_k: bool),
      load _ regs cs mem op (cs', v) →
      bind_any_src (mk_bind_st is_k cs regs mem) (mk_bind_st is_k cs' regs mem)
op v.
```

To bind a src_pv type of instruction operand, load its value and apply the effects of RelSpPop addressing mode. Additionally, if the instruction expects an integer, but gets a pointer with a tag, the pointer erasure happens (since v.1.4.1).

```
  Inductive bind_src: binding_state → binding_state → in_any → @primitive_value
word → Prop :=
  | bind_src_int_apply: ∀ regs (mem:State.memory) (cs cs':State.callstack)
(op:in_any) (v:word) (is_k: bool),
      bind_any_src (mk_bind_st is_k cs regs mem) (mk_bind_st is_k cs' regs mem)
op (IntValue v) →
      bind_src (mk_bind_st is_k cs regs mem) (mk_bind_st is_k cs' regs mem) op
(IntValue v)

  | bind_src_ptr_apply: ∀ regs (mem:State.memory) (cs cs':State.callstack)
(op:in_any) (v v':word)
                          (decoded decoded_erased: fat_ptr_nullable)
                          (encoded_erased high128: u128)
                          (is_k: bool),
      bind_any_src (mk_bind_st is_k cs regs mem) (mk_bind_st is_k cs' regs mem)
op (PtrValue v) →

      Some (high128, decoded) = decode_fat_ptr_word v →
      decoded_erased = fat_ptr_nullable_erase is_k decoded →
      Some encoded_erased = encode_fat_ptr decoded →
      v' = high128 ## encoded_erased →
      bind_src (mk_bind_st is_k cs regs mem) (mk_bind_st is_k cs' regs mem) op
(IntValue v')
  .
```

To bind a dest_pv type of instruction operand, store its value and apply the effects of RelSpPush addressing mode.

```
  Inductive bind_dest: binding_state → binding_state → out_any →
primitive_value → Prop :=
  | bind_dest_apply: ∀ regs mem cs regs' mem' cs' op val is_k,
      store _ regs cs mem op val (regs', mem', cs') →
      bind_dest (mk_bind_st is_k cs regs mem) (mk_bind_st is_k cs' regs' mem')
op val.
```

To bind src_fat_ptr or any other compound value encoded in a binary form, bind both the encoded and decoded value.

```
  Inductive bind_fat_ptr: binding_state → binding_state → in_any → option
(@primitive_value (u128 × fat_ptr_nullable)) → Prop :=
  | bind_fat_ptr_apply : ∀ op v s s' decoded high128,
      bind_src s s' op v →
```

```
      Some (high128, decoded) = decode_fat_ptr_word v.(value) →
      bind_fat_ptr s s' op (Some (PtrValue (high128, decoded)))
  .

  Inductive bind_heap_ptr: binding_state → binding_state → in_any → option
(@primitive_value (u224 × heap_ptr)) → Prop :=
  | bind_heap_ptr_apply : ∀ op v s s' decoded,
      bind_src s s' op v →
      Some decoded = decode_heap_ptr v.(value) →
      bind_heap_ptr s s' op (Some (IntValue decoded)).

  Inductive bind_farcall_params: binding_state → binding_state → in_any →
option (@primitive_value FarCallABI.params) → Prop :=
  | bind_farcall_params_apply : ∀ op v s s' decoded tag,
      bind_src s s' op v →
      Some decoded = FarCallABI.coder.(decode) v.(value) →
      bind_farcall_params s s' op (Some (mk_pv tag (decoded)))
  .

  Inductive bind_nearcall_params: binding_state → binding_state → in_any →
option (@primitive_value (u224× NearCallABI.params)) → Prop :=
  | bind_nearcall_params_apply : ∀ op v s s' decoded tag,
      bind_src s s' op v →
      Some decoded = NearCallABI.decode_word v.(value) →
      bind_nearcall_params s s' op (Some (mk_pv tag decoded)).

  Inductive bind_farret_params: binding_state → binding_state → in_any → option
(@primitive_value FarRetABI.params) → Prop :=
  | bind_farret_params_apply : ∀ op v s s' decoded tag,
      bind_src s s' op v →
      Some decoded = FarRetABI.coder.(decode) v.(value) →
      bind_farret_params s s' op (Some (mk_pv tag decoded)).

  Inductive bind_precompile_params: binding_state → binding_state → in_any →
option (@primitive_value PrecompileParametersABI.params) → Prop :=
  | bind_precompile_params_apply : ∀ op v s s' decoded tag,
      bind_src s s' op v →
      Some decoded = PrecompileParametersABI.ABI.(decode) v.(value) →
      bind_precompile_params s s' op (Some (mk_pv tag decoded)).

  Inductive bind_dest_fat_ptr: binding_state → binding_state → out_any →
                               option (@primitive_value (u128 ×
fat_ptr_nullable) ) → Prop :=
  | bind_dest_fat_ptr_apply: ∀ s s' op encoded ptr (high128:u128),
      bind_dest s s' op (PtrValue encoded) →
      encode_fat_ptr_word high128 ptr = Some encoded →
      bind_dest_fat_ptr s s' op (Some (PtrValue (high128, ptr)))
  .

  Inductive bind_dest_heap_ptr: binding_state → binding_state → out_any →
option (@primitive_value (u224 × heap_ptr) ) → Prop :=
  | bind_dest_heap_ptr_apply: ∀ s s' op (encoded:word) high224 hptr,
      bind_dest s s' op (IntValue encoded) →
      encode_heap_ptr_word high224 hptr = Some encoded →
      bind_dest_heap_ptr s s' op (Some (IntValue (high224, hptr)))
  .
```

```
  Inductive bind_dest_meta_params: binding_state → binding_state → out_any →
option (@primitive_value (MetaParametersABI.params)) → Prop :=
  | bind_dest_meta_params_apply: ∀ s s' op encoded params,
      bind_dest s s' op (IntValue encoded) →
      MetaParametersABI.coder.(encode) params = Some encoded →
      bind_dest_meta_params s s' op (Some (IntValue params))
  .

  Definition bind_relation :=
    {|
      mf_src_pv := bind_src;
      mf_src_fat_ptr := bind_fat_ptr;
      mf_src_heap_ptr := bind_heap_ptr;
      mf_src_farcall_params := bind_farcall_params;
      mf_src_nearcall_params := bind_nearcall_params;
      mf_src_ret_params := bind_farret_params;
      mf_src_precompile_params := bind_precompile_params;
      mf_dest_pv := bind_dest;
      mf_dest_fat_ptr := bind_dest_fat_ptr;
      mf_dest_heap_ptr := bind_dest_heap_ptr;
      mf_dest_meta_params := bind_dest_meta_params;
    |}.

  #[local]
  Definition get_binding_state_ts (s: transient_state) : binding_state :=
    {|
      bs_is_kernel_mode := in_kernel_mode (gs_callstack s);
      bs_regs := gs_regs s;
      bs_mem := gs_pages s;
      bs_cs := gs_callstack s;
    |}.

  #[local]
  Definition get_binding_state (s: state) : binding_state :=
get_binding_state_ts s.

  Inductive relate_transient_states (P: binding_state → binding_state → Prop):
transient_state → transient_state → Prop :=
  | rts_apply:
    ∀ ts1 ts2,
      P (get_binding_state_ts ts1) (get_binding_state_ts ts2)→
      relate_transient_states P ts1 ts2.

  #[local]
  Definition merge_binding_transient_state: binding_state → transient_state →
transient_state :=
    fun bs s1 ⇒
      match bs with
      | mk_bind_st _ cs regs gmem ⇒ s1
                                    <| gs_regs := regs |>
                                    <| gs_pages := gmem |>
                                    <| gs_callstack := cs |>
      end.

  #[local]
  Definition merge_binding_state : state → binding_state → state :=
    fun s1 bs ⇒
```

```
      match bs with
      | mk_bind_st _ cs regs gmem ⇒ s1 <| gs_transient ::=
merge_binding_transient_state bs |>
      end.

  #[local]
  Definition bind_operands_binding_state (s1 s2: binding_state) :
    @instruction decoded → @instruction bound → Prop :=
    ins_srelate bind_relation s1 s2 .
```

The definition `bind_operands` relates two `transient_state`s before and after binding, and two instructions:

- i_1 is a decoded instruction obtained by applying `to_core` to `Assembly.asm_instruction`;
- i_2 is a bound instruction where fetching and storing values, as well as encoding and decoding are abstracted.

The values of operands in i_2 can be further bound by relations, see e.g. `step_add`.

```
  #[global]
  Definition bind_operands (s1 s2:transient_state) (i1: @instruction decoded)
(i2: @instruction bound) : Prop :=
    relate_transient_states (fun bs1 bs2 ⇒ bind_operands_binding_state bs1 bs2
i1 i2) s1 s2.

End OperandBinding.
```

Library EraVM.History

```
Require Common.

Section History.
  Import ssreflect ssrfun ssrbool eqtype ssreflect.tuple seq.
```

History

History is a data structure supporting appending elements of type `T` to it.

```
  Context (T:eqType).

  Definition history := seq T.

  Context (l:history).
```

`history` supports checking if an element is contained in it.

```
  Definition contains (elem:T): bool := if has (fun e => e == elem) l then true
else false.

End History.
```

Library EraVM.Event

```
From mathcomp Require ssreflect ssrfun ssrbool eqtype tuple zmodp.
Require Core memory.Depot TransientMemory.
Import Core memory.Depot TransientMemory.

Section Events.
  Import ssreflect ssreflect.tuple ssreflect.eqtype ssrbool.

  Context {contract_address precompile_params: eqType}.
```

Events

VM interfaces with two queues:

1. L1 `l1_msg` events (see `gs_l1_msgs`), emitted by `OpToL1Message`.
2. L2 `events` events (see `gs_events`), emitted by `OpEvent`.

These queues are subject to `rollback`s: in case of revert or panic, the events emitted during the function or contract execution are rolled back.

```
  Record event := {
      ev_shard_id: shard_id;
      ev_is_first: bool;
      ev_tx_number_in_block: tx_num;
      ev_address: contract_address;
      ev_key: word;
      ev_value: word;
    }.
```

equality on events

```
  Definition ev_eqn (x y:event) : bool :=
    match x,y with
    | Build_event ev_shard_id1 ev_is_first1 ev_tx_number_in_block1 ev_address1
ev_key1 ev_value1 ,
      Build_event ev_shard_id2 ev_is_first2 ev_tx_number_in_block2 ev_address2
ev_key2 ev_value2 =>
        (ev_shard_id1 == ev_shard_id2 ) &&
          (ev_is_first1 == ev_is_first2) &&
```

```
          (ev_tx_number_in_block1 == ev_tx_number_in_block2) &&
          (ev_address1 == ev_address2) &&
          (ev_key1 == ev_key2) &&
          (ev_value1 == ev_value2)
    end.

  Lemma ev_eqnP : Equality.axiom ev_eqn.
  Proof.
    move => [a b c d e g] [a' b' c' d' e' g'].
    simpl.
    Local Ltac orelse H := try rewrite! Bool.andb_false_r; try rewrite H;
constructor; injection; intros; subst; by rewrite eq_refl in H.
    destruct (a == a') eqn: H1; [move: (eqP H1) => -> | by orelse H1].
    destruct (b == b') eqn: H2; [move: (eqP H2) => -> | by orelse H2].
    destruct (c == c') eqn: H3; [move: (eqP H3) => -> | by orelse H3].
    destruct (d == d') eqn: H4; [move: (eqP H4) => -> | by orelse H4].
    destruct (e == e') eqn: H5; [move: (eqP H5) => -> | by orelse H5].
    destruct (g == g') eqn: H6; [move: (eqP H6) => -> | by orelse H6].
    by rewrite eq_refl; constructor.
  Qed.

  Canonical ev_eqMixin := EqMixin ev_eqnP.
  Canonical ev_eqType := Eval hnf in EqType _ ev_eqMixin.
  Definition l1_msg := event.

  Record precompile_query := {
      q_tx_number_in_block: tx_num;
      q_shard_id: shard_id;
      q_contract_address: contract_address;
      q_key: precompile_params
    }.
```

equality on precompile queries

```
  Definition pq_eqn (x y:precompile_query) : bool :=
    match x,y with
    | Build_precompile_query q_tx_number_in_block1 q_shard_id1
q_contract_address1 q_key1,
      Build_precompile_query q_tx_number_in_block2 q_shard_id2
q_contract_address2 q_key2 =>
        (q_tx_number_in_block1 == q_tx_number_in_block2) &&
          (q_shard_id1 == q_shard_id2) &&
          (q_contract_address1 == q_contract_address2)&&
          (q_key1 == q_key2)
    end .

  Lemma pq_eqnP : Equality.axiom pq_eqn.
  Proof.
    move => [a b c d] [a' b' c' d'] =>//=.
    destruct (a == a') eqn: H1; move: H1; last by constructor; injection;
intros; subst; move: H1; rewrite eq_refl.
    move /eqP => -> //=.
    destruct (b == b') eqn: H2; [| by rewrite H2; constructor; injection;
intros; subst; rewrite eq_refl in H2].
    destruct (c == c') eqn: H3; [| by rewrite ! Bool.andb_false_r; constructor;
injection; intros; subst; rewrite eq_refl in H3].
    destruct (d == d') eqn: H4; [| by rewrite ! Bool.andb_false_r; constructor;
injection; intros; subst; rewrite eq_refl in H4].
```

```
    move: H2 (eqP H2) => -> -> //=. constructor.
    by rewrite (eqP H3) (eqP H4).
  Qed.

  Canonical pq_eqMixin := EqMixin pq_eqnP.
  Canonical pq_eqType := Eval hnf in EqType _ pq_eqMixin.

  Inductive query :=
  | EventQuery : event -> query
  | L1MsgQuery : l1_msg -> query
  | PrecompileQuery : precompile_query -> query.
```

equality on queries

```
  Definition query_eqn (x y: query) : bool :=
    match x,y with
    | EventQuery x, EventQuery y => x == y
    | L1MsgQuery x, L1MsgQuery y => x == y
    | PrecompileQuery x, PrecompileQuery y => x == y
    | _,_ => false
    end.

  Lemma query_eqnP : Equality.axiom query_eqn.
  Proof.
    unfold query_eqn.
    move => x y.
    destruct x, y =>//; try destruct (_ == _) eqn: Heq; try (by done);
constructor; try move /eqP in Heq; by [subst|injection].
  Qed.

  Canonical query_eqMixin := EqMixin query_eqnP.
  Canonical query_eqType := Eval hnf in EqType _ query_eqMixin.
  (* todo: probably these structures can be redesigned *)
End Events.
```

Library EraVM.MemoryBase

```
From RecordUpdate Require Import RecordSet.
From Bits Require spec.

Require BinNums FMapPositive ZArith.
Require Common.

Section MemoryBase.
  Import ssreflect.
  Import BinNums ZArith spec FMapPositive.
```

Memory is modeled as a mapping from addresses (integers of `address_bits` bits) to values of type `addressable_block`. Unmapped addresses are mapped to the `default_value`.

```
  Record mem_descr := mk_mem_descr {
                        addressable_block: Type;
                        default_value: addressable_block;
```

```
    address_bits: nat;
    writable: bool;
  }.

  Context (mem_params: mem_descr)
    (default_value := mem_params.(default_value))
    (address_bits := mem_params.(address_bits))
    (addressable_block := mem_params.(addressable_block)).

  Axiom AX_ADDRESS_BITS_GT_0: address_bits ≠ 0.

  Definition address := BITS address_bits.

  Record mem_parameterized : Type := mk_mem {
                                       contents :> PositiveMap.t
addressable_block
                                     }.
```

We use a map from positive numbers implemented as a tree to store values in memory. However, address space starts at zero. Therefore, having an address A we map it to the key $K(A)$ as follows:

$$K(A) := A + 1$$

```
  Program Definition addr_to_key (addr: address): positive :=
    match address_bits with
    | 0 ⇒ fun _ ⇒ _
    | S bts ⇒ fun Heq ⇒ Z.to_pos ((@toZ bts addr) + 1)
    end (@eq_refl Z).
```

All memory addresses are initialized to the default value at memory genesis.

```
  Definition load (addr : address) (m : mem_parameterized) : addressable_block
:=
    match PositiveMap.find (addr_to_key addr) m.(contents) with
    | None ⇒ default_value
    | Some v ⇒ v
    end.

  Definition store (val:addressable_block) (addr : address) (m :
mem_parameterized) : mem_parameterized :=
    mk_mem (PositiveMap.add (addr_to_key addr) val m).
```

An empty memory.

```
  Definition empty := mk_mem (PositiveMap.empty addressable_block).
```

```
  Inductive load_result (addr: address) (m: mem_parameterized)
(v:addressable_block) : Prop :=
  | LoadResultOK: load addr m = v → load_result addr m v.

  Inductive store_result (addr: address) (m : mem_parameterized)
(v:addressable_block) (m'
      : mem_parameterized) : Prop :=
  | StoreResultOK:
    writable mem_params = true →
    store v addr m = m' → store_result addr m v m'.
```

Heap variants are byte-addressable, but reads and words operate on 256-bit `word`s. Multicell loads return `len` consecutive bytes from memory `m` at an address `a`.

```
  Import Arith.
  Open Scope bits_scope.
  Fixpoint load_multicell (a:address) (len:nat) (m:mem_parameterized)
    : option (list addressable_block) :=
    match len with
    | O ⇒ Some nil
    | S lft ⇒ let value := load a m in
              let (overflow, nextaddr) := uadd_of a (fromZ 1) in
              if overflow then None else
                match load_multicell nextaddr lft m with
                | Some tail ⇒ Some (cons value tail)
                | None ⇒ None
                end
  end.

Inductive load_multicell_result:
  address → ∀ len: nat, mem_parameterized → list addressable_block → Prop :=
| lmr_end : ∀ a m,
    load_multicell_result a 0%nat m nil

| lmr_progress: ∀ addr nextaddr mem value n tail,
    (false, nextaddr) = uinc_of addr →
    load_result addr mem value →
    load_multicell_result nextaddr n mem tail →
    load_multicell_result addr (S n) mem (cons value tail)

.

Theorem load_multicell_spec:
  ∀ a len m ls,
    load_multicell a len m = Some ls →
    load_multicell_result a len m ls.

Theorem load_multicell_result_size:
  ∀ tl a m v,
    load_multicell_result a tl m v →
    seq.size v = tl.
```

Similarly, `store_multicell` accepts a list of values and puts them in memory starting from the address `a`.

```
  Import seq.
  Fixpoint store_multicell (a:address) (vals: list addressable_block)
(m:mem_parameterized)
    : option mem_parameterized :=
    if writable mem_params then
      match vals with
      | [::] ⇒ Some m
      | v :: tail ⇒
          let stored := store v a m in
          let (overflow, nextaddr) := uinc_of a in
          if overflow then None else
            store_multicell nextaddr tail stored
      end
    else None.

  Inductive store_multicell_result:
    address → list addressable_block → mem_parameterized → mem_parameterized →
Prop :=
  | smr_end : ∀ a m,
      writable mem_params = true →
      store_multicell_result a [::] m m

  | smr_progress: ∀ addr nextaddr mem mem' mem'' value tail,
      writable mem_params = true →
      (false, nextaddr) = uinc_of addr →
      store_result addr mem value mem' →
      store_multicell_result nextaddr tail mem' mem'' →
      store_multicell_result addr (value::tail) mem mem''
  .

  Theorem store_multicell_spec:
    ∀ ls a m m',
      store_multicell a ls m = Some m' →
      store_multicell_result a ls m m'.

End MemoryBase.
```

Library EraVM.MemoryOps

```
Require Addressing Core Common List Pointer TransientMemory Resolution Slice.
Import ssreflect eqtype.
Import Addressing Core ZArith Common CallStack GPR MemoryBase PrimitiveValue
Pointer Resolution Slice TransientMemory.
Section MemoryOps.
```

Data loading and storing

This section formalizes reading from memory or registers (`fetch`) and writing to memory or registers (`store`).

```
  Import Addressing.Coercions.

  Context {instruction: Type} (instruction_invalid:instruction)
    {state_checkpoint: Type}
    (callstack:= @callstack state_checkpoint)
  .

  Section FetchStore.
    Context (memory := @memory (@code_page instruction instruction_invalid)
const_page data_page stack_page)
    .
    Context (regs: regs_state) (cs: callstack) (mem: memory) .

    Inductive fetch_result : Type :=
    | FetchIns (ins: instruction)
    | FetchPV (pv: @primitive_value word).

    Inductive fetch: loc → fetch_result → Prop :=
    | fetch_reg: ∀ name reg_val,
        reg_val = fetch_gpr regs name →
        fetch (LocReg name) (FetchPV reg_val)
    | fetch_imm: ∀ imm imm',
        fetch (LocImm imm) (FetchPV (IntValue imm'))
    | fetch_stackaddr:
      ∀ stackpage (value: primitive_value) addr,
        active_stackpage (page_has_id mem) cs stackpage →
        MemoryBase.load_result _ addr stackpage value →
        fetch (LocStackAddress addr) (FetchPV value)
    | fetch_codeaddr:
      ∀ codepage addr ins,
        active_codepage (page_has_id mem) cs codepage →
        load_result _ addr codepage ins →
        fetch (LocCodeAddr addr) (FetchIns ins)
    | fetch_constaddr:
      ∀ constpage addr value,
        active_constpage (page_has_id mem) cs constpage →
        load_result _ addr constpage value →
        fetch (LocConstAddr addr) (FetchPV (IntValue value))
    .

    Definition next_ins := LocCodeAddr (pc_get cs).

    Definition fetch_instr := fetch next_ins.

    Inductive store_loc: primitive_value → loc → (regs_state × memory) → Prop
:=
```

```
| store_lreg:
  ∀ new_regs reg_name value mem,
    store_gpr regs reg_name value = new_regs →
    store_loc value (LocReg reg_name) (new_regs, mem)
| store_lstackaddr:
  ∀ new_mem stackpage addr value pid new_stackpage,
    active_stackpage (page_has_id mem) cs stackpage →
    store_result _ addr stackpage value new_stackpage →
    page_replace pid (StackPage new_stackpage) mem new_mem →
    store_loc value (LocStackAddress addr) (regs, new_mem)
.
```

Loading a primitive value from registers or memory, applying effects of `RelSpPop` if necessary.

```
Inductive load: in_any → callstack × primitive_value → Prop :=
| ld_apply : ∀ (arg:in_any) loc res new_cs,
    resolve_apply regs cs arg (new_cs, loc) →
    fetch loc (FetchPV res) →
    load arg (new_cs, res).
```

A special version of `load` for registers because it has less potential effects on the state, which makes it easier and more precise.

```
Inductive load_reg: in_reg → primitive_value → Prop :=
| ldr_apply : ∀ loc res,
    fetch_gpr regs loc = res →
    load_reg (Reg loc) res.

Definition load_int a cs w := load a (cs, IntValue w).
Definition load_reg_int a w := load_reg a (IntValue w).
```

Storing a primitive value to registers or memory, applying effects of `RelSpPush` if necessary.

```
Inductive store: out_any → primitive_value → regs_state × memory ×
callstack → Prop :=
| st_apply: ∀ (arg:out_any) loc new_regs new_mem new_cs pv,
    resolve_apply regs cs arg (new_cs, loc) →
    store_loc pv loc (new_regs, new_mem)→
    store arg pv (new_regs, new_mem, new_cs).
```

A special version of `store` for registers because it has less potential effects on the state, which makes it easier and more precise.

```
Inductive store_reg: out_reg → primitive_value → regs_state → Prop :=
| sr_apply: ∀ (arg:out_reg) loc new_regs pv,
```

```
        store_gpr regs loc pv = new_regs →
        store_reg arg pv new_regs.

    Definition store_int a w rs m cs := store a (IntValue w) (rs, m, cs).

  End FetchStore.

  Inductive loads (regs:regs_state) (cs:callstack) (mem:memory) : list (in_any
× primitive_value) → callstack → Prop :=
  | rsl_nil:
    loads regs cs mem nil cs

  | rsl_cons: ∀ (arg:in_any) pv (tail: list (in_any × primitive_value)) cs1
cs2,
      load regs cs mem arg (cs1, pv) →
      loads regs cs1 mem tail cs2 →
      loads regs cs mem ((arg,pv)::tail) cs2.

  Inductive load_regs (regs:regs_state) : list (in_reg × primitive_value) →
Prop :=
  | rslr_nil:
    load_regs regs nil

  | rslr_cons: ∀ (arg:in_reg) pv (tail: list (in_reg × primitive_value)),
      load_reg regs arg pv →
      load_regs regs tail →
      load_regs regs ((arg,pv)::tail).

  Inductive stores (regs:regs_state) (cs:callstack) (mem:memory) : list
(out_any × primitive_value) → (regs_state × memory × callstack) → Prop :=
  | rss_nil:
    stores regs cs mem nil (regs, mem, cs)
  | rss_cons: ∀ (arg:out_any) pv (tail: list (out_any × primitive_value)) regs1
mem1 cs1 regs2 mem2 cs2,
      store regs cs mem arg pv (regs1, mem1, cs1) →
      stores regs1 cs1 mem1 tail (regs2, mem2, cs2) →
      stores regs cs mem ((arg,pv)::tail) (regs2, mem2, cs2).

  Inductive store_regs (regs:regs_state) : list (out_reg × primitive_value) →
regs_state → Prop :=
  | rssr_nil:
    store_regs regs nil regs
  | rssr_cons: ∀ (arg:out_reg) pv (tail: list (out_reg × primitive_value))
regs1 regs2 ,
      store_reg regs arg pv regs1 →
      store_regs regs1 tail regs2 →
      store_regs regs ((arg,pv)::tail) regs2.

End MemoryOps.

Section Multibyte.
```

Multibyte loads and stores

Instructions such as `OpStore` and `OpLoadPointer` operate with a byte addressable `data_page`s, but load or store 256-bit `word`s. Therefore, their effects are formalized separately.

```
Inductive endianness := LittleEndian | BigEndian.

Context (e:endianness) (mem:data_page).
Definition mb_load_word (addr:mem_address) :option word.
Defined.

Inductive mb_load_result : mem_address → word → Prop :=
| mldr_apply: ∀ (addr:mem_address) res,
    mb_load_word addr = Some res →
    mb_load_result addr res.

Definition mb_store_word (addr:mem_address) (val: word) : option data_page :=
  let ls := match e with
            | LittleEndian ⇒ word_to_bytes val
            | BigEndian ⇒tuple.rev_tuple (word_to_bytes val)
            end in
  store_multicell _ addr (tuple.tval ls) mem.

  Inductive mb_store_word_result: mem_address → word → data_page → Prop :=
  | sdr_apply :
    ∀ addr val page',
      mb_store_word addr val = Some page' →
      mb_store_word_result addr val page'.
```

Reading from memory slices

Reading from `slice` is particular in the following way: if the accessed word passes over bounds, the bytes below the bound are formally assigned zeros. See `Slices`.

```
  Definition mb_load_slice_word (slc:data_slice) (addr:mem_address) :option
word.
  Defined.

  Inductive mb_load_slice_result (slc:data_slice): mem_address → word → Prop :=
  | mlsr_apply: ∀ (addr:mem_address) res,
      mb_load_slice_word slc addr = Some res →
      mb_load_slice_result slc addr res.

End Multibyte.
```

Library EraVM.MemoryManagement

```
Require CallStack Ergs TransientMemory Pointer.

Import CallStack Common Core TransientMemory MemoryContext Ergs Pointer.

Section MemoryForwarding.
  Open Scope ZMod_scope.

  Context {state_checkpoint: Type} (callstack:=@callstack state_checkpoint).
```

Memory forwarding

Contracts communicate by passing each other `fat_ptr`. Far returns and far calls are able to:

- create them `ForwardNewFatPointer`
- reuse existing pointers `ForwardExistingFatPointer`.

They chose the action based on an instance of `fwd_memory` passed through ABIs.

```
Inductive fwd_memory :=
  ForwardExistingFatPointer (p:fat_ptr_nullable)
| ForwardNewFatPointer (heap_var: data_page_type) (s:span).
```

- A fat pointer defines a `slice` of memory and provides a read-only access to it.
- Fat pointers are created from a slice of heap or auxheap of a current contract.
- If the `span` of a new fat pointer crosses the heap boundary `heap_variant_bound` then the heap has to grow, and that difference has to be paid for.
- `growth_query` defines by how much a a heap variant should be grown.

```
Definition growth_query := @option (prod data_page_type mem_address).

Definition cost_of_growth (diff:growth_query) : ergs :=
  match diff with
  | Some (_, x) => Ergs.growth_cost x
  | None => zero32
  end.

Inductive growth_to_bound: page_bound → callstack → growth_query → Prop :=
| goq_grow: ∀ (hv:data_page_type) (cs: callstack) diff query,
    let current_bound : mem_address := heap_variant_bound cs hv in
    query - current_bound = (false, diff) →
    growth_to_bound (hv, query) cs (Some (hv, diff))
| goq_nogrow: ∀ hv (cs: callstack) __ query,
```

```
      let current_bound := heap_variant_bound cs hv in
      query - current_bound = (true, __) →
      growth_to_bound (hv, query) cs None
  .

  Inductive grow: growth_query → callstack → callstack → Prop :=
  | gr_grow: ∀ hv cs1 cs2 diff new_apages,
      let apages := get_mem_ctx cs1 in
      grow_heap_variant hv diff apages new_apages →
      cs2 = update_memory_context new_apages cs1 →
      grow (Some (hv, diff)) cs1 cs2
  | gr_nogrow: ∀ cs,
      grow None cs cs.

  Inductive bound_grow_pay: page_bound → callstack → callstack → Prop :=
  | bgp_apply: ∀ bound query cs1 cs2 cs3,
      growth_to_bound bound cs1 query →
      pay (cost_of_growth query) cs1 cs2 →
      grow query cs2 cs3 →
      bound_grow_pay bound cs1 cs3.

  Inductive span_grow_pay: span → data_page_type → callstack → callstack → Prop
:=
  | sgp_apply: ∀ s hv cs1 cs2 bound,
      bound_of_span s hv bound →
      bound_grow_pay bound cs1 cs2 →
      span_grow_pay s hv cs1 cs2.

  Inductive paid_forward_new_fat_ptr: data_page_type → span → callstack →
fat_ptr × callstack → Prop :=
  | pfnfp_apply: ∀ s heap_id type cs0 cs1 ,
      span_grow_pay s type cs0 cs1 →
      heap_id = heap_variant_page_id cs0 type →
      paid_forward_new_fat_ptr type s cs0 (mk_fat_ptr (Some heap_id) (fresh_ptr
s), cs1).

  Inductive growth_to_bound_unaffordable (cs:callstack) bound : Prop :=
    | gtb_apply: ∀ query,
        growth_to_bound bound cs query →
        false = affordable cs (cost_of_growth query) →
        growth_to_bound_unaffordable cs bound.

  Inductive growth_to_span_unaffordable (cs:callstack) heap_type hspan : Prop
:=
    | gts_apply: ∀ bound,
        bound_of_span hspan heap_type bound →
        growth_to_bound_unaffordable cs bound →
        growth_to_span_unaffordable cs heap_type hspan.
End MemoryForwarding.
```

Library EraVM.Addressing

```
Require Common TransientMemory GPR.

Import Common TransientMemory GPR.
```

```
Section Addressing.
```

Addressing modes

This section describes the addressing modes in `asm_instruction`. Section `InstructionArguments` describes the types of the instruction arguments; each type corresponds to one or multiple possible addressing modes. Assembly instruction formats with the types of their arguments are described by `asm_instruction`.

Core instructions `instruction decoded` have different types of operands detailed by `instruction` and `decoded`.

Operands are entities operated upon by instructions. They serve as sources of data, or as destinations for the results of the instruction execution.

Addressing mode refers to the way in which an instruction specifies the location of data that needs to be accessed or operated upon.

Abstract EraVM supports 8 addressing modes. Some of them only support reading (indicated by “in”), or writing (indicated by “out”).

1. Register (in/out).

   Concrete syntax example. Use `r1` as a source:

   ```
   add r1, r0, r3
   ```

2. Imm (in)

   Concrete syntax example. Use immediate `42` as a source:

   ```
   add 42, r0, r3
   ```

3. Code page, relative to GPR (in)

   Concrete syntax example. Use 42-th `word` on the code page as a source:

   ```
   add code[42], r0, r3
   ```

   Note: words are enumerated starting at 0, each word contains 4 instructions, adjacent words are disjoint.

4. Const page, relative to GPR (in)

   Currently, the concrete syntax is absent because code and constant pages coincide in current EraVM implementation. Use the following instead:

   ```
   add code[42], r0, r3
   ```

5. Stack page, relative to GPR (in/out)

   Concrete syntax example. Use `(r1+42)`-th `word` on the stack page as a source has two equivalent forms:

```
add stack[r1+42], r0, r3
add stack=[r1+42], r0, r3
```

6. Stack page, relative to GPR and SP (in/out)

   Concrete syntax example. Use `(r1+42)`-th `word` on the stack page as a source:

   ```
   add stack-[r1+42], r0, r3
   ```

   Note, that the following form is forbidden:

   ```
   add stack-[r1+42], r0, r3
   ```

7. Stack page, relative to GPR and SP, with decreasing SP (in)

   Concrete syntax example. Use `SP-(r1+42)`-th `word` on the stack page as a source:

   ```
   add stack-=[r1+42], r0, r3
   ```

   Note, that the following form is forbidden:

   ```
   add stack+=[r1+42], r0, r3
   ```

8. Stack page, relative to GPR and SP, with increasing SP (out)

   Concrete syntax example. Use `SP+(r1+42)`-th `word` on the stack page as a destination:

   ```
   add r3, r0, stack+=[r1+42]
   ```

Note that the current implementation encodes some of these modes in the same way e.g. mode 7 and mode 8 only differ by *in* or *out* position.

Predicate `resolve` formalizes resolving operands to immediate values, registers and memory locations.

`MemoryOps` formalizes reading and writing to locations.

1. **Register addressing** (in/out)
   - Refers to one of General Purpose Registers (GPR).
   - *Concrete syntax example*. Use `r1` as a source:

     ```
     add r1, r0, r3
     ```

```
Inductive reg_io : Type := Reg (reg:reg_name).
```

2. **Immediate 16-bit value** (in)
   - *Concrete syntax example*. Use immediate `42` as a source:

```
        add 42, r0, r3
```

```
Inductive imm_in : Type := Imm (imm: u16).
```

3. **Address on a code page, relative to a GPR** (in)
   - Resolved to $(reg+imm) \mod 2^{16}$. See `rslv_code`.
   - Code and const pages may coincide in the implementation.
   - *Concrete syntax example*. Use 42-th `word` on the code page as a source:

     ```
     add code[42], r0, r3
     ```

     Note: words are enumerated starting at 0, each word contains 4 instructions, adjacent words are disjoint.

```
Inductive code_in : Type := CodeAddr (reg:reg_name) (imm:code_address).
```

4. **Address on a const page, relative to a GPR** (in)
   - Resolved to $(reg+imm) \mod 2^{16}$. See `rslv_const`.
   - Code and const pages may coincide.
   - Currently, the concrete syntax is absent because code and constant pages coincide in current EraVM implementation. Use the following instead:

     ```
     add code[42], r0, r3
     ```

```
Inductive const_in: Type := ConstAddr (reg:reg_name) (imm:code_address).
```

5. **Address on a stack page, relative to a GPR** (in/out)
   - Resolved to $(reg+imm) \mod 2^{16}$. See `rslv_stack_abs`.
   - *Concrete syntax example*. Use `(r1+42)`-th `word` on the stack page as a source has two equivalent forms:

     ```
     add stack[r1+42], r0, r3
     add stack=[r1+42], r0, r3
     ```

```
Inductive stack_io : Type :=
```

```
| Absolute (reg:reg_name) (imm: stack_address)
```

6. **Address on a stack page, relative to SP and GPR**
   - Resolved to $(SP-(reg+imm)) \mod 2^{16}$. See `rslv_stack_rel`.
   - Unlike `RelSpPop`, the direction of offset does not change depending on read/write.
   - *Concrete syntax example*. Use `(r1+42)`-th `word` on the stack page as a source:

     ```
     add stack-[r1+42], r0, r3
     ```

     Note, that the following form is forbidden:

     ```
     add stack[r1+42], r0, r3
     ```

```
| RelSP (reg:reg_name) (offset: stack_address)
.
```

7. **Stack page, relative to GPR and SP, accompanied by decreasing SP (in).**
   - A generalized version of `pop` operation.
   - Resolved to $(SP+(reg+imm)) \mod 2^{16}$. See `rslv_stack_rel`.
   - Additionally, after the resolution, SP is modified: `SP -= (reg + imm)`.
   - *Concrete syntax example*. Use `SP-(r1+42)`-th `word` on the stack page as a source:

     ```
     add stack-=[r1+42], r0, r3
     ```

     Note, that the following form is forbidden:

     ```
     add stack+=[r1+42], r0, r3
     ```

```
Inductive stack_in_only : Type :=
| RelSpPop (reg:reg_name) (offset: stack_address)
.
```

8. **Stack page, relative to GPR and SP, accompanied by increasing SP (out).**
   - A generalized version of `push` operation.
   - Resolved to $(SP+(reg+imm)) \mod 2^{16}$.

- Additionally, after the resolution, SP is modified: `SP += (reg + imm)`.
- *Concrete syntax example*. Use `SP+(r1+42)`-th `word` on the stack page as a destination:

  ```
  add r3, r0, stack+=[r1+42]
  ```

WARNING: KNOWN DIVERGENCE (in versions prior to v1.4.1) implementation of earlier versions diverged from the described spec:

- Implementation: the write happens to the new SP address
- Specification: the write happens to the old SP address

Since v 1.4.1 the implementation conforms to the spec.

```
Inductive stack_out_only : Type :=
| RelSpPush (reg:reg_name) (offset: stack_address)
.

Section InstructionArguments.
```

Operand types

This section details the types of operand for `asm_instruction`. The types of operands for `instruction` are different and detailed by `instruction`, `descr`, `decoded` and `bound`.

Instruction may have *input* and *output* operands.

- There are three types of input operands:
  - `in_reg` : read from a GPR.
  - `in_any` : read from reg, immediate value, or any memory. May be a generalized pop `RelSpPop`.
  - `in_regimm` read from either reg or immmediate value.
- There are two types of input operands:
  - `out_reg` : store to a GPR.
  - `out_any` : store to a GPR or any writable memory location. May be a generalized push `RelSpPush`.

To describe these types, we create a hierarchy of subtypes ordered by inclusion (see `Coercions`).

We denote input arguments as in_1, in_2, and output arguments as out_1, out_2. Many instructions have 2 input arguments and 1 output argument. The encoding limits the number of arguments of type `in_any` and `out_any`:

- For each instruction, there can be no more one argument of type `in_any`.
- For each instruction, there can be no more one argument of type `out_any`.

It is allowed to have both `in_any` and `out_any` in the same instruction.

```
Inductive stack_in : Type :=
| StackInOnly (arg: stack_in_only)
| StackInAny (arg: stack_io)
.

Inductive stack_out: Type :=
| StackOutOnly (arg: stack_out_only)
| StackOutAny (arg: stack_io)
.

Inductive stack_any : Type :=
| StackAnyIO (arg: stack_io)
| StackAnyIn (arg: stack_in_only)
| StackAnyOut (arg: stack_out_only)
.
```

Utility conversions, click to unfold

```
Definition stack_in_to_any (s:stack_in) : stack_any :=
  match s with
  | StackInOnly arg ⇒ StackAnyIn arg
  | StackInAny arg ⇒ StackAnyIO arg
  end.

Definition stack_out_to_any (s:stack_out) : stack_any :=
  match s with
  | StackOutOnly arg ⇒ StackAnyOut arg
  | StackOutAny arg ⇒ StackAnyIO arg
  end.
```

The `any` auxiliary argument type allows for all addressing modes; it never occurs in instructions but is used to `resolve` argument locations.

```
Inductive any : Type :=
| AnyReg : reg_io → any
| AnyImm : imm_in → any
| AnyStack: stack_any→ any
| AnyCode : code_in → any
| AnyConst: const_in → any
.
```

Input arguments

Instructions may have no more than two input arguments.

Usually, in_1 supports any types of arguments, except for `RelSpPush`.

```
Inductive in_any : Type :=
| InReg : reg_io → in_any
| InImm : imm_in → in_any
| InStack: stack_in → in_any
| InCode : code_in → in_any
| InConst: const_in → in_any
.
```

Inclusion function

```
Definition in_any_incl (ia: in_any) : any :=
  match ia with
   | InReg x ⇒ AnyReg x
   | InImm x ⇒ AnyImm x
   | InStack x ⇒ AnyStack (stack_in_to_any x)
   | InCode x ⇒ AnyCode x
   | InConst x ⇒ AnyConst x
  end.
```

Usually, in_2 supports only arguments in GPRs.

```
Definition in_reg : Type := reg_io.
```

In exotic cases, an input argument may either be a register, or an immediate value, but not anything else.

```
Inductive in_regimm : Type :=
| RegImmR : reg_io → in_regimm
| RegImmI : imm_in → in_regimm
.
```

Inclusion function

```
Definition in_regimm_incl (ri: in_regimm) : in_any :=
  match ri with
   | RegImmR r ⇒ InReg r
   | RegImmI i ⇒ InImm i
  end.
```

Output arguments

Instructions may have no more than two output arguments.

Output arguments can not be immediate values.

A single immediate value is not sufficient to identify a memory cell, because we have multiple pages (see `page`).

Out arguments can not resolve to the addresses of constants or instructions, because `code_page` and `const_page` are not writable.

```
    Inductive out_any : Type :=
    | OutReg : reg_io → out_any
    | OutStack: stack_out → out_any
    .
```

Inclusion function

```
    Definition out_any_incl (ia: out_any) : any :=
      match ia with
      | OutReg x ⇒ AnyReg x
      | OutStack x ⇒ AnyStack (stack_out_to_any x)
      end.
    Definition out_reg : Type := reg_io.
  End InstructionArguments.

End Addressing.
```

Therefore, we do not define `out_regimm`, because it is impossible to write to immediate values.

```
Module Coercions.
  Coercion in_any_incl: in_any >-> any.
  Coercion out_any_incl : out_any >-> any.

  Coercion Imm : u16 >-> imm_in.
  Coercion InReg : reg_io >-> in_any.
  Coercion InImm : imm_in >-> in_any.
  Coercion InStack: stack_in >-> in_any.
  Coercion InCode: code_in >-> in_any.
  Coercion InConst: const_in >-> in_any.
  Coercion StackInOnly: stack_in_only >-> stack_in.
  Coercion stack_in_to_any: stack_in >-> stack_any.
  Coercion OutReg : reg_io >-> out_any.
  Coercion OutStack: stack_out >-> out_any.
  Coercion AnyStack: stack_any >-> any.
  Coercion StackOutOnly: stack_out_only >-> stack_out.
  Coercion in_regimm_incl: in_regimm >-> in_any.
  Coercion StackInAny : stack_io >-> stack_in.
  Coercion Reg : reg_name >-> reg_io.
End Coercions.
```

Library EraVM.Resolution

```
Require Core Addressing CallStack .

Import ssreflect ssrfun ssrbool eqtype ssreflect.tuple.
Import Addressing Core Common ZArith CallStack GPR PrimitiveValue Pointer
TransientMemory.

Section AllResolution.

Section AddressingUtils.
  Import MemoryBase.
  Open Scope ZMod_scope.
```

Predicate `reg_rel` implements the resolution for register-based relative addressing. Its specializations implement relative addressing for:

- the `code_page`: `reg_rel_code`;
- the `const_page`: `reg_rel_const`;
- the `stack_page`: `reg_rel_const`;

```
  (* Inductive equal_bits {n m} (x :BITS n) (y: BITS m) (H: n = m) : Prop := *)

  Definition low16 (w: word) : u16 := low 16 w.
  Definition low32 (w: word) : u32 := low 32 w.

  Inductive reg_rel : regs_state → reg_name → u16 → u16 → Prop :=
  | rca_code_pp: ∀ regs reg reg_val base ofs
                  abs OF_ignored,
      fetch_gpr regs reg = IntValue reg_val →
      base = low16 reg_val →
      base + ofs = (OF_ignored, abs) →
      reg_rel regs reg ofs abs.

  Definition reg_rel_code : regs_state → reg_name → u16 → code_address → Prop
    := reg_rel.
  Definition reg_rel_const : regs_state → reg_name → u16 → const_address → Prop
    := reg_rel.
  Definition reg_rel_stack : regs_state → reg_name → u16 → stack_address → Prop
    := reg_rel.
```

Note: in `sp_displ`, `delta` = `reg` + `imm`.

```
  Definition sp_displ: regs_state → reg_name → u16 → stack_address → Prop :=
reg_rel_stack.
End AddressingUtils.
```

Address resolution

Instructions have multiple ways of addressing data, i.e. immediate 16-bit values, GPRs, absolute or relative addresses in stack etc. They are described in section `Addressing` by types such as `in_any`, `out_reg`, and so on.

Location stands for a source and/or destination for data, addressable by instructions.

Address resolution is a matching between instruction operands and locations using the supported address modes.

There are five main locations that instructions can address:

1. Immediate data: the operand is provided directly as an unsigned 16-bit integer value.
2. Register: data can be fetched or stored to general purpose registers `r1`–`r15`.
3. Stack address: data can be fetched or stored to stack.
4. Code address: instructions can be fetched from a code page.
5. Constant address: data can be fetched from a read-only page holding constant words.

```
Inductive loc : Type :=
| LocImm: u16 → loc
| LocReg : reg_name → loc
| LocStackAddress: stack_address → loc
| LocCodeAddr: code_address → loc
| LocConstAddr: const_address → loc
.
```

Additionally, data can be fetched and stored to data pages; this process is more complicated and requires putting in registers specially formed pointers `heap_ptr`.

```
Section Resolution.
  Import Addressing.Coercions.

  Open Scope ZMod_scope.
```

Resolution of `RelSpPop` and `RelSpPush` addressing modes modifies the stack pointer. This, and possible future effects, is described by `resolve_effect` predicate.

```
  Inductive resolve_effect := | NoEffect | NewSP (val: stack_address).

  Record resolve_result :=
    mk_resolved {
        effect: resolve_effect;
        location:> loc;
      }.
```

```
  Context {state_checkpoint}
    (callstack := @callstack state_checkpoint)
    (rs:regs_state)
    (cs: callstack)
    (sp:= sp_get cs).

  Reserved Notation "[[ resolved ]]" (at level 9, no associativity).
  Reserved Notation "[[ resolved ; 'SP' <- newsp ]]" (at level 9, no
associativity).

  Declare Scope Resolution_scope.
  Open Scope Resolution_scope.
```

Address resolution is formalized by the predicate `resolve`.

```
  Inductive resolve : any → resolve_result → Prop :=
```

- Registers and immediate values are resolved to themselves.

```
  | rslv_reg : ∀ reg,
      resolve (Reg reg) [[ LocReg reg ]]
| rslv_imm: ∀ imm,
      resolve (Imm imm) [[ LocImm imm ]]
```

- `Absolute` : Absolute stack addressing with a general purpose register and an immediate displacement is resolved to $reg+imm$.

```
| rslv_stack_abs: ∀ regs reg imm abs,
    reg_rel_stack regs reg imm abs →
    resolve (Absolute reg imm) [[ LocStackAddress abs ]]
```

- `RelSP` : Addressing relative to SP with a general purpose register and an immediate displacement is resolved to $sp-(reg+imm)$.

```
| rslv_stack_rel: ∀ reg ofs delta_sp sp_rel,
    sp_displ rs reg ofs delta_sp →

    (false, sp_rel) = sp - delta_sp→
    resolve (RelSP reg ofs) [[ LocStackAddress sp_rel ]]
```

- `RelSpPop` : **Reading** relative to SP with a general purpose register and an immediate displacement, **and SP decrement**, is resolved to $sp-(reg+imm)$; additionally, SP is assigned

the same value $sp-(reg+imm)$. The new SP value will then be used for the resolution of other operands.

In other words, it is equivalent to a sequence of two actions:

1. SP = SP - (reg + imm)
2. [SP] -> result

In other words, it is equivalent to:

1. pop the stack (reg + imm - 1) times, discard these values
2. pop value from stack and return it

```
| rslv_stack_gpop: ∀ reg ofs delta_sp new_sp,
    sp_displ rs reg ofs delta_sp →
    (false, new_sp) = sp - delta_sp→
    resolve (RelSpPop reg ofs) [[ LocStackAddress new_sp ; SP <- new_sp ]]
```

- `RelSpPush` : **Writing** relative to SP with a general purpose register and an immediate displacement, **and SP increment**, is resolved to **the current value of SP**. Additionally, SP is assigned a new value $sp+(reg+imm)$. The new SP value will then be used for the resolution of other operands.

  In other words, it is equivalent to a sequence of two actions:

  1. [SP] <- input value
  2. SP = SP + (reg + imm)

  In other words, it is equivalent to:

  1. push the input value to the stack
  2. allocate (reg+imm-1) slots in stack

```
  | rslv_stack_gpush: ∀ reg ofs delta_sp new_sp,
      sp_displ rs reg ofs delta_sp →
      (false, new_sp) = sp + delta_sp →
      resolve (RelSpPush reg ofs) [[ LocStackAddress sp ; SP <- new_sp ]]

  | rslv_code: ∀ reg abs_imm addr,
      reg_rel_code rs reg abs_imm addr →
      resolve (CodeAddr reg abs_imm) [[ LocCodeAddr addr ]]
| rslv_const: ∀ reg abs_imm addr,
      reg_rel_const rs reg abs_imm addr →
      resolve (ConstAddr reg abs_imm) [[ LocConstAddr addr ]]
where
  "[[ resolved ]]" := (mk_resolved NoEffect resolved) : Resolution_scope
  and
  "[[ resolved ; 'SP' <- newsp ]]" := (mk_resolved (NewSP newsp) resolved) :
Resolution_scope.
```

Note: the reason of the asymmetry between `RelSpPush` and `RelSpPop` is because they are generalizations of `push` and `pop` operations.

Consider a stack implemented as an array with a pointer. Suppose that, like in EraVM, SP points the the next address after the last value in stack; in other words, the topmost element in the stack is located at $sp-1$. Then `push` and `pop` can be implemented in the following way:

- Push:
    1. [SP] <- new_value
    2. SP = SP + 1
- Pop:
    1. SP = SP - 1
    2. [SP] -> result

This asymmetry naturally generalizes to `RelSpPush` and `RelSpPop`.

Using `RelSpPop` and `RelSpPush` in one instruction

Suppose an instruction is using both `RelSpPop` and `RelSpPush`, then both effects are applied in order:

- First, the “in” effect of `RelSpPop`.
- Then, the “out” effect of `RelSpPush`, where SP is already changed by `RelSpPop`.
- If the instruction accesses SP, both effects will be applied prior to the instruction-specific logic.

See an example in `sem.ModSP.step`.

Predicate `apply_effects` formalizes the application of address resolution side effects to the state. In the current state of EraVM, only SP modifications are allowed, therefore the effect is limited to the topmost frame of callstack, which holds the current value of sp in `cf_sp`.

```
  Inductive apply_effects : resolve_effect → callstack → Prop :=
  | ae_none: ∀ s,
      apply_effects NoEffect s
  | ae_sp: ∀ cs' new_sp,
      sp_map_spec (fun _ ⇒ new_sp) cs cs' →
      apply_effects (NewSP new_sp) cs'.

  Inductive resolve_apply
    : any → (callstack × loc) → Prop :=
  | ra_no_effect : ∀ arg loc,
      resolve arg [[ loc ]] →
      resolve_apply arg (cs, loc)
  | ra_effect : ∀ cs' arg loc new_sp,
      resolve arg [[ loc ; SP <- new_sp ]] →
      sp_map_spec (fun _ ⇒ new_sp) cs cs' →
      resolve_apply arg (cs', loc).

End Resolution.
```

```
End AllResolution.
```

Library EraVM.isa.Instructions

```
Section InstructionSets.
```

EraVM instruction sets

Note If you are interested in the instruction set exposed to the assembly programmer, skim this section quickly and then proceed to `AssemblyInstructionSet`, `CoreInstructionSet` and `to_core`.

The spec introduces the following layers of abstraction for the instruction set, from lowest level to the highest level:

1. Binary encoding (as [BITS 64] type instances). Binary encoded instructions, each instruction is 64-bit wide.

   The exact type for such instructions is `BITS 64`.

   The definition `encode_mach_instruction` formalizes the encoding algorithm. It depends on `encode_predicate` and `encode_opcode`.

   The encoding is an injection, because most instructions ignore the values of some instruction fields such as `op_dst1`.

2. Low-level machine instructions `mach_instruction`.

   - Fixed format with two input operands, two output operands, two immediate values.
   - Restricted sources/destinations (e.g. second input can only be fetched from registers), instructions may ignore some instruction fields.

3. Assembly instructions (see section `AssemblyInstructionSet`).

   An assembly instruction is an instance of `predicated asm_instruction`. The type `asm_instruction` defines instruction format with the exact operand types and all their supported modifiers.

   The instruction semantic is defined for these instructions; see `step` in section `SmallStep`.

   The following abstraction levels simplify the definition of instruction semantic but are invisible for assembly programmer.

---

4. Core instructions (see the type `instruction` in section `CoreInstructionSet`). The exact type for such instructions is `predicated (instruction decoded)`.

   A simplified instruction set where the `mod_swap` modifier is applied and there are no restrictions on the operand sources or destinations, such as “the second input operand may only be fetched from register”. Because of fewer restrictions, the definitions are more uniform. The translation from `asm_instruction` to `instruction` is implemented in section `AssemblyToCore`.

Additionally, the type `instruction` describes the meaningful operand types for instructions after decoding, e.g. `OpFarRet` loads an operand from a register, but then deserializes it into a compound value. The types of such compound values are explicitly provided by `bound` definition.

The type `instruction` describes a schema of core instructions; this schema can be specialized to describe:

- fetched and decoded instruction `instruction decoded`;
- an instruction where input and output operands are bound to the memory locations `instruction bound`. This abstracts loading and storing operands, and their serialization/deserialization in case of ABI encoded instruction parameters.

Adding a layer of `instruction bound` allows describing instruction semantics as if instructions were accepting structural values directly, omitting fetching values, binary encoding and decoding.

- See `step_add` for an example of how instruction-specific semantic is defined omitting loading and storing details.
- See `step_ContextMeta` for an example of how instruction-specific semantic is defined omitting not only loading and storing, but also serialization/deserialization details.
- See `OperandBinding` for the details of binding, e.g. `bind_farret_params` describes how `OpFarRet`'s ABI parameters are deserialized.

```
End InstructionSets.
```

Library EraVM.isa.Modifiers

```
Require Flags.
Import Flags.

Section Modifiers.
```

Instruction modifiers

There are two modifiers common between multiple `asm_instruction`: `swap` and `set_flags`.

Swap modifier

This modifier, when applied, swaps two input operands. Definition `asm_instruction` shows which instructions support it.

Input operands usually have different addressing modes, e.g. the divisor in `Assembly.OpDiv` may be only fetched from registers, not memory. Applying `swap` modifier allows to fetch rather the divisor from memory, and the divident can be fetched from register.

The function `to_core` transforms `asm_instruction` into a core `instruction` and applies the `swap` modifier.

```
Inductive mod_swap := Swap | NoSwap.

Definition apply_swap {T} (md: mod_swap) (a b:T) : T×T :=
  match md with
  | NoSwap ⇒ (a,b)
  | Swap ⇒ (b,a)
  end.
```

Set flags

Only instructions *with the modifier* `mod_set_flags` *set* may change the state of flags in `gs_flags`. Therefore, if `mod_set_flags` is not set, it is guaranteed that the instruction will not change the flags state.

```
Inductive mod_set_flags := SetFlags | PreserveFlags.
```

If set flags modifier `md` is set, preserve the old flags state `f`; otherwise, return the new state `f'`.

```
  Definition apply_set_flags (md: mod_set_flags) (f f':flags_state) :
flags_state :=
    match md with
    | SetFlags ⇒ f'
    | PreserveFlags ⇒ f
    end.

End Modifiers.
```

Library EraVM.isa.Assembly

```
Require Addressing Common memory.Depot TransientMemory Pointer Predication
isa.Modifiers.

Import Addressing Common memory.Depot TransientMemory Pointer Predication
Modifiers.

Section AssemblyInstructionSet.
```

EraVM assembly instruction set

This section describes an instruction set `asm_instruction` which is a target for compiler.

The type `asm_instruction` defines instruction format with the precise types of their operands, and all their supported modifiers. This set is a slice in the middle of the abstractions hierarchy:

- The next lower level is machine instructions. The assembly encodes `asm_instruction` to the lower-level machine instructions which are then mapped to their binary encodings.
- The next higher level are **core instructions** described in section `CoreInstructionSet`. These instructions have are simplified formats, impose less constraints on the operand sources and destinations, and do not support the `mod_swap` modifier.

For all practical purposes, the reader of the specification should start at this level, unless their interest is in lower-level encoding details. The encoding layout is formalized by `mach_instruction` type, which is then serialized to binary by `encode_mach_instruction`.

The function `base_cost` defines the basic costs of each instruction in **`ergs`**.

```
Inductive asm_instruction: Type :=
| OpInvalid
| OpNoOp
| OpSpAdd (in1: in_reg) (ofs: imm_in)
(* encoded as NoOp with out_1 in address mode Addressing.RelSpPush*)
  | OpSpSub (in1: in_reg) (ofs: imm_in)
(* encoded as NoOp with in_1  in address mode Addressing.RelSpPop *)
  | OpJump (dest: in_any)
  | OpAnd (in1: in_any) (in2: in_reg) (out1: out_any) (flags:mod_set_flags)
  | OpOr (in1: in_any) (in2: in_reg) (out1: out_any) (flags:mod_set_flags)
  | OpXor (in1: in_any) (in2: in_reg) (out1: out_any) (flags:mod_set_flags)
  | OpAdd (in1: in_any) (in2: in_reg) (out1: out_any) (flags:mod_set_flags)
  | OpSub (in1: in_any) (in2: in_reg) (out1: out_any) (swap:mod_swap)
(flags:mod_set_flags)

  | OpShl (in1: in_any) (in2: in_reg) (out1: out_any) (swap:mod_swap)
(flags:mod_set_flags)
  | OpShr (in1: in_any) (in2: in_reg) (out1: out_any) (swap:mod_swap)
(flags:mod_set_flags)
  | OpRol (in1: in_any) (in2: in_reg) (out1: out_any) (swap:mod_swap)
(flags:mod_set_flags)
  | OpRor (in1: in_any) (in2: in_reg) (out1: out_any) (swap:mod_swap)
(flags:mod_set_flags)

  | OpMul (in1: in_any) (in2: in_reg) (out1: out_any) (out2: out_reg)
(flags:mod_set_flags)
  | OpDiv (in1: in_any) (in2: in_reg) (out1: out_any) (out2: out_reg)
(swap:mod_swap) (flags:mod_set_flags)

  | OpNearCall (arg: in_reg) (dest: imm_in) (handler: imm_in)
  | OpFarCall (enc: in_reg) (dest: in_reg) (handler: imm_in) (is_static:bool)
(is_shard_provided: bool)
```

```
  | OpMimicCall (enc: in_reg) (dest: in_reg) (handler: imm_in) (is_static:bool)
(is_shard_provided: bool)
  | OpDelegateCall(enc: in_reg) (dest: in_reg) (handler: imm_in)
(is_static:bool) (is_shard_provided: bool)

  | OpNearRet
  | OpNearRetTo (dest: imm_in)
  | OpFarRet (args: in_reg)

  | OpNearRevert
  | OpNearRevertTo(dest: imm_in)
  | OpFarRevert (args: in_reg)
  | OpNearPanicTo (dest: imm_in)
  | OpPanic

  | OpPtrAdd (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
  | OpPtrSub (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
  | OpPtrShrink (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
  | OpPtrPack (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)


  | OpLoad (ptr: in_regimm) (res: out_reg) (mem:data_page_type)
  | OpLoadInc (ptr: in_regimm) (res: out_reg) (mem:data_page_type) (inc_ptr:
out_reg)
  | OpStore (ptr: in_regimm) (val: in_reg) (mem:data_page_type)
  | OpStoreInc (ptr: in_regimm) (val: in_reg) (mem:data_page_type) (inc_ptr:
out_reg)

  | OpLoadPointer (ptr: in_reg) (res: out_reg)
  | OpLoadPointerInc (ptr: in_reg) (res: out_reg) (inc_ptr: out_reg)


  | OpContextThis (out: out_reg)
  | OpContextCaller (out: out_reg)
  | OpContextCodeAddress (out: out_reg)
  | OpContextMeta (out: out_reg)
  | OpContextErgsLeft (out: out_reg)
  | OpContextSp (out: out_reg)
  | OpContextGetContextU128 (out: out_reg)
  | OpContextSetContextU128 (in1: in_reg)
  | OpContextSetErgsPerPubdataByte (in1: in_reg)
  | OpContextIncrementTxNumber


  | OpSLoad (in1: in_reg) (out: out_reg)
  | OpSStore (in1: in_reg) (in2: in_reg)

  | OpPrecompileCall (in1: in_reg) (in2: in_reg) (out: out_reg)

  | OpEvent (in1: in_reg) (in2: in_reg) (is_first: bool)
  | OpToL1Message (in1: in_reg) (in2: in_reg) (is_first: bool)
  .

End AssemblyInstructionSet.
```

Library EraVM.isa.CoreSet

```
Require Addressing isa.Modifiers Pointer Predication TransientMemory.
Import Addressing Modifiers TransientMemory Pointer Predication.

Section CoreInstructionSet.
```

Core instruction set

This section describes a schema `instruction` of a simplified instruction set appearing in semantic definitions for `asm_instruction`. The meaning of "schema" here is that `instruction` needs to be specialized with a proper instance of `descr` record to describe an instruction at different stages of its execution.

The schema [instruction] is parameterized with types of various instruction operands. Conventionally, their names reflect the following information:

- the prefix is `src_` for source operands and `dest_` for output operands.
- the suffix shows the type of data fetched or stored from register/memory.

For example:

- `src_pv` means "a source operand with a meaning of a `primitive_value`";
- `src_fat_ptr` means "a source operand which will be decoded to a `fat_ptr`";
- `dest_heap_ptr` means "a destination operand with a meaning of `heap_ptr`; such pointer will be encoded and written to memory".

Therefore, the exact definitions of instructions should be specialized with an instance of `descr` to give precise meaning to instruction operands. Currently, two such instances are used:

- `instruction decoded` is decoded from `asm_instruction`.
- `instruction bound` is an instruction where operands have been bound to their source/ destination locations in memory; at this stage, reads, writes, encoding and decodings (parts of ABI) are accounted for.

Having `instruction bound` allows attributing semantic in a more concise way. See `OperandBinding`. For additional versatility, the definitions may bind both the decoded values and their representations as primitive values; see `bound`.

```
Structure descr: Type := {
    src_pv: Type;
    src_fat_ptr: Type;
    src_heap_ptr: Type ;
    src_farcall_params: Type;
    src_nearcall_params: Type;
    src_ret_params: Type;
    src_precompile_params: Type;
    dest_pv: Type;
    dest_heap_ptr: Type;
    dest_fat_ptr: Type;
    dest_meta_params: Type;
  }.
```

```
  Context {d: descr}
    (src_pv:= src_pv d)
    (src_fat_ptr:= src_fat_ptr d)
    (src_heap_ptr:= src_heap_ptr d)
    (src_farcall_params:= src_farcall_params d)
    (src_nearcall_params:= src_nearcall_params d)
    (src_ret_params:= src_ret_params d)
    (src_precompile_params:= src_precompile_params d)
    (dest_pv:= dest_pv d)
    (dest_heap_ptr:= dest_heap_ptr d)
    (dest_fat_ptr:= dest_fat_ptr d)
    (dest_meta_params:= dest_meta_params d)
  .

  Inductive instruction: Type :=
  | OpInvalid
  | OpNoOp
  | OpSpAdd (in1: src_pv) (ofs: stack_address)
(* encoded as NoOp with $out_1$ *)
  | OpSpSub (in1: src_pv) (ofs: stack_address)
(* encoded as NoOp with $in_1$ *)

  | OpJump (dest: src_pv)
  | OpAnd (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)
  | OpOr (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)
  | OpXor (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)
  | OpAdd (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)
  | OpSub (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)
  | OpShl (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)
  | OpShr (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)
  | OpRol (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)
  | OpRor (in1: src_pv) (in2: src_pv) (out1: dest_pv) (flags:mod_set_flags)

  | OpMul (in1: src_pv) (in2: src_pv) (out1: dest_pv) (out2: dest_pv)
(flags:mod_set_flags)
  | OpDiv (in1: src_pv) (in2: src_pv) (out1: dest_pv) (out2: dest_pv)
(flags:mod_set_flags)
  | OpNearCall (in1: src_nearcall_params) (dest: code_address) (handler:
code_address)
  | OpFarCall (enc: src_farcall_params) (dest: src_pv) (handler: code_address)
(is_static:bool) (is_shard_provided: bool)
  | OpMimicCall (enc: src_farcall_params) (dest: src_pv) (handler:
code_address) (is_static:bool) (is_shard_provided: bool)
  | OpDelegateCall(enc: src_farcall_params) (dest: src_pv) (handler:
code_address) (is_static:bool) (is_shard_provided: bool)

  | OpNearRet
  | OpNearRetTo (label: code_address)
  | OpFarRet (args: src_ret_params)

  | OpNearRevert
  | OpNearRevertTo (label: code_address)
  | OpFarRevert (args: src_ret_params)
  | OpNearPanicTo (label: code_address)
  | OpPanic
```

```
  | OpPtrAdd (in1: src_fat_ptr) (in2: src_pv) (out: dest_fat_ptr)
  | OpPtrSub (in1: src_fat_ptr) (in2: src_pv) (out: dest_fat_ptr)
  | OpPtrShrink (in1: src_fat_ptr) (in2: src_pv) (out: dest_fat_ptr)
  | OpPtrPack (in1: src_fat_ptr) (in2: src_pv) (out: dest_pv)


  | OpLoad (ptr: src_heap_ptr) (res: dest_pv) (mem:data_page_type)
  | OpLoadInc (ptr: src_heap_ptr) (res: dest_pv) (mem:data_page_type) (inc_ptr:
dest_heap_ptr)
  | OpStore (ptr: src_heap_ptr) (val: src_pv) (mem:data_page_type)
  | OpStoreInc (ptr: src_heap_ptr) (val: src_pv) (mem:data_page_type) (inc_ptr:
dest_heap_ptr)


  | OpLoadPointer (ptr: src_fat_ptr) (res: dest_pv)
  | OpLoadPointerInc (ptr: src_fat_ptr) (res: dest_pv) (inc_ptr: dest_fat_ptr)


  | OpContextThis (out: dest_pv)
  | OpContextCaller (out: dest_pv)
  | OpContextCodeAddress (out: dest_pv)
  | OpContextMeta (out: dest_meta_params)
  | OpContextErgsLeft (out: dest_pv)
  | OpContextSp (out: dest_pv)
  | OpContextGetContextU128 (out: dest_pv)
  | OpContextSetContextU128 (in1: src_pv)
  | OpContextSetErgsPerPubdataByte (in1: src_pv)
  | OpContextIncrementTxNumber

  | OpSLoad (in1: src_pv) (out: dest_pv)
  | OpSStore (in1: src_pv) (in2: src_pv)
  | OpToL1Message (in1: src_pv) (in2: src_pv) (is_first: bool)
  | OpEvent (in1: src_pv) (in2: src_pv) (is_first: bool)
  | OpPrecompileCall (in1: src_precompile_params) (ergs:src_pv) (out: dest_pv)
  .

End CoreInstructionSet.

#[global]
  Canonical Structure decoded: descr :=
  let src := in_any in
  let dest := out_any in
  {|
    src_pv := src;
    src_fat_ptr :=src;
    src_heap_ptr :=src;
    src_farcall_params := src;
    src_nearcall_params := src;
    src_ret_params := src;
    src_precompile_params := src;
    dest_pv := dest;
    dest_heap_ptr := dest;
    dest_fat_ptr := dest;
    dest_meta_params := dest;
  |}.
```

```
Section InstructionMapper.

  Notation state_rel S OP V := (S → S → OP → V → Prop).

  Record relate_st {S} {A B: descr}: Type :=
    {
      mf_src_pv : state_rel S (src_pv A) (src_pv B);
      mf_src_fat_ptr: state_rel S (src_fat_ptr A) (src_fat_ptr B);
      mf_src_heap_ptr: state_rel S (src_heap_ptr A) (src_heap_ptr B);
      mf_src_farcall_params: state_rel S (src_farcall_params A)
(src_farcall_params B);
      mf_src_nearcall_params: state_rel S (src_nearcall_params A)
(src_nearcall_params B);
      mf_src_ret_params: state_rel S (src_ret_params A) (src_ret_params B);
      mf_src_precompile_params: state_rel S (src_precompile_params A)
(src_precompile_params B);
      mf_dest_pv: state_rel S (dest_pv A) (dest_pv B);
      mf_dest_heap_ptr: state_rel S (dest_heap_ptr A) (dest_heap_ptr B);
      mf_dest_fat_ptr : state_rel S (dest_fat_ptr A) (dest_fat_ptr B);
      mf_dest_meta_params: state_rel S (dest_meta_params A) (dest_meta_params
B);
    }.

  Generalizable Variables s i o imm fs.

  Context {S A B} (m:@relate_st S A B)
    (mf_src_pv := mf_src_pv m)
    (mf_src_fat_ptr := mf_src_fat_ptr m)
    (mf_src_heap_ptr := mf_src_heap_ptr m)
    (mf_src_farcall_params := mf_src_farcall_params m)
    (mf_src_nearcall_params := mf_src_nearcall_params m)
    (mf_src_ret_params := mf_src_ret_params m)
    (mf_src_precompile_params := mf_src_precompile_params m)
    (mf_dest_pv := mf_dest_pv m)
    (mf_dest_heap_ptr := mf_dest_heap_ptr m)
    (mf_dest_fat_ptr := mf_dest_fat_ptr m)
    (mf_dest_meta_params := mf_dest_meta_params m)
.

Inductive ins_srelate : S → S → @instruction A → @instruction B → Prop :=
|sim_noop: ∀ s, ins_srelate s s OpNoOp OpNoOp
|sim_invalid: ∀ s, ins_srelate s s OpInvalid OpInvalid
|sim_sp_add: `(
                mf_src_pv s0 s1 i1 i1' →
                ins_srelate s0 s1 (OpSpAdd i1 imm1) (OpSpAdd i1' imm1)
              )
|sim_sp_sub: `(
                mf_src_pv s0 s1 i1 i1' →
                ins_srelate s0 s1 (OpSpSub i1 imm1) (OpSpSub i1' imm1)
              )
|sim_jump: `(
               mf_src_pv s0 s1 i1 i1' →
               ins_srelate s0 s1 (OpJump i1) (OpJump i1')
             )
|sim_and: `(
              mf_src_pv s0 s1 i1 i1' →
              mf_src_pv s1 s2 i2 i2' →
```

```
            mf_dest_pv s2 s3 o1 o1' →
            ins_srelate s0 s3 (OpAnd i1 i2 o1 fs) (OpAnd i1' i2' o1' fs)
          )
|sim_or: `(
           mf_src_pv s0 s1 i1 i1' →
           mf_src_pv s1 s2 i2 i2' →
           mf_dest_pv s2 s3 o1 o1' →
           ins_srelate s0 s3 (OpOr i1 i2 o1 fs) (OpOr i1' i2' o1' fs)
         )
|sim_xor: `(
            mf_src_pv s0 s1 i1 i1' →
            mf_src_pv s1 s2 i2 i2' →
            mf_dest_pv s2 s3 o1 o1' →
            ins_srelate s0 s3 (OpXor i1 i2 o1 fs) (OpXor i1' i2' o1' fs)
          )
|sim_add: `(
            mf_src_pv s0 s1 i1 i1' →
            mf_src_pv s1 s2 i2 i2' →
            mf_dest_pv s2 s3 o1 o1' →
            ins_srelate s0 s3 (OpAdd i1 i2 o1 fs) (OpAdd i1' i2' o1' fs)
          )
|sim_sub: `(
            mf_src_pv s0 s1 i1 i1' →
            mf_src_pv s1 s2 i2 i2' →
            mf_dest_pv s2 s3 o1 o1' →
            ins_srelate s0 s3 (OpSub i1 i2 o1 fs) (OpSub i1' i2' o1' fs)
          )
|sim_shl: `(
            mf_src_pv s0 s1 i1 i1' →
            mf_src_pv s1 s2 i2 i2' →
            mf_dest_pv s2 s3 o1 o1' →
            ins_srelate s0 s3 (OpShl i1 i2 o1 fs) (OpShl i1' i2' o1' fs)
          )
|sim_shr: `(
            mf_src_pv s0 s1 i1 i1' →
            mf_src_pv s1 s2 i2 i2' →
            mf_dest_pv s2 s3 o1 o1' →
            ins_srelate s0 s3 (OpShr i1 i2 o1 fs) (OpShr i1' i2' o1' fs)
          )
|sim_rol: `(
            mf_src_pv s0 s1 i1 i1' →
            mf_src_pv s1 s2 i2 i2' →
            mf_dest_pv s2 s3 o1 o1' →
            ins_srelate s0 s3 (OpRol i1 i2 o1 fs) (OpRol i1' i2' o1' fs)
          )
|sim_ror: `(
            mf_src_pv s0 s1 i1 i1' →
            mf_src_pv s1 s2 i2 i2' →
            mf_dest_pv s2 s3 o1 o1' →
            ins_srelate s0 s3 (OpRor i1 i2 o1 fs) (OpRor i1' i2' o1' fs)
          )
|sim_PtrAdd: `(
               mf_src_fat_ptr s0 s1 i1 i1' →
               mf_src_pv s1 s2 i2 i2' →
               mf_dest_fat_ptr s2 s3 o1 o1' →
               ins_srelate s0 s3 (OpPtrAdd i1 i2 o1) (OpPtrAdd i1' i2' o1')
             )
```

```
  |sim_PtrSub: `(
                  mf_src_fat_ptr s0 s1 i1 i1' →
                  mf_src_pv s1 s2 i2 i2' →
                  mf_dest_fat_ptr s2 s3 o1 o1' →
                  ins_srelate s0 s3 (OpPtrSub i1 i2 o1) (OpPtrSub i1' i2' o1')
                )
  |sim_PtrShrink: `(
                     mf_src_fat_ptr s0 s1 i1 i1' →
                     mf_src_pv s1 s2 i2 i2' →
                     mf_dest_fat_ptr s2 s3 o1 o1' →
                     ins_srelate s0 s3 (OpPtrShrink i1 i2 o1) (OpPtrShrink i1'
i2' o1')
                   )
  |sim_PtrPack: `(
                    mf_src_fat_ptr s0 s1 i1 i1' →
                    mf_src_pv s1 s2 i2 i2' →
                    mf_dest_pv s2 s3 o1 o1' →
                    ins_srelate s0 s3 (OpPtrPack i1 i2 o1) (OpPtrPack i1' i2'
o1')
                  )
  |sim_mul: `(
                mf_src_pv s0 s1 i1 i1' →
                mf_src_pv s1 s2 i2 i2' →
                mf_dest_pv s2 s3 o1 o1' →
                mf_dest_pv s3 s4 o2 o2' →
                ins_srelate s0 s4 (OpMul i1 i2 o1 o2 fs) (OpMul i1' i2' o1' o2'
fs )
              )
  |sim div: `(
                mf_src_pv s0 s1 i1 i1' →
                mf_src_pv s1 s2 i2 i2' →
                mf_dest_pv s2 s3 o1 o1' →
                mf_dest_pv s3 s4 o2 o2' →
                ins_srelate s0 s4 (OpDiv i1 i2 o1 o2 fs) (OpDiv i1' i2' o1' o2'
fs)
              )
  |sim_nearcall: `(
                    ∀ dest handler,
                      mf_src_nearcall_params s0 s1 i1 i1' →
                      ins_srelate s0 s1 (OpNearCall i1 dest handler)
(OpNearCall i1' dest handler)
                  )
  |sim_farcall: `(
                   ∀ handler static shard,
                     mf_src_farcall_params s0 s1 i1 i1' →
                     mf_src_pv s1 s2 i2 i2' →
                     ins_srelate s0 s2 (OpFarCall i1 i2 handler static shard)
(OpFarCall i1' i2' handler static shard)
                 )
  |sim_mimiccall: `(
                     ∀ handler static shard,
                       mf_src_farcall_params s0 s1 i1 i1' →
                       mf_src_pv s1 s2 i2 i2' →
                       ins_srelate s0 s2 (OpMimicCall i1 i2 handler static
shard) (OpMimicCall i1' i2' handler static shard)
                   )
  |sim_delegatecall: `(
```

```
                        ∀ handler static shard,
                          mf_src_farcall_params s0 s1 i1 i1' →
                          mf_src_pv s1 s2 i2 i2' →
                          ins_srelate s0 s2(OpDelegateCall i1 i2 handler
static shard) (OpDelegateCall i1' i2' handler static shard)
                       )
  |sim_nearret: ∀ s, ins_srelate s s OpNearRet OpNearRet
  |sim_nearrevert: ∀ s, ins_srelate s s OpNearRevert OpNearRevert
  |sim_nearpanic: ∀ s, ins_srelate s s OpPanic OpPanic
  |sim_nearretto: ∀ s l, ins_srelate s s (OpNearRetTo l) (OpNearRetTo l)
  |sim_nearrevertto: ∀ s l, ins_srelate s s (OpNearRevertTo l) (OpNearRevertTo
l)
  |sim_nearpanicto: ∀ s l, ins_srelate s s (OpNearPanicTo l) (OpNearPanicTo l)
  |sim_farret: `(
                   mf_src_ret_params s0 s1 i1 i1' →
                   ins_srelate s0 s1 (OpFarRet i1) (OpFarRet i1')
                 )
  |sim_farrevert: `(
                      mf_src_ret_params s0 s1 i1 i1' →
                      ins_srelate s0 s1 (OpFarRevert i1) (OpFarRevert i1')
                    )
  |sim_load:`( ∀ type,
         mf_src_heap_ptr s0 s1 i1 i1' →
         mf_dest_pv s1 s2 o1 o1' →
         ins_srelate s0 s2 (OpLoad i1 o1 type) (OpLoad i1' o1' type)
     )
  |sim_loadptr:`(
       mf_src_fat_ptr s0 s1 i1 i1' →
       mf_dest_pv s1 s2 o1 o1' →
       ins_srelate s0 s2 (OpLoadPointer i1 o1) (OpLoadPointer i1' o1' )
     )
  |sim_loadinc:`( ∀ type,
         mf_src_heap_ptr s0 s1 i1 i1' →
         mf_dest_pv s1 s2 o1 o1' →
         mf_dest_heap_ptr s2 s3 o2 o2' →
         ins_srelate s0 s3 (OpLoadInc i1 o1 type o2) (OpLoadInc i1' o1' type
o2')
     )
  |sim_loadptrinc:`(
       mf_src_fat_ptr s0 s1 i1 i1' →
       mf_dest_pv s1 s2 o1 o1' →
       mf_dest_fat_ptr s2 s3 o2 o2' →
       ins_srelate s0 s3 (OpLoadPointerInc i1 o1 o2) (OpLoadPointerInc i1' o1'
o2')
     )
  |sim_store:`( ∀ type,
         mf_src_heap_ptr s0 s1 i1 i1' →
         mf_src_pv s1 s2 i2 i2' →
         mf_dest_pv s2 s3 o1 o1' →
         ins_srelate s0 s3 (OpStore i1 i2 type) (OpStore i1' i2' type)
     )
  |sim_storeinc:`( ∀ type,
         mf_src_heap_ptr s0 s1 i1 i1' →
         mf_src_pv s1 s2 i2 i2' →
         mf_dest_heap_ptr s2 s3 o1 o1' →
         ins_srelate s0 s3 (OpStoreInc i1 i2 type o1) (OpStoreInc i1' i2' type
o1')
```

```
    )
  |sim_OpContextThis: `(
                          mf_dest_pv s0 s1 o1 o1' →
                          ins_srelate s0 s1 (OpContextThis o1) (OpContextThis
o1'))
  |sim_OpContextCaller: `(
                            mf_dest_pv s0 s1 o1 o1' →
                            ins_srelate s0 s1 (OpContextCaller o1)
(OpContextCaller o1'))
  |sim_OpContextCodeAddress: `(
                                mf_dest_pv s0 s1 o1 o1' →
                                ins_srelate s0 s1 (OpContextCodeAddress o1)
(OpContextCodeAddress o1'))
  |sim_OpContextMeta: `(
                          mf_dest_meta_params s0 s1 o1 o1' →
                          ins_srelate s0 s1 (OpContextMeta o1) (OpContextMeta
o1'))
  |sim_OpContextErgsLeft: `(
                             mf_dest_pv s0 s1 o1 o1' →
                             ins_srelate s0 s1 (OpContextErgsLeft o1)
(OpContextErgsLeft o1'))
  |sim_OpContextSp: `(
                        mf_dest_pv s0 s1 o1 o1' →
                        ins_srelate s0 s1 (OpContextSp o1) (OpContextSp o1'))
  |sim_OpContextGetContextU128: `(
                                   mf_dest_pv s0 s1 o1 o1' →
                                   ins_srelate s0 s1 (OpContextGetContextU128
o1) (OpContextGetContextU128 o1'))
  |sim_OpContextSetContextU128: `(
                                   mf_src_pv s0 s1 i1 i1' →
                                   ins_srelate s0 s1 (OpContextSetContextU128
i1) (OpContextSetContextU128 i1'))
  |sim_OpContextSetErgsPerPubdataByte: `(
                                          mf_src_pv s0 s1 i1 i1' →
                                          ins_srelate s0 s1
(OpContextSetErgsPerPubdataByte i1) (OpContextSetErgsPerPubdataByte i1'))
  |sim_OpContextIncrementTxNumber: `(
                                      mf_dest_pv s0 s1 o1 o1' →
                                      ins_srelate s0 s1
(OpContextIncrementTxNumber ) (OpContextIncrementTxNumber ))
  |sim_OpSLoad: `(
                    mf_src_pv s0 s1 i1 i1' →
                    mf_dest_pv s1 s2 o1 o1' →
                    ins_srelate s0 s2 (OpSLoad i1 o1) (OpSLoad i1' o1'))
  |sim_OpSStore: `(
                     mf_src_pv s0 s1 i1 i1' →
                     mf_src_pv s1 s2 i2 i2' →
                     ins_srelate s0 s2 (OpSStore i1 i2) (OpSStore i1' i2'))
  |sim_OpToL1Message: `(∀ first,
                           mf_src_pv s0 s1 i1 i1' →
                           mf_src_pv s1 s2 i2 i2' →
                           ins_srelate s0 s2 (OpToL1Message i1 i2 first)
(OpToL1Message i1' i2' first))
  |sim_OpEvent: `(∀ first,
                     mf_src_pv s0 s1 i1 i1' →
                     mf_src_pv s1 s2 i2 i2' →
                     ins_srelate s0 s2 (OpEvent i1 i2 first) (OpEvent i1' i2'
```

```
first))
  |sim_OpPrecompileCall: `(
                           mf_src_precompile_params s0 s1 i1 i1' →
                           mf_src_pv s1 s2 i2 i2' →
                           mf_dest_pv s2 s3 o1 o1' →
                           ins_srelate s0 s3 (OpPrecompileCall i1 i2 o1)
(OpPrecompileCall i1' i2' o1'))
  .

  Generalizable No Variables.
End InstructionMapper.
```

Library EraVM.isa.GeneratedMachISA

```
(* GENERATED FILE, DO NOT EDIT MANUALLY. *)
From RecordUpdate Require Import RecordSet.
Require isa.Modifiers isa.Assembly Predication.
Import Types Modifiers Predication.

Inductive src_mode :=
| SrcReg
| SrcSpRelativePop
| SrcSpRelative
| SrcStackAbsolute
| SrcImm
| SrcCodeAddr
.

Inductive src_special_mode := | SrcSpecialReg | SrcSpecialImm.

Inductive dst_mode :=
| DstReg
| DstSpRelativePush
| DstSpRelative
| DstStackAbsolute
.
Inductive mod_context: Set :=
  | This
  | Caller
  | CodeAddress
  | Meta
  | ErgsLeft
  | Sp
  | GetContextU128
  | SetContextU128
  | SetErgsPerPubdataByte
  | IncrementTxNumber

.
Inductive mach_opcode : Type :=
| OpShr (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
(set_flags: mod_set_flags)
```

```
| OpNoOp (src0_mode: src_mode) (dst0_mode: dst_mode)
| OpDiv (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
(set_flags: mod_set_flags)
| OpContextSp
| OpRet (to_label: bool)
| OpStoreHeap (src0_mode: src_special_mode) (inc: bool)
| OpRor (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
(set_flags: mod_set_flags)
| OpContextSetContextU128
| OpLogToL1 (is_first: bool)
| OpContextThis
| OpLogEvent (is_first: bool)
| OpLoadPtr (inc: bool)
| OpLoadHeap (src0_mode: src_special_mode) (inc: bool)
| OpSub (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
(set_flags: mod_set_flags)
| OpContextSetErgsPerPubdataByte
| OpShl (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
(set_flags: mod_set_flags)
| OpContextCodeAddress
| OpContextIncrementTxNumber
| OpXor (src0_mode: src_mode) (dst0_mode: dst_mode) (set_flags: mod_set_flags)
| OpContextMeta
| OpSstore
| OpPtrSub (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
| OpJump (src0_mode: src_mode)
| OpLoadAuxHeap (src0_mode: src_special_mode) (inc: bool)
| OpPtrAdd (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
| OpSload
| OpAnd (src0_mode: src_mode) (dst0_mode: dst_mode) (set_flags: mod_set_flags)
| OpDelegate (is_shard: bool) (is_static: bool)
| OpRol (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
(set_flags: mod_set_flags)
| OpPtrPack (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
| OpMul (src0_mode: src_mode) (dst0_mode: dst_mode) (set_flags: mod_set_flags)
| OpOr (src0_mode: src_mode) (dst0_mode: dst_mode) (set_flags: mod_set_flags)
| OpPtrShrink (src0_mode: src_mode) (dst0_mode: dst_mode) (swap: mod_swap)
| OpContextErgsLeft
| OpContextGetContextU128
| OpCall
| OpContextCaller
| OpFarcall (is_shard: bool) (is_static: bool)
| OpLogPrecompile
| OpInvalid
| OpAdd (src0_mode: src_mode) (dst0_mode: dst_mode) (set_flags: mod_set_flags)
| OpMimic (is_shard: bool) (is_static: bool)
| OpRevert (to_label: bool)
| OpPanic (to_label: bool)
| OpStoreAuxHeap (src0_mode: src_special_mode) (inc: bool)
.
```

The definition `mach_instruction` showcases the layout of instruction fields. This layout is applied against the instruction's binary representation.

```
Section MachInstructionDefinition.
  Context {reg_type imm_type: Type}.
  Record mach_instruction :=
    mk_ins {
        op_code: mach_opcode;
        op_predicate: predicate;
        op_src0: reg_type;
        op_src1: reg_type;
        op_dst0: reg_type;
        op_dst1: reg_type;
        op_imm0: imm_type;
        op_imm1: imm_type;
      }.
  #[export] Instance etaIns : Settable _ := settable! mk_ins < op_code;
op_predicate; op_src0; op_src1; op_dst0; op_dst1; op_imm0; op_imm1>.
End MachInstructionDefinition.
```

Library EraVM.isa.AssemblyToCore

```
Require Addressing Assembly isa.Modifiers CoreSet.
Import Addressing isa.Modifiers CoreSet.

Import Addressing.Coercions.
Section AssemblyToCore.
```

Syntactically translate `asm_instruction` to a core `instruction`, preceding execution:

1. apply `mod_swap` modifier to instructions where applicable;
2. remove the restrictions on operand types e.g. in assembly `OpAdd` may accept the first argument in memory and the second argument only in register, but in core set both arguments can be fetched from either memory or registers. This simplifies attributing semantic to instructions.

```
  Definition to_core (input: Assembly.asm_instruction) : @instruction decoded
:=
    match input with
    | Assembly.OpInvalid ⇒ OpInvalid
    | Assembly.OpNoOp ⇒ OpNoOp
    | Assembly.OpSpAdd in1 (Imm ofs) ⇒ @OpSpAdd decoded in1 ofs
    | Assembly.OpSpSub in1 (Imm ofs) ⇒ @OpSpSub decoded in1 ofs
    | Assembly.OpJump dest ⇒ @OpJump decoded dest
    | Assembly.OpAnd in1 in2 out1 flags ⇒
        @OpAnd decoded in1 in2 out1 flags
    | Assembly.OpOr in1 in2 out1 flags ⇒
        @OpOr decoded in1 in2 out1 flags
    | Assembly.OpXor in1 in2 out1 flags ⇒
        @OpXor decoded in1 in2 out1 flags
    | Assembly.OpAdd in1 in2 out1 flags ⇒
        @OpAdd decoded in1 in2 out1 flags
    | Assembly.OpSub in1 in2 out1 swap flags ⇒
        let (in1', in2') := apply_swap swap in1 in2 in
        @OpSub decoded in1' in2' out1 flags
```

```
| Assembly.OpShl in1 in2 out1 swap flags ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpShl decoded in1' in2' out1 flags
| Assembly.OpShr in1 in2 out1 swap flags ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpShr decoded in1' in2' out1 flags
| Assembly.OpRol in1 in2 out1 swap flags ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpRol decoded in1' in2' out1 flags
| Assembly.OpRor in1 in2 out1 swap flags ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpRor decoded in1' in2' out1 flags
| Assembly.OpMul in1 in2 out1 out2 flags ⇒
    @OpMul decoded in1 in2 out1 out2 flags
| Assembly.OpDiv in1 in2 out1 out2 swap flags ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpDiv decoded in1' in2' out1 out2 flags
| Assembly.OpPtrAdd in1 in2 out swap ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpPtrAdd decoded in1' in2' out
| Assembly.OpPtrSub in1 in2 out swap ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpPtrSub decoded in1' in2' out
| Assembly.OpPtrShrink in1 in2 out swap ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpPtrShrink decoded in1' in2' out
| Assembly.OpPtrPack in1 in2 out swap ⇒
    let (in1', in2') := apply_swap swap in1 in2 in
    @OpPtrPack decoded in1' in2' out
    | Assembly.OpStore ptr val mem ⇒
        @OpStore decoded ptr val mem

    | Assembly.OpNearCall in1 (Imm dest) (Imm handler) ⇒ @OpNearCall decoded
in1 dest handler
    | Assembly.OpFarCall enc dest (Imm handler) is_static is_shard_provided ⇒
        @OpFarCall decoded enc dest handler is_static is_shard_provided
    | Assembly.OpMimicCall enc dest (Imm handler) is_static is_shard_provided ⇒
        @OpMimicCall decoded enc dest handler is_static is_shard_provided
    | Assembly.OpDelegateCall enc dest (Imm handler) is_static
is_shard_provided ⇒
        @OpDelegateCall decoded enc dest handler is_static is_shard_provided
    | Assembly.OpNearRet ⇒ OpNearRet
    | Assembly.OpNearRetTo (Imm dest) ⇒ OpNearRetTo dest

    | Assembly.OpFarRet args ⇒ @OpFarRet decoded args
    | Assembly.OpNearRevert ⇒ @OpNearRevert decoded
    | Assembly.OpNearRevertTo (Imm dest) ⇒ @OpNearRevertTo decoded dest
    | Assembly.OpFarRevert args ⇒ @OpFarRevert decoded args
    | Assembly.OpNearPanicTo (Imm label) ⇒ @OpNearPanicTo decoded label
    | Assembly.OpPanic ⇒ @OpPanic decoded
    | Assembly.OpLoad ptr res mem ⇒ @OpLoad decoded ptr res mem
    | Assembly.OpLoadInc ptr res mem inc_ptr ⇒ @OpLoadInc decoded ptr res mem
inc_ptr
    | Assembly.OpStoreInc ptr val mem inc_ptr ⇒ @OpStoreInc decoded ptr val mem
inc_ptr
    | Assembly.OpLoadPointer ptr res ⇒ @OpLoadPointer decoded ptr res
    | Assembly.OpLoadPointerInc ptr res inc_ptr ⇒ @OpLoadPointerInc decoded ptr
```

```
res inc_ptr
    | Assembly.OpContextThis out => @OpContextThis decoded out
    | Assembly.OpContextCaller out => @OpContextCaller decoded out
    | Assembly.OpContextCodeAddress out => @OpContextCodeAddress decoded out
    | Assembly.OpContextMeta out => @OpContextMeta decoded out
    | Assembly.OpContextErgsLeft out => @OpContextErgsLeft decoded out
    | Assembly.OpContextSp out => @OpContextSp decoded out
    | Assembly.OpContextGetContextU128 out => @OpContextGetContextU128 decoded
out
    | Assembly.OpContextSetContextU128 in1 => @OpContextSetContextU128 decoded
in1
    | Assembly.OpContextSetErgsPerPubdataByte in1 =>
@OpContextSetErgsPerPubdataByte decoded in1
    | Assembly.OpContextIncrementTxNumber => @OpContextIncrementTxNumber decoded
    | Assembly.OpSLoad in1 out => @OpSLoad decoded in1 out
    | Assembly.OpSStore in1 in2 =>
        @OpSStore decoded in1 in2
    | Assembly.OpToL1Message in1 in2 is_first =>
        @OpToL1Message decoded in1 in2 is_first
    | Assembly.OpEvent in1 in2 is_first =>
        @OpEvent decoded in1 in2 is_first
    | Assembly.OpPrecompileCall in1 in2 out =>
        @OpPrecompileCall decoded in1 in2 out
    end.

End AssemblyToCore.

Module Coercions.
  Coercion to_core: Assembly.asm_instruction >-> CoreSet.instruction.
End Coercions.
```

Library EraVM.isa.AssemblyToMach

```
From RecordUpdate Require Import RecordSet.
Require GeneratedMachISA.
Require Addressing isa.Modifiers isa.Assembly Predication.
Import ssreflect.
Import RecordSetNotations.
Import Assembly Addressing Common GeneratedMachISA Modifiers TransientMemory
Pointer Predication.
Import Addressing.Coercions.
```

Encoding of `asm_instruction` to the universal instruction layout

In the lowest level, all instructions are encoded to a uniform 64-bit format; its fields are described by `mach_instruction`.

This file details the encoding of `asm_instruction` to `mach_instruction`.

```
Definition src_mode_of (op:in_any) :=
  match op with
  | InReg x ⇒ SrcReg
  | InImm x ⇒ SrcImm
  | InStack (StackInOnly (RelSpPop _ _)) ⇒ SrcSpRelativePop
  | InStack (StackInAny (Absolute _ _)) ⇒ SrcStackAbsolute
  | InStack (StackInAny (RelSP _ _)) ⇒ SrcSpRelative
  | InCode x ⇒ SrcCodeAddr
  | InConst x ⇒ SrcCodeAddr
  end
.
Definition src_special_mode_of (op:in_regimm) :=
  match op with
  | RegImmR x ⇒ SrcSpecialReg
  | RegImmI x ⇒ SrcSpecialImm
  end
.
Definition dst_mode_of (op:out_any) :=
  match op with
  | OutReg x ⇒ DstReg
  | OutStack (StackOutOnly (RelSpPush _ _)) ⇒ DstSpRelativePush
  | OutStack (StackOutAny (Absolute _ _)) ⇒ DstStackAbsolute
  | OutStack (StackOutAny (RelSP _ _)) ⇒ DstSpRelative
  end
.
#[local]
  Coercion src_mode_of: in_any >-> src_mode.
#[local]
  Coercion src_special_mode_of: in_regimm >-> src_special_mode.
#[local]
  Coercion dst_mode_of: out_any >-> dst_mode.
```

The Opcode field includes information such as modifiers and addressing modes.

```
Definition opcode_of (ins: asm_instruction) : mach_opcode :=
  let no_label := false in
  let with_label := true in
  let no_inc := false in
  let inc := true in
```

```
  match ins with
  | Assembly.OpInvalid ⇒ OpInvalid
  | Assembly.OpNoOp
  | Assembly.OpSpAdd _ _ ⇒ OpNoOp SrcReg DstSpRelativePush
  | Assembly.OpSpSub _ _ ⇒ OpNoOp SrcSpRelativePop DstReg
  | Assembly.OpJump dest ⇒ OpJump dest
  | Assembly.OpAnd src0 _ out sflags ⇒ OpAnd src0 out sflags
  | Assembly.OpOr src0 _ out sflags ⇒ OpOr src0 out sflags
  | Assembly.OpXor src0 _ out sflags ⇒ OpXor src0 out sflags
  | Assembly.OpAdd src0 _ out sflags ⇒ OpAdd src0 out sflags
  | Assembly.OpSub src0 src1 dst swap sflags ⇒ OpSub src0 dst swap sflags
  | Assembly.OpShl src0 src1 dst swap sflags ⇒ OpShl src0 dst swap sflags
  | Assembly.OpShr src0 src1 dst swap sflags ⇒ OpShr src0 dst swap sflags
  | Assembly.OpRol src0 src1 dst swap sflags ⇒ OpRol src0 dst swap sflags
  | Assembly.OpRor src0 src1 dst swap sflags ⇒ OpRor src0 dst swap sflags
  | Assembly.OpMul src0 _ dst0 _ sflags ⇒ OpMul src0 dst0 sflags
  | Assembly.OpDiv src0 _ dst0 _ swap sflags ⇒ OpDiv src0 dst0 swap sflags
  | Assembly.OpNearCall _ _ _ ⇒ OpCall
  | Assembly.OpFarCall enc dest handler is_static is_shard_provided ⇒ OpFarcall
is_static is_shard_provided
  | Assembly.OpMimicCall _ _ _ is_static is_shard_provided ⇒ OpMimic is_static
is_shard_provided
  | Assembly.OpDelegateCall _ _ _ is_static is_shard_provided ⇒ OpDelegate
is_static is_shard_provided
  | Assembly.OpNearRet ⇒ OpRet no_label
  | Assembly.OpNearRetTo _ ⇒ OpRet with_label
  | Assembly.OpFarRet _ ⇒ OpRet with_label
  | Assembly.OpNearRevert ⇒ OpRevert no_label
  | Assembly.OpNearRevertTo _ ⇒ OpRevert with_label
  | Assembly.OpFarRevert _ ⇒ OpRevert no_label
  | Assembly.OpNearPanicTo _ ⇒ OpPanic with_label
  | Assembly.OpPanic ⇒ OpPanic no_label
  | Assembly.OpPtrAdd src0 _ dst0 swap ⇒ OpPtrAdd src0 dst0 swap
  | Assembly.OpPtrSub src0 _ dst0 swap ⇒ OpPtrSub src0 dst0 swap
  | Assembly.OpPtrShrink src0 _ dst0 swap ⇒ OpPtrShrink src0 dst0 swap
  | Assembly.OpPtrPack src0 _ dst0 swap ⇒ OpPtrPack src0 dst0 swap
  | Assembly.OpLoad src0 dst0 Heap ⇒ OpLoadHeap src0 no_inc
  | Assembly.OpLoad src0 dst0 AuxHeap ⇒ OpLoadAuxHeap src0 no_inc
  | Assembly.OpLoadInc src0 dst0 Heap _ ⇒ OpLoadHeap src0 inc
  | Assembly.OpLoadInc src0 dst0 AuxHeap _ ⇒ OpLoadAuxHeap src0 inc
  | Assembly.OpStore src0 _ Heap ⇒ OpStoreHeap src0 no_inc
  | Assembly.OpStore src0 _ AuxHeap ⇒ OpStoreAuxHeap src0 no_inc
  | Assembly.OpStoreInc src0 _ Heap _ ⇒ OpStoreHeap src0 inc
  | Assembly.OpStoreInc src0 _ AuxHeap _ ⇒ OpStoreAuxHeap src0 inc
  | Assembly.OpLoadPointer _ _ ⇒ OpLoadPtr no_inc
  | Assembly.OpLoadPointerInc _ _ _ ⇒ OpLoadPtr inc
  | Assembly.OpContextThis _ ⇒ OpContextThis
  | Assembly.OpContextCaller _ ⇒ OpContextCaller
  | Assembly.OpContextCodeAddress _ ⇒ OpContextCodeAddress
  | Assembly.OpContextMeta _ ⇒ OpContextMeta
  | Assembly.OpContextErgsLeft _ ⇒ OpContextErgsLeft
  | Assembly.OpContextSp _ ⇒ OpContextSp
  | Assembly.OpContextGetContextU128 _ ⇒ OpContextGetContextU128
  | Assembly.OpContextSetContextU128 _ ⇒ OpContextSetContextU128
  | Assembly.OpContextSetErgsPerPubdataByte _ ⇒ OpContextSetErgsPerPubdataByte
  | Assembly.OpContextIncrementTxNumber ⇒ OpContextIncrementTxNumber
  | Assembly.OpSLoad _ _ ⇒ OpSload
```

```
  | Assembly.OpSStore _ _ ⇒ OpSstore
  | Assembly.OpPrecompileCall _ _ _ ⇒ OpLogPrecompile
  | Assembly.OpEvent _ _ is_first ⇒ OpLogEvent is_first
  | Assembly.OpToL1Message _ _ is_first ⇒ OpLogToL1 is_first
  end.

#[local]
Definition set_src0 (src0: in_any) : mach_instruction → mach_instruction :=
  fun ins ⇒
    match src0 with
    | InReg (Reg name) ⇒ ins <| op_src0 := Some name |>
    | InImm (Imm val) ⇒ ins <| op_imm0 := Some val |>
    | InStack (StackInOnly (RelSpPop reg ofs))
    | InStack (StackInAny (Absolute reg ofs) )
    | InStack (StackInAny (RelSP reg ofs) )
    | InCode (CodeAddr reg ofs)
    | InConst (ConstAddr reg ofs) ⇒ ins <| op_src0 := Some reg |> <| op_imm0 :=
Some ofs|>
    end
.

#[local]
Definition set_src0_special (src0: in_regimm) : mach_instruction →
mach_instruction :=
  fun ins ⇒
    match src0 with
    | RegImmR (Reg name) ⇒ ins <| op_src0 := Some name |>
    | RegImmI (Imm val) ⇒ ins <| op_imm0 := Some val |>
    end
.

#[local]
Definition set_dst0 (dst0: out_any) : mach_instruction → mach_instruction :=
  fun ins ⇒
    match dst0 with
    | OutReg (Reg name) ⇒ ins <| op_dst0 := Some name |>
    | OutStack (StackOutOnly (RelSpPush reg ofs))
    | OutStack (StackOutAny (Absolute reg ofs) )
    | OutStack (StackOutAny (RelSP reg ofs) ) ⇒
        ins <| op_dst0 := Some reg |> <| op_imm1 := Some ofs|>
    end
.

Section AsmToMachConversion.

  Import ssrfun.
```

The encoding of `asm_instruction` to `mach_instruction` happens in two stages:

1. Put the information in the fields of `mach_instruction` but keep ignored fields uninitialized (equal to `None`).
2. Flatten `mach_instruction`, erasing difference between meaningful and ignored fields. Fields that were equal to `None` are assigned default values: zero for immediates, `R0` for

registers.

```
  Definition asm_to_mach_opt (ins: predicated asm_instruction) : option
(@mach_instruction (option GPR.reg_name) (option u16) ):=
    match ins with
    | Ins ins pred ⇒
        let mk src0 src1 dst0 dst1 imm0 imm1 := mk_ins (opcode_of ins) pred
src0 src1 dst0 dst1 imm0 imm1 in
        let template : mach_instruction := mk None None None None None None in
        match ins with
        | Assembly.OpInvalid
        | Assembly.OpNoOp
        | Assembly.OpPanic
        | Assembly.OpContextIncrementTxNumber
          ⇒ Some template
        | Assembly.OpContextSetContextU128 (Reg src0)
        | Assembly.OpContextSetErgsPerPubdataByte (Reg src0) ⇒ Some (template
<| op_src0 := Some src0 |> )
        | Assembly.OpSpAdd (Reg reg) (Imm ofs)
        | Assembly.OpSpSub (Reg reg) (Imm ofs) ⇒ Some (template <| op_src0 :=
Some reg |> <| op_imm0 := Some ofs |> )
        | Assembly.OpJump dest ⇒ Some (set_src0 dest template)
        | Assembly.OpAdd src0 (Reg src1) dst0 _
        | Assembly.OpOr src0 (Reg src1) dst0 _
        | Assembly.OpXor src0 (Reg src1) dst0 _
        | Assembly.OpAnd src0 (Reg src1) dst0 _
        | Assembly.OpSub src0 (Reg src1) dst0 _ _
        | Assembly.OpShl src0 (Reg src1) dst0 _ _
        | Assembly.OpShr src0 (Reg src1) dst0 _ _
        | Assembly.OpRol src0 (Reg src1) dst0 _ _
        | Assembly.OpRor src0 (Reg src1) dst0 _ _
        | Assembly.OpPtrAdd src0 (Reg src1) dst0 _
        | Assembly.OpPtrSub src0 (Reg src1) dst0 _
        | Assembly.OpPtrShrink src0 (Reg src1) dst0 _
        | Assembly.OpPtrPack src0 (Reg src1) dst0 _
          ⇒ Some (set_src0 src0 (set_dst0 dst0 (template <| op_src1 := Some
src1 |>)))
        | Assembly.OpMul src0 (Reg src1) dst0 (Reg dst1) _
        | Assembly.OpDiv src0 (Reg src1) dst0 (Reg dst1) _ _
          ⇒ Some (set_src0 src0 (set_dst0 dst0 (template <| op_src1 := Some
src1 |>
                                                                    <|
op_dst1 := Some dst1 |>)))

        | Assembly.OpNearCall (Reg arg) (Imm dest) (Imm handler) ⇒
            Some ({|
                  op_code := opcode_of ins;
                  op_predicate := pred;
                  op_src0 := Some arg;
                  op_src1 := None;
                  op_dst0 := None;
                  op_dst1 := None;
                  op_imm0 := Some dest;
                  op_imm1 := Some handler;
```

```
                |})
        | Assembly.OpNearRet
        | Assembly.OpNearRevert => Some template
        | Assembly.OpNearRetTo (Imm dest)
        | Assembly.OpNearPanicTo (Imm dest)
        | Assembly.OpNearRevertTo (Imm dest) => Some (template <| op_imm0 :=
Some dest |>)
        | Assembly.OpFarRet (Reg args)
        | Assembly.OpFarRevert (Reg args) => Some (template <| op_src0 := Some
args |>)

        | Assembly.OpSStore (Reg src0) (Reg src1)
        | Assembly.OpEvent (Reg src0) (Reg src1) _
        | Assembly.OpToL1Message (Reg src0) (Reg src1) _ =>
            Some (template <| op_src0 := Some src0 |> <| op_src1 := Some src1
|>)
        | Assembly.OpSLoad (Reg src0) (Reg dst0) =>
            Some (template <| op_src0 := Some src0 |> <| op_dst0:= Some dst0|>)

        | Assembly.OpContextThis (Reg dst0)
        | Assembly.OpContextCaller (Reg dst0)
        | Assembly.OpContextCodeAddress (Reg dst0)
        | Assembly.OpContextMeta (Reg dst0)
        | Assembly.OpContextErgsLeft (Reg dst0)
        | Assembly.OpContextSp (Reg dst0)
        | Assembly.OpContextGetContextU128 (Reg dst0) =>
            Some (template <| op_dst0:= Some dst0|>)

        | Assembly.OpPrecompileCall (Reg src0) (Reg src1) (Reg dst0) =>
            Some (template <| op_dst0:= Some dst0|>)

        | Assembly.OpFarCall (Reg params) (Reg dest) (Imm handler) _ _
        | Assembly.OpMimicCall (Reg params) (Reg dest) (Imm handler) _ _
        | Assembly.OpDelegateCall (Reg params) (Reg dest) (Imm handler) _ _ =>
            Some (template
                    <| op_src0 := Some params |>
                                <| op_src1 := Some dest |>
                                            <| op_imm0 := Some handler
|> )

        | Assembly.OpLoad ptr (Reg res) _
        | Assembly.OpStore ptr (Reg res) _ =>
            Some (set_src0_special ptr (template <| op_dst0 := Some res|> ))
        | Assembly.OpLoadPointer (Reg name) (Reg res) =>
            Some (template <| op_src0 := Some name |> <| op_dst0 := Some res|>
)
        | Assembly.OpLoadInc ptr (Reg res) _ (Reg inc_ptr)
        | Assembly.OpStoreInc ptr (Reg res) _ (Reg inc_ptr) =>
            Some (set_src0_special ptr (template <| op_dst0 := Some res|> <|
op_dst1 := Some inc_ptr |>))

        | Assembly.OpLoadPointerInc (Reg ptr) (Reg res) (Reg inc_ptr) =>
            Some (template <| op_src0 := Some ptr |> <| op_dst0 := Some res|>
<| op_dst1 := Some inc_ptr |>)
        end
    end.
```

```
  Definition mach_flatten (i:@mach_instruction (option GPR.reg_name) (option
u16)) : @mach_instruction GPR.reg_name u16 :=
    let or_r0 := Option.default GPR.R0 in
    let or_zero := Option.default zero16 in
    match i with
    | mk_ins op_code op_predicate op_src0 op_src1 op_dst0 op_dst1 op_imm0
op_imm1 ⇒
        mk_ins op_code op_predicate
          (or_r0 op_src0)
          (or_r0 op_src1)
          (or_r0 op_dst0)
          (or_r0 op_dst1)
          (or_zero op_imm0)
          (or_zero op_imm1)
    end.

  Definition asm_to_mach (asm_ins: predicated asm_instruction) : option
mach_instruction :=
   option_map mach_flatten (asm_to_mach_opt asm_ins).

End AsmToMachConversion.
```

Library EraVM.Ergs

```
Require Common isa.Assembly TransientMemory.
Import Common Assembly TransientMemory ZArith.

Section Ergs.
Open Scope Z_scope.
```

Ergs

Ergs is the resource spent on executing actions in EraVM.

The most common action consuming ergs is executing an instruction. Instructions have a fixed `base_cost`, failure to pay this cost results in panic.

In EraVM, the instructions are `predicated`. If an instruction is not executed because of mismatch between their predicate `ins_cond` with the current `gs_flags`, its base cost is still paid. Therefore, it is cheaper to jump over expensive instructions like `OpFarCall` than to predicate them so that they are not executed.

Additionally, actions like decommitting code for execution, accessing contract storage, or growing memory bounds, also cost ergs.

Internally, ergs are 32-bit unsigned numbers.

```
Definition ergs_bits := 32%nat.
Definition ergs := BITS ergs_bits.
```

```
Definition ergs_of : Z → ergs := fromZ.
```

Ergs and callstack

Every frame in `callstack`, whether external or internal, keeps its associated ergs in the field `cf_ergs_remaining`. Spending ergs decreases this value `cf_ergs_remaining`.

Calls

Calling functions/contracts requires passing ergs to the new calling frame, so that the callee's code would be able to operate and spend ergs (see e.g. `step_nearcall`).

For far calls, it is not possible to pass more than `max_passable` ergs (currently 63/64 of ergs available in current frame). For near calls, passing 0 ergs leads to passing all ergs in the current frame.

Returns

If a function returns without panic, the remaining ergs are returned to its parent frame i.e. added to the parent frame's `cf_ergs_remaining`.

If a function panics, all its ergs are burned (see `sem.Panic.step_panic`). Panic does not burn ergs of parent frames.

The following return instructions lead to returning remaining ergs to the caller:

- `OpNearRet`
- `OpNearRetTo`
- `OpFarRet`
- `OpNearRevert`
- `OpNearRevertTo`
- `OpFarRevert`
- `OpNearPanicTo`

Actions consuming ergs

Each instruction has a fixed based cost that gets deducted before executing it (see `base_cost`).

Additionally, the following actions lead to spending ergs:

1. Decommitting contract code. Performing far call to a contract which was not called during the construction of the current block costs ergs per each word of contract code. See `Decommitter`, `FarCall`.
2. TODO Accessing storage
3. Memory growth. Data pages holding heap variants are bounded, and only accesses to addresses within these bounds are free. Reading or writing to these pages outside bounds forces the **memory growth** with bound adjustment. The number of bytes by which the bounded area has grown has to be paid; see `grow_and_pay`.
4. Passing messages to L1 by `OpToL1Message`.

Burning ergs

Burning ergs refers to a situation of panic, when the topmost `callstack` frame is destroyed with its allocated ergs. The general rule is: if some invariant of execution breaks, VM panics, burning all ergs in the current frame. This is a fail-fast behavior in case of irrecoverable errors. Some examples are:

- using an integer value where pointer value is expected
- executing kernel-only instruction in user mode
- call stack overflow

Some situations that provoke panic are:

- having not enough ergs to pay, e.g. for memory growth;
- attempting to execute an instruction with an invalid encoding;
- attempting to execute kernel-only instruction in user mode.

See section `Panics` for the full description.

Parameters

The following definitions are used to derive the costs of instructions and other actions.

```
Definition VM_CYCLE_COST_IN_ERGS: Z := 4.
Definition RAM_PERMUTATION_COST_IN_ERGS: Z := 1.
Definition CODE_DECOMMITMENT_COST_PER_WORD_IN_ERGS: Z := 4.
Definition STORAGE_APPLICATION_COST_IN_ERGS: Z := 678.
Definition CODE_DECOMMITTER_SORTER_COST_IN_ERGS: Z := 1.
Definition LOG_DEMUXER_COST_IN_ERGS: Z := 1.
Definition STORAGE_SORTER_COST_IN_ERGS: Z := 2.
Definition EVENTS_OR_L1_MESSAGES_SORTER_COST_IN_ERGS: Z := 1.
Definition INITIAL_WRITES_PUBDATA_HASHER_COST_IN_ERGS: Z := 18.
Definition REPEATED_WRITES_PUBDATA_HASHER_COST_IN_ERGS: Z := 11.
Definition CODE_DECOMMITMENT_SORTER_COST_IN_ERGS: Z := 1.

Definition L1_MESSAGE_MIN_COST_IN_ERGS: Z := 156250.
Definition INITIAL_WRITES_PUBDATA_HASHER_MIN_COST_IN_ERGS: Z := 0.
Definition REPEATED_WRITES_PUBDATA_HASHER_MIN_COST_IN_ERGS: Z := 0.

Definition STORAGE_WRITE_HASHER_MIN_COST_IN_ERGS: Z := 0.

Definition KECCAK256_CIRCUIT_COST_IN_ERGS: Z := 40.
Definition SHA256_CIRCUIT_COST_IN_ERGS: Z := 7.
Definition ECRECOVER_CIRCUIT_COST_IN_ERGS: Z := 1112.

Definition INVALID_OPCODE_ERGS: Z := unsigned_max 32.

Definition RICH_ADDRESSING_OPCODE_ERGS: Z
  := VM_CYCLE_COST_IN_ERGS + 2 × RAM_PERMUTATION_COST_IN_ERGS.
Definition AVERAGE_OPCODE_ERGS: Z
  := VM_CYCLE_COST_IN_ERGS + RAM_PERMUTATION_COST_IN_ERGS.
```

```
Definition STORAGE_READ_IO_PRICE: Z := 150.
Definition STORAGE_WRITE_IO_PRICE: Z := 250.
Definition EVENT_IO_PRICE: Z := 25.
Definition L1_MESSAGE_IO_PRICE: Z := 100.
Definition CALL_LIKE_ERGS_COST: Z := 20.
Definition ERGS_PER_CODE_WORD_DECOMMITTMENT: Z :=
CODE_DECOMMITMENT_COST_PER_WORD_IN_ERGS.

Definition DECOMMITMENT_MSG_VALUE_SIMULATOR_OVERHEAD: Z := 64000.
Definition MSG_VALUE_SIMULATOR_ADDITIVE_COST: Z := 11500 +
DECOMMITMENT_MSG_VALUE_SIMULATOR_OVERHEAD.
Definition MSG_VALUE_SIMULATOR_MIN_USED_ERGS: Z := 8000 +
DECOMMITMENT_MSG_VALUE_SIMULATOR_OVERHEAD.

Definition MIN_STORAGE_WRITE_PRICE_FOR_REENTRANCY_PROTECTION: Z := Z.max
                                                                  (MSG_VALUE_SIMULATOR_ADDITIVE_COST
- MSG_VALUE_SIMULATOR_MIN_USED_ERGS + 1)
                                                                  (2300 + 1).
Definition MIN_STORAGE_WRITE_COST: Z := Z.max
                                        MIN_STORAGE_WRITE_PRICE_FOR_REENTRANCY_PROTECTION
                                        STORAGE_WRITE_HASHER_MIN_COST_IN_ERGS.

Definition INITIAL_STORAGE_WRITE_PUBDATA_BYTES: Z := 64.
Definition REPEATED_STORAGE_WRITE_PUBDATA_BYTES: Z := 40.
Definition L1_MESSAGE_PUBDATA_BYTES: Z := (1 + 1 + 2 + 20 + 32 + 32).

Definition growth_cost (diff:mem_address) : ergs := diff.
End Ergs.
```

```
Section Costs.
  Open Scope Z_scope.
```

Costs

Basic costs of all instructions. They are paid when the instruction starts executing; see `Semantics.step`.

Instructions may also impose additional costs e.g. far returns and far calls may grow heap; far calls also may induce code `decommitment_cost`.

```
  Definition base_cost (ins:asm_instruction) :=
    (match ins with
     | OpInvalid ⇒ INVALID_OPCODE_ERGS
     | OpNoOp | OpSpAdd _ _ | OpSpSub _ _ ⇒ RICH_ADDRESSING_OPCODE_ERGS
     | OpJump _ ⇒ RICH_ADDRESSING_OPCODE_ERGS
     | OpAnd _ _ _ _ ⇒ RICH_ADDRESSING_OPCODE_ERGS
     | OpOr _ _ _ _ ⇒ RICH_ADDRESSING_OPCODE_ERGS
     | OpXor _ _ _ _ ⇒ RICH_ADDRESSING_OPCODE_ERGS
     | OpAdd _ _ _ _ ⇒ RICH_ADDRESSING_OPCODE_ERGS
     | OpSub _ _ _ _ _ ⇒ RICH_ADDRESSING_OPCODE_ERGS

     | OpShl _ _ _ _ _ ⇒ RICH_ADDRESSING_OPCODE_ERGS
```

```
      | OpShr _ _ _ _ _ => RICH_ADDRESSING_OPCODE_ERGS
      | OpRol _ _ _ _ _ => RICH_ADDRESSING_OPCODE_ERGS
      | OpRor _ _ _ _ _ => RICH_ADDRESSING_OPCODE_ERGS

      | OpMul _ _ _ _ _ => RICH_ADDRESSING_OPCODE_ERGS
      | OpDiv _ _ _ _ _ _ => RICH_ADDRESSING_OPCODE_ERGS
      | OpNearCall _ _ _ => AVERAGE_OPCODE_ERGS + CALL_LIKE_ERGS_COST
      | OpFarCall _ _ _ _ _
      | OpDelegateCall _ _ _ _ _
      | OpMimicCall _ _ _ _ _ => 2 × VM_CYCLE_COST_IN_ERGS
                               + RAM_PERMUTATION_COST_IN_ERGS
                               + STORAGE_READ_IO_PRICE
                               + CALL_LIKE_ERGS_COST
                               + STORAGE_SORTER_COST_IN_ERGS
                               + CODE_DECOMMITMENT_SORTER_COST_IN_ERGS

      | OpNearRet | OpNearRetTo _ | OpNearRevert | OpNearRevertTo _ |
OpNearPanicTo _
      | OpFarRet _ | OpFarRevert _
      | OpPanic
        => AVERAGE_OPCODE_ERGS
      | OpPtrAdd _ _ _ _
      | OpPtrSub _ _ _ _
      | OpPtrShrink _ _ _ _
      | OpPtrPack _ _ _ _ => RICH_ADDRESSING_OPCODE_ERGS
      |
        OpStore _ _ _
      | OpStoreInc _ _ _ _
        => 2 × VM_CYCLE_COST_IN_ERGS + 5 × RAM_PERMUTATION_COST_IN_ERGS

    | OpLoad _ _ _
    | OpLoadInc _ _ _ _
    | OpLoadPointer _ _
    | OpLoadPointerInc _ _ _
      => VM_CYCLE_COST_IN_ERGS + 3 × RAM_PERMUTATION_COST_IN_ERGS

    | OpContextThis _
    | OpContextCaller _
    | OpContextCodeAddress _
    | OpContextMeta _
    | OpContextErgsLeft _
    | OpContextSp _
    | OpContextGetContextU128 _
    | OpContextSetContextU128 _
    | OpContextSetErgsPerPubdataByte _
    | OpContextIncrementTxNumber => AVERAGE_OPCODE_ERGS
    | OpSLoad _ _=> STORAGE_READ_IO_PRICE
                  + VM_CYCLE_COST_IN_ERGS
                  + RAM_PERMUTATION_COST_IN_ERGS
                  + LOG_DEMUXER_COST_IN_ERGS
                  + STORAGE_SORTER_COST_IN_ERGS
    | OpSStore _ _ =>
                 Z.max MIN_STORAGE_WRITE_COST (
                     STORAGE_WRITE_IO_PRICE
                     + 2 × VM_CYCLE_COST_IN_ERGS
                     + RAM_PERMUTATION_COST_IN_ERGS
                     + 2 × LOG_DEMUXER_COST_IN_ERGS
```

```
                  + 2 × STORAGE_SORTER_COST_IN_ERGS)
    | OpToL1Message _ _ _ ⇒
              let intrinsic_cost := L1_MESSAGE_IO_PRICE
                  + 2 × VM_CYCLE_COST_IN_ERGS
                  + RAM_PERMUTATION_COST_IN_ERGS
                  + 2 × LOG_DEMUXER_COST_IN_ERGS
                  + 2 × EVENTS_OR_L1_MESSAGES_SORTER_COST_IN_ERGS in
              Z.max intrinsic_cost L1_MESSAGE_MIN_COST_IN_ERGS
    | OpEvent _ _ _ ⇒ EVENT_IO_PRICE
                  + 2 × VM_CYCLE_COST_IN_ERGS
                  + RAM_PERMUTATION_COST_IN_ERGS
                  + 2 × LOG_DEMUXER_COST_IN_ERGS
                  + 2 × EVENTS_OR_L1_MESSAGES_SORTER_COST_IN_ERGS
    | OpPrecompileCall _ _ _ ⇒
       VM_CYCLE_COST_IN_ERGS + RAM_PERMUTATION_COST_IN_ERGS +
LOG_DEMUXER_COST_IN_ERGS
    end)%Z.
```

Implementation note: Coq allows partially evaluating `base_cost` to get the absolute erg costs for each instruction:

```
Compute base_costs.
```

Current costs are:

```
| Invalid => 4294967295
| NearCall => 25
| FarCall
| MimicCall
| DelegateCall => 182
| Store
| StoreInc => 13
| Load
| LoadInc
| LoadPointer
| LoadPointerInc => 7
| NearRet
| NearRetTo
| FarRet
| NearRevert
| NearRevertTo
| FarRevert
| NearPanicTo
| Panic
| ContextThis
| ContextCaller
| ContextCodeAddress
| ContextMeta
| ContextErgsLeft
| ContextSp
| ContextGetContextU128
| ContextSetContextU128
| ContextSetErgsPerPubdataByte
| ContextIncrementTxNumber => 5
```

```
| SLoad => 158
| SStore => 3501
| ToL1Message => 156250
| Event => 38
| <otherwise> => 6
```

```
End Costs.
```

Library EraVM.KernelMode

```
Require isa.CoreSet memory.Depot.

Import CoreSet TransientMemory memory.Depot.

Section KernelMode.
  Import ZArith Arith spec.
  Open Scope Z_scope.
  Open Scope ZMod_scope.

  Context {descr: CoreSet.descr}.
```

Kernel Mode

EraVM operates either in **kernel** or in **user mode**. Some instructions (see `requires_kernel` are only allowed in kernel mode; executing them in user mode results in panic.

Current mode is determined by the address of the currently executed contract C:

- if $C <$ `KERNEL_MODE_MAXADDR_LIMIT`, EraVM is in kernel mode;
- otherwise, EraVM is in user mode.

```
Definition KERNEL_MODE_MAXADDR_LIMIT : contract_address := fromZ (2^16).

Definition addr_is_kernel (addr:contract_address) : bool :=
  addr < KERNEL_MODE_MAXADDR_LIMIT.
```

Current contract's address can be obtained from the active external frame in `callstack`. Topmost external frame (active frame) is obtained through `active_extframe`, it contains the current contract's address in its field `ecf_this_address`.

The list of instructions requiring kernel mode is encoded by the definition `requires_kernel`. If `requires_kernel ins == true`, the instruction `ins` is only allowed in kernel mode.

```
  Definition requires_kernel (ins: @instruction descr) : bool :=
    match ins with
    | OpMimicCall _ _ _ _ _
    | OpContextSetContextU128 _
    | OpContextSetErgsPerPubdataByte _
    | OpContextIncrementTxNumber
    | OpEvent _ _ _
    | OpToL1Message _ _ _
    | OpPrecompileCall _ _ _
      ⇒ true
    | _ ⇒ false
    end.
```

Function `check_requires_kernel` returns `false` if:

- an instruction `ins` requires kernel mode, and
- VM is not in kernel mode, as indicated by `in_kernel`.

```
  Definition check_requires_kernel
    (ins: @instruction descr)
    (in_kernel: bool) : bool :=
    (negb in_kernel) || in_kernel.

End KernelMode.
```

Library EraVM.StaticMode

```
Require isa.CoreSet.

Import isa.CoreSet Addressing GPR Common.
Section StaticMode.
```

Static mode

Static mode is a mode of execution.

Intuitively, executing code in static mode aims at limiting its effects on the global state, similar to executing pure functions.

If VM is in static mode, attempting to execute instructions affecting global system state results in panic with reason `ForbiddenInStaticMode`. Refer to `forbidden_static` for a full list of instructions forbidden in static mode.

Current mode is determined by the flag `ecf_is_static` in the `active_extframe` in `gs_callstack`.

Entering static mode

To execute the code of a contract in static mode, use one of far call instructions with a static modifier, for example:

```
OpFarCall (Reg R1) (Reg R2) (Imm zero16) true false
                                         ^ is_static
```

The same applies to `OpMimicCall` and `OpDelegateCall`.

Exiting static mode

If VM executes a contract C in static mode, the mode will persist until the end of execution of C. If C calls itself or other contracts, these calls will be automatically marked as static, even if the far call instruction was not explicit about it.

There is no other way to exit the static mode.

Usage

Static mode is unrelated and orthogonal to kernel mode.

Executing a contract C in static mode restricts the changes to the state produced by C or any other code that it might call.

Static calls are guaranteed to preserve the state of storage, will not emit events, or modify the `gs_context_u128` register.

Function `forbidden_static` returns `true` if instruction `ins` is forbidden in static mode.

```
Context (forbidden := true) (allowed := false) {descr:descr}.
Definition forbidden_static (ins:@instruction descr) : bool :=
  match ins with
  | OpContextSetContextU128 _
  | OpContextSetErgsPerPubdataByte _
  | OpContextIncrementTxNumber
  | OpSStore _ _
  | OpEvent _ _ _
  | OpToL1Message _ _ _
    ⇒ forbidden
  | _ ⇒ allowed
  end.
```

Function `check_forbidden_static` returns `false` if:

- an instruction `ins` is not allowed in static mode, and
- the current mode is static, as indicated by `static_mode_active`.

```
  Definition check_forbidden_static
    (ins: instruction)
    (static_mode_active: bool) : bool :=
    if static_mode_active
    then negb (forbidden_static ins)
    else true.

End StaticMode.
```

Library EraVM.Steps

```
From RecordUpdate Require Import RecordSet.

Require
  Flags
    isa.CoreSet
    KernelMode
    State.
Import
  Flags
    isa.CoreSet
    KernelMode
    RecordSetNotations
    State.
```

A type of a small step relation, in style of structural operational semantics.

```
Definition smallstep := state → state → Prop .
Definition tsmallstep := transient_state → transient_state → Prop.
Definition flags_tsmallstep := flags_state → flags_state → Prop.
Definition callstack_smallstep := callstack → callstack → Prop.
```

Relations `step_transient_only` and `step_transient` define the type of small steps only affecting `gs_transient` part.

```
Inductive step_transient_only (xs1 xs2:transient_state) : smallstep :=
| stransient_oapply:
  ∀ gs,
    step_transient_only xs1 xs2 {|
                          gs_transient := xs1;
```

```
    gs_global := gs;
  |}
  {|
    gs_transient := xs2;
    gs_global := gs;
  |}.

Inductive step_transient (S: transient_state → transient_state → Prop) :
smallstep :=
| stransient_apply:
  ∀ xs1 xs2 s1 s2 ,
    S xs1 xs2 →
    step_transient_only xs1 xs2 s1 s2→
    step_transient S s1 s2.
```

Relations `tstep_flags` and `step_transient_callstack` help defining smallstep relations where only flags or callstack are changing.

```
Definition tstep_flags {descr:CoreSet.descr} (P: @instruction descr →
flags_tsmallstep): @instruction descr → tsmallstep :=
  fun i xs xs' ⇒ ∀ f2, P i (gs_flags xs) f2 → xs' = xs <| gs_flags := f2 |>.
```

Relations `step_transient_callstack` and help defining smallstep relations where only callstack is changing.

```
Inductive step_transient_callstack (S: callstack → callstack → Prop) :
transient_state → transient_state → Prop :=
| scs_apply:
  ∀ flags regs pages ctx cs1 cs2 xs1 xs2 status,
    S cs1 cs2 →
    xs1 = {|
            gs_callstack := cs1;

            gs_flags := flags;
            gs_regs := regs;
            gs_pages := pages;
            gs_context_u128 := ctx;
            gs_status := status;
          |} →
    xs2 = {|
            gs_callstack := cs2;


            gs_flags := flags;
            gs_regs := regs;
            gs_pages := pages;
            gs_context_u128 := ctx;
            gs_status := status;
          |} →

    step_transient_callstack S xs1 xs2.
```

```
Inductive step_callstack (S: callstack → callstack → Prop) : smallstep :=
| sc_apply: ∀ xs1 xs2 s1 s2,
    step_transient_callstack S xs1 xs2 →
    step_transient_only xs1 xs2 s1 s2 →
    step_callstack S s1 s2.
```

Library EraVM.Semantics

```
From RecordUpdate Require Import RecordSet.
Require VMPanic StaticMode isa.AssemblyToCore sem.SemanticCommon.


Section VMParameters.
  Local Open Scope ZMod_scope.

  Definition VM_INITIAL_FRAME_ERGS: nat := Z.to_nat (unsigned_max ergs_bits).
  Context (CALL_LIKE_ERGS_COST := Z.to_nat CALL_LIKE_ERGS_COST).
  Definition VM_MAX_STACK_DEPTH: nat := VM_INITIAL_FRAME_ERGS /
CALL_LIKE_ERGS_COST + 80.
End VMParameters.

Section SmallStep.
  Local Open Scope ZMod_scope.

  Context (ins := @instruction bound).
  Definition update_pc_regular : callstack → callstack :=
    pc_map (fun x => uadd_wrap x # 1).
```

Every instruction is either executed, skipped, or triggers panic instantly. Panic can also be triggered later during the execution.

```
  Inductive action: Type := Execute | Skip | Panic : reason → action.
```

After the instruction is selected, panic is immediately triggered:

- on call stack overflow;
- if the `base_cost` of instruction is unaffordable;
- if the instruction is not allowed in user mode, and VM is in user mode;
- if the instruction is not allowed in static mode, and VM is in static mode.

```
  Definition chose_action (s:transient_state) (i:@predicated asm_instruction) :
action :=
    if stack_overflow VM_MAX_STACK_DEPTH (gs_callstack s) then
      Panic CallStackOverflow
    else
      if negb (check_requires_kernel i.(ins_spec _) (in_kernel_mode
(gs_callstack s))) then
```

```
        Panic NotInKernelMode
      else
        if negb (check_forbidden_static i.(ins_spec _) (active_extframe
(gs_callstack s)).(ecf_is_static)) then
          Panic ForbiddenInStaticMode
        else
          if ergs_of (base_cost i.(ins_spec _)) > ergs_remaining (gs_callstack
s) then
            Panic NotEnoughErgsToPayBaseCost
          else
            if negb (predicate_holds i.(ins_cond _) (gs_flags s)) then
              Skip
            else Execute.
```

The definition `smallsteps` gathers the references to all the small step predicates for various `asm_instruction`s.

```
  Definition smallsteps : list (@instruction bound → smallstep) :=
  [
                                      step_nop ;
  fun i ⇒ step_transient (tstep_flags step_add i);
  fun i ⇒ step_transient (tstep_flags step_sub i);
  fun i ⇒ step_transient (tstep_flags step_and i);
                                      step_context;
  fun i ⇒ step_transient (tstep_flags step_mul i);
  fun i ⇒ step_transient (tstep_flags step_div i);
  fun i ⇒ step_transient ( step_farret i);
                                      step_farrevert ;
                                      step_farcall ;
                                      step_jump;
  fun i ⇒ step_transient ( step_load i);
  fun i ⇒ step_transient ( step_load_inc i);
  fun i ⇒ step_transient ( step_load_ptr i);
  fun i ⇒ step_transient ( step_load_ptr_inc i);
  fun i ⇒ step_transient (tstep_flags step_mul i);
  fun i ⇒ step_callstack ( step_sp_add i);
  fun i ⇒ step_callstack ( step_sp_sub i);
                                      step_nearcall ;
                                      step_panicto ;
  fun i ⇒ step_transient ( step_nearret i);
  fun i ⇒ step_transient ( step_nearretto i);
                                      step_nearrevert ;
                                      step_nearrevertto ;
                                      step_event ;
  fun i ⇒ step_transient (tstep_flags step_or i);
                                      step_oppanic ;
                                      step_precompile ;
                                      step_ptradd ;
                                      step_ptrsub ;
                                      step_ptrshrink ;
                                      step_ptrpack ;
  fun i ⇒ step_transient (tstep_flags step_rol i);
  fun i ⇒ step_transient (tstep_flags step_ror i);
                                      step_sload ;
```

```
                                        step_sstore ;
fun i => step_transient (tstep_flags step_shl i);
fun i => step_transient (tstep_flags step_shr i);
fun i => step_transient ( step_store i);
fun i => step_transient ( step_storeinc i);
                                        step_tol1 ;
fun i => step_transient (tstep_flags step_xor i)

].

Inductive dispatch: @instruction bound → smallstep :=
| dispatch_apply: ∀ s1 s2 ins S,
    In S smallsteps →
    S ins s1 s2 →
    dispatch ins s1 s2.

Generalizable Variables cs.

Inductive execute_action: action → @instruction decoded → smallstep :=
| ea_execute:
  `(∀ instr gs instr_bound new_s xs0 xs1,
        cs0 = gs_callstack xs0 →

        cs1 = update_pc_regular cs0 →
        pay (ergs_of (base_cost instr)) cs1 cs2 →
        bind_operands (xs0 <| gs_callstack := cs2 |>) xs1 instr instr_bound →
        let s1 := mk_state xs1 gs in
        dispatch instr_bound s1 new_s →
        execute_action Execute instr (mk_state xs0 gs) new_s
    )
  | ea_skip:
    `(∀ instr gs xs0 xs1,
          cs0 = gs_callstack xs0 →

          cs1 = update_pc_regular cs0 →
          pay (ergs_of (base_cost instr)) cs1 cs2 →
          let new_s := mk_state xs1 gs in
          execute_action Skip instr (mk_state xs0 gs) new_s
    )
  | ea_panic:
    ∀ reason s new_s instr,
      step_panic reason s new_s →
      execute_action (Panic reason) instr s new_s
  .

  Generalizable No Variables.

  Definition fetch_predicated_instruction (s: transient_state) ins :=
    @fetch_instr _ instruction_invalid _ (gs_regs s) (gs_callstack s) (gs_pages
s)
      (@FetchIns (predicated asm_instruction) ins).
```

`step` is the main predicate defining a VM transition in a small step structural operational style.

```
  Inductive step: smallstep :=
  | step_correct:
    ∀ (s new_s : state) cond
      (instr:asm_instruction)
      (ins_bound:@instruction bound),

      fetch_predicated_instruction s (Ins _ instr cond) →
      execute_action (chose_action s (Ins _ instr cond)) instr s new_s →
      step s new_s.
End SmallStep.
```

Library EraVM.VMPanic

```
Require Common isa.CoreSet.
Import Common isa.CoreSet.

Section Panics.
```

Panic

Panic refers to a situation of irrecoverable error. It can occur for one of the following reasons:

- There are not enough ergs to execute an action.
- Executing an instruction requiring kernel mode in user mode.
- Executing an instruction mutating global state in static mode.
- Violation of one of VM inner invariants.
- Overflow of callstack.
- Attempt to execute an invalid instruction.
- Providing an integer value (with the tag cleared) instead of a pointer value (with the tag set) to an instruction that expects a tagged fat pointer value, e.g. `OpPtrAdd`.

Complete list of panic reasons

The type `reason` describes all situations where EraVM panics.

Note this is an exhaustive list! If there is a panic situation which does not match to any condition described by `reason`, please, report it to Igor `iz@matterlabs.dev`.

```
Inductive reason :=
```

- See `step_RetExt_ForwardFatPointer_requires_ptrtag`.

```
| RetABIExistingFatPointerWithoutTag
```

- See `step_RetExt_ForwardFatPointer_returning_older_pointer`.

| `RetABIReturnsPointerCreatedByCaller`

- Malformed `fat_ptr` is such that `validate` returns `false`.

| `FatPointerMalformed`

- See `step_PtrAdd_overflow` and `step_PtrSub_underflow`.

| `FatPointerOverflow`

- See `step_PtrAdd_diff_too_large` and `step_PtrSub_diff_too_large`.

| `FatPointerDeltaTooLarge`

- See e.g. `step_RetExt_heapvar_growth_unaffordable`

| `FatPointerCreationUnaffordable`

- See `chose_action`.

| `NotInKernelMode`

- See `chose_action`.

| `ForbiddenInStaticMode`

- See `chose_action`.

| `CallStackOverflow`

- See `step_PtrPack_notzero`.

| `PtrPackExpectsOp2Low128BitsZero`

- Instruction expects a tagged `fat_ptr`, e.g. `step_PtrAdd_in1_not_ptr`.

| `ExpectedFatPointer`

- Instruction expects a non-tagged primitive value, e.g. `step_PtrAdd_in2_ptr`.

| `ExpectedInteger`

- Executing `OpPanic` or `OpNearPanicTo`.

| `TriggeredExplicitly`

- Attempt to dereference a pointer past `MAX_OFFSET_TO_DEREF_LOW_U32`.

| `HeapPtrOffsetTooLarge`

- Not enough ergs to pay for growing heap beyound the current bound.

| `HeapGrowthUnaffordable`

- Incrementing a heap pointer by executing `OpLoadInc` or `OpStoreInc` results in overflow.

| `HeapPtrIncOverflow`

- Incrementing a fat pointer by executing `OpLoadPtrInc` results in overflow.

| `FatPtrIncOverflow`

- Instructions e.g. `OpLoad` expect a heap pointer with a `is_ptr` tag reset, not a fat pointer with a set tag.

| `ExpectedHeapPointer`

- Not enough ergs to pay `base_cost` of instruction. Reminder: the cost is paid even if instruction is skipped by predication.

| `NotEnoughErgsToPayBaseCost`

- Far call expects to return an existing `fat_ptr` but the provided value is not tagged as pointer.

| `FarCallInputIsNotPointerWhenExpected`

- Far call expects the storage of `DEPLOYER_SYSTEM_CONTRACT_ADDRESS` to hold a valid `versioned_hash` for the callee contract, but the hash for the callee contract is malformed.

| `FarCallInvalidCodeHashFormat`

- In a far call, not enough ergs to pay for code decommitment.

| `FarCallNotEnoughErgsToDecommit`

- In a far call, attempt to return a new `fat_ptr` but it requires growing heap and there are not enough ergs to pay for this growth.

| `FarCallNotEnoughErgsToGrowMemory`

- Trying to far call a contract whose code is not yet deployed.

| `FarCallCallInNowConstructedSystemContract`

- Attempt to write to a storage by executing `OpSStore` but not enough ergs to pay the associated cost. See `step_SStore_unaffordable`.

```
  | StorageWriteUnaffordable
  .

  Inductive status :=
  | NoPanic
  | Panic : reason → status
  .

End Panics.
```

Library EraVM.sem.SemanticCommon

```
From RecordUpdate Require Import RecordSet.

Require
ABI
Addressing
Binding
CallStack
Common
Flags
KernelMode
MemoryContext
MemoryOps
State
Steps
TransientMemory
VMPanic
sem.StepPanic
.

Import ssreflect ssrfun ssrbool eqtype ssreflect.tuple.

Import
  Addressing
    Bool
    Common
    Coder
    Core
    Flags
    CallStack
    Decommitter
    Ergs
    GPR
    List
    ListNotations
```

```
    KernelMode
    MemoryContext
    memory.Depot
    MemoryBase
    MemoryOps
    Pointer
    PrimitiveValue
    RecordSetNotations
    State
    TransientMemory
    ZArith
    ZBits.
Export Steps Binding VMPanic StepPanic.

Section Params.
  Open Scope ZMod_scope.
  Definition MAX_OFFSET_TO_DEREF_LOW_U32: u32 := fromZ (2^32 - 33)%Z.
  Definition MAX_OFFSET_FOR_ADD_SUB: u256 := fromZ (2^32)%Z.
End Params.

Section Depot.
  Definition is_rollup (xstack: callstack) : bool := zero8 == current_shard
xstack.
  Definition net_pubdata cs : Z := if is_rollup cs then
INITIAL_STORAGE_WRITE_PUBDATA_BYTES else 0%Z.

End Depot.

Definition current storage fqa (xstack:callstack) : fqa storage :=
  mk_fqa_storage (current_shard xstack) (current_contract xstack).

(* FIXME *)
Local Open Scope ZMod_scope.
Definition bitwise_flags (result: Core.word) : Flags.flags_state :=
Flags.bflags false (result == zero256) false.

Definition topmost_128_bits_match (x y : Core.word) : Prop := @high 128 128 x =
@high 128 128 y.
```

Library EraVM.sem.Nop

```
Require SemanticCommon.

Import Core isa.CoreSet State SemanticCommon.

Section NoOpDefinition.
```

Nop

Abstract Syntax

`OpNoOp`

Syntax

`nop`

Summary

Do nothing.

Affected parts of VM state

- execution stack : PC is increased.

Similar instructions

`OpSpAdd`, `OpSpSub` and `OpNoOp` are translated to `mach_instruction` with the same `mach_opcode`. See `asm_to_mach`.

Encoding

- `NoOp`, `SpAdd`, `SpSub` are encoded as the same instruction.

```
  Inductive step_nop: @instruction bound → smallstep :=
  | step_NoOp:
    ∀ s, step_nop OpNoOp s s
  .

End NoOpDefinition.
```

Library EraVM.sem.SpAdd

```
Require SemanticCommon.

Import Arith Addressing CallStack Core isa.CoreSet TransientMemory Resolution
```

```
State SemanticCommon PrimitiveValue spec.

Section SpAddDefinition.

  Open Scope ZMod_scope.
```

SpAdd

Abstract Syntax

`OpSpAdd (in1: in_reg) (ofs: imm_in)`

Syntax

```
nop r0, r0, stack+=[reg+ofs]
```

Summary

Add `(in1 + ofs)` to SP.

Semantic

- Advances PC
- $SP_{new} := SP + (in_1 + ofs)$, but only if there was no overflow.

Affected parts of VM state

- execution stack : PC is increased; SP may be increased.

Usage

Adjusting SP e.g. reserving space on stack.

Similar instructions

`OpSpSub` subtracts a value from SP.

Encoding

- `NoOp`, `SpAdd`, `SpSub` are encoded as the same instruction.

```
  Inductive step_sp_add : instruction → callstack_smallstep :=
  | step_op_sp_add:
    ∀ (cs0 new_cs: callstack) (old_sp intermediate_sp new_sp ofs:
stack_address) op __,
      sp_get cs0 = old_sp →
      (false, intermediate_sp) = old_sp + (low stack_address_bits op) →
      (false, new_sp) = intermediate_sp + ofs→
      new_cs = sp_update new_sp cs0 →
      step_sp_add (OpSpAdd (mk_pv __ op) ofs) cs0 new_cs
  .

End SpAddDefinition.
```

Library EraVM.sem.SpSub

```
Require SemanticCommon.
Import Arith Addressing CallStack Core isa.CoreSet TransientMemory Resolution
State SemanticCommon PrimitiveValue spec.

Section SpSubDefinition.

  Open Scope ZMod_scope.
```

SpSub

Abstract Syntax

```
OpSpSub (in1: in_reg) (ofs: imm_in)
```

Syntax

```
nop stack-=[reg+ofs]
```

Summary

Subtract `(in1 + ofs)` from SP.

Semantic

- Advances PC
- $SP_{new} := SP - (in_1 + ofs)$, but only if there was no overflow.

Affected parts of VM state

- execution stack : PC is increased; SP may be decreased.

Usage

Adjusting SP e.g. deallocating space on stack.

Similar instructions

`OpSpSub` subtracts value from SP.

Encoding

- `NoOp`, `SpAdd`, `SpSub` are encoded as the same instruction.

```
  Inductive step_sp_sub : instruction → callstack_smallstep :=
  | step_op_sp_sub:
    ∀ (cs0 new_cs: callstack) (old_sp intermediate_sp new_sp ofs:
stack_address) op __,
      sp_get cs0 = old_sp →
      (false, intermediate_sp) = old_sp + (low stack_address_bits op) →
      (false, new_sp) = intermediate_sp - ofs→
      new_cs = sp_update new_sp cs0 →
      step_sp_sub (OpSpSub (mk_pv __ op) ofs) cs0 new_cs
  .

End SpSubDefinition.
```

Library EraVM.sem.Jump

```
From RecordUpdate Require Import RecordSet.
```

```
Require SemanticCommon.

Import Addressing Bool Core Common Predication GPR CallStack TransientMemory
MemoryOps isa.CoreSet State
  PrimitiveValue SemanticCommon RecordSetNotations.

Section JumpDefinition.

Inductive step_jump_aux: @instruction bound → callstack → callstack → Prop :=
```

Jump

Unconditional jump (becomes conditional through predication).

Abstract Syntax

`OpJump (dest: in_any)`

Syntax

- `jump destination`

Note: Argument `destination` uses the full addressing mode `in_any`, therefore can be immediate 16-bit value, register, a register value with an offset, and so on.

Semantic

- Fetch a new address from operand `destination`.
- Assign to current PC the fetched value truncated to `code_address_bits` bits.

```
| step_jump_apply:
  ∀ (dest_val: word) (cs new_cs: callstack) __,

    let dest_addr := low code_address_bits dest_val in
    new_cs = pc_set dest_addr cs →

    step_jump_aux (OpJump (mk_pv __ dest_val)) cs new_cs.
```

Affected parts of VM state

- execution stack: PC is overwritten with a new value.

Usage

- Unconditional jumps
- In EraVM, all instructions are predicated (see `Predication.cond`), therefore in conjunction with a required condition type `jump` implements a conditional jump instruction.
- Currently, the compiler may emit jumps rather than `OpNearCall`/`OpNearRet` and similar instructions when possible. It is cheaper, and most functions do not require to install a non-default `cf_exception_handler_location`, nor passing less than all available ergs.

Similar instructions

- Calls: see `OpNearCall`, `OpFarCall`, `OpDelegateCall`, `OpMimicCall`.

```
Inductive step_jump: @instruction_bound → smallstep :=
| step_Jump: ∀ ins (s1 s2:state),
    step_callstack (step_jump_aux ins) s1 s2 →
    step_jump ins s1 s2.

End JumpDefinition.
```

Library EraVM.sem.Add

```
Require SemanticCommon.

Import Bool Common Flags CoreSet TransientMemory Modifiers State PrimitiveValue
SemanticCommon.
Import ssreflect.tuple ssreflect.eqtype.

Section AddDefinition.
  Open Scope ZMod_scope.

  Generalizable Variables op tag.
  Inductive step_add: instruction → flags_tsmallstep :=
```

Add

Abstract Syntax

```
OpAdd (in1: in_any) (in2: in_reg) (out1: out_any) (flags:mod_set_flags)
```

Syntax

- `add in1, in2, out`
- `add! in1, in2, out`, to set `set flags` modifier.

Summary

Unsigned overflowing addition of two numbers modulo 2^{256}.

Semantic

- result is computed by unsigned addition of two numbers with overflow modulo 2^{256}.

$$result := op_1 + op_2 \mod 2^{256}$$

- flags are computed as follows:
  - `LT_OF` is set if overflow occurs, i.e. $op_1 + op_2 \geq 2^{256}$
  - `EQ` is set if $result = 0$.
  - `GT` is set if both `LT_OF` and `EQ` are cleared.

Reminder: flags are only set if `set_flags` modifier is set.

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

- Arithmetic operations.
- There is no dedicated `mov` instruction, so `add` is used to copy values around. Copying A to B is implemented as `add A, r0, B`.

Similar instructions

Flags are computed exactly as in `sub`, but the meaning of overflow is different for addition and subtraction.

```
| step_Add:
  ∀ mod_sf old_flags new_flags result new_OF,
    `(
        (new_OF, result) = op1 + op2 →
        let new_EQ := result == zero256 in
        let new_GT := negb new_EQ && negb new_OF in

        new_flags = apply_set_flags mod_sf
                      old_flags
                      (bflags new_OF new_EQ new_GT) →
          step_add (OpAdd (mk_pv tag1 op1) (mk_pv tag2 op2) (IntValue result)
mod_sf) old_flags new_flags)
    .
  Generalizable No Variables.

End AddDefinition.
```

Library EraVM.sem.Sub

```
Require SemanticCommon.

Import Bool Common Flags CoreSet TransientMemory Modifiers State
  PrimitiveValue SemanticCommon.
Import ssreflect.tuple ssreflect.eqtype.

Section SubDefinition.
  Open Scope ZMod_scope.

  Generalizable Variables op tag.

  Inductive step_sub: instruction → flags_tsmallstep :=
```

Sub

Abstract Syntax

```
OpSub (in1: in_any) (in2: in_reg) (out1: out_any) (swap:mod_swap)
(flags:mod_set_flags)
```

Syntax

- `sub in1, in2, out`
- `sub.s in1, in2, out`, to set `swap` modifier.
- `sub! in1, in2, out`, to set `set flags` modifier.
- `sub.s! in1, in2, out`, to set both `swap` and `set flags` modifiers.

Summary

Unsigned overflowing subtraction of two numbers modulo 2^{256}.

Semantic

- result is computed by unsigned subtraction of two numbers with overflow modulo 2^{256}.

$$result := \begin{cases} op_1 - op_2 & ,\text{if } op_1 \geq op_2 \\ 2^{256} - (op_2 - op_1) & ,\text{if } op_1 < op_2 \end{cases}$$

- flags are computed as follows:
  - `LT_OF` is set if overflow occurs, i.e. $op_1 < op_2$
  - `EQ` is set if $result = 0$.
  - `GT` is set if both `LT_OF` and `EQ` are cleared.

Reminder: flags are only set if `set_flags` modifier is set.

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- Registers, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

Arithmetic operations.

Similar instructions

Flags are computed exactly as in `add`, but the meaning of overflow is different for addition and subtraction.

```
  | step_Sub:
    ∀ mod_sf old_flags new_flags,
      `(
          let (new_OF, result) := op1 - op2 in
          let new_EQ := result == zero256 in
          let new_GT := negb new_EQ && negb new_OF in

          new_flags = apply_set_flags mod_sf
                        old_flags
                        (bflags new_OF new_EQ new_GT) →

          step_sub (OpSub (mk_pv tag1 op1) (mk_pv tag2 op2) (IntValue result)
mod_sf) old_flags new_flags)
      .
    Generalizable No Variables.
End SubDefinition.
```

Library EraVM.sem.Mul

```
Require sem.SemanticCommon.

Import Bool Core Modifiers Common Flags isa.CoreSet CallStack TransientMemory
MemoryOps State
  ZArith PrimitiveValue SemanticCommon List ListNotations.

Import ssreflect.eqtype ssreflect.tuple.

Section MulDefinition.
  Open Scope ZMod_scope.

  Generalizable Variables tag.

  Inductive step_mul: instruction → flags_tsmallstep :=
```

Mul

Abstract Syntax

```
OpMul (in1: in_any) (in2: in_reg) (out1: out_any) (out2: out_reg)
(flags:mod_set_flags)
```

Syntax

- `mul in1, in2, out1, out2`
- `mul! in1, in2, out1, out2`, to set `set flags` modifier.

Summary

Unsigned multiplication of two numbers modulo 2^{512}; the high and low 256 bits of the result are returned in two separate operands.

Semantic

1. Compute result by unsigned multiplication of `in1` by `in2`.

$$\begin{cases} result_{high} := \frac{op_1 \times op_2}{2^{256}} \\ result_{low} := op_1 \times op_2 \mod 2^{256} \end{cases}$$

2. Flags are computed as follows:
   - `LT_OF` is set if overflow occurs, i.e. $op_1 \times op_2 \geq 2^{256}$
   - `EQ` is set if $result_{low} = 0$.
   - `GT` is set if `LT_OF` and `EQ` are cleared.

   Reminder: flags are only set if `set_flags` modifier is set.

```
| step_Mul:
  ∀ mod_sf old_flags new_flags high low high256 low256 op1 op2,
    `(
        high256 = high 256 (op1×op2) →
        low256 = low 256 (op1×op2) →

        let new_EQ := low256 == zero256 in
        let new_OF := high256 != zero256 in
        let new_GT := negb new_EQ && negb new_OF in

        new_flags = apply_set_flags mod_sf old_flags
```

```
                        (bflags new_OF new_EQ new_GT) →

          step_mul (OpMul (mk_pv tag1 op1) (mk_pv tag2 op2) (IntValue low256)
(IntValue high256) mod_sf) old_flags new_flags
        ).
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out1` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, by `out2` and `out1`, provided `out1` resolves to GPR.
- flags, if `set_flags` modifier is set.

Usage

Arithmetic operations.

Similar instructions

- See `OpDiv`.

```
  Generalizable No Variables.
End MulDefinition.
```

Library EraVM.sem.Div

```
Require sem.SemanticCommon.

Import Addressing Bool ZArith Common Flags GPR isa.CoreSet CallStack Modifiers
State
  ZBits Addressing.Coercions PrimitiveValue SemanticCommon List ListNotations.

Section DivDefinition.
  Open Scope Z_scope.

  Generalizable Variables tag.

  Inductive step_div: instruction → flags_tsmallstep :=
```

Div

Abstract Syntax

```
OpDiv (in1: in_any) (in2: in_reg) (out1: out_any) (out2: out_reg) (swap:mod_swap)
(flags:mod_set_flags)
```

Syntax

- `div in1, in2, out1, out2`
- `div.s in1, in2, out1, out2`, to set `swap` modifier.
- `div! in1, in2, out1, out2`, to set `set flags` modifier.
- `div.s! in1, in2, out1, out2`, to set both `swap` and `set flags` modifiers.

Summary

Unsigned division of `in1` by `in2`. The quotient is returned in `out1`, the remainder is returned in `out2`.

Semantic

- If `in2` $\neq 0$:

$$\begin{cases} out_1 := \frac{op_1}{op_2} \\ out_2 := \text{rem } op_1\ op_2 \end{cases}$$

  Flags are computed as follows:

  - `LT_OF` is cleared;
  - `EQ` is set if the quotient is zero;
  - `GT` is set if the reminder is zero.

- If `in2` $= 0$:

$$\begin{cases} out_1 := 0 \\ out_2 := 0 \end{cases}$$

  Flags are computed as follows:

  - `LT_OF` is set.
  - `EQ` is set if $result_{low} = 0$.
  - `GT` is set if `LT_OF` and `EQ` are cleared.

Reminder: flags are only set if `set_flags` modifier is set.

```
  | step_Div_no_overflow:
    `(∀ mod_sf old_flags new_flags w_quot w_rem quot rem (x y:Z) op1 op2,
          x = toZ op1 →
          y = toZ op2 →
          y ≠ 0 →
          quot = Z.div x y →
          rem = Z.rem x y →
          w_quot = u256_of quot →
          w_rem = u256_of rem →

          let new_EQ := quot =? 0 in
          let new_GT := rem =? 0 in
          new_flags = apply_set_flags mod_sf
                        old_flags
                        (bflags false new_EQ new_GT) →

          step_div (OpDiv (mk_pv tag1 op1) (mk_pv tag2 op2) (IntValue w_quot)
(IntValue w_rem) mod_sf) old_flags new_flags)
  | step_Div_overflow:
    ∀ mod_sf old_flags new_flags (x y:Z) op1 op2,
      `(
          x = toZ op1 →
          y = toZ op2 →
          new_flags = apply_set_flags mod_sf old_flags (bflags true false
false) →

          step_div (OpDiv (mk_pv tag1 op1) (mk_pv tag2 op2) (IntValue zero256)
(IntValue zero256) mod_sf) old_flags new_flags
        )
.
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out1` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, by `out2` and `out1`, provided `out1` resolves to GPR.
- flags, if `set_flags` modifier is set.

Usage

Arithmetic operations.

Similar instructions

- See `OpMul`.

```
End DivDefinition.
```

Library EraVM.sem.And

```
Require SemanticCommon.

Import Common CoreSet Modifiers SemanticCommon PrimitiveValue.

Section AndDefinition.
  Open Scope ZMod_scope.

  Generalizable Variables op tag.

  Inductive step_and: instruction → flags_tsmallstep :=
```

Bitwise AND

Abstract Syntax

```
OpAnd (in1: in_any) (in2: in_reg) (out1: out_any) (flags:mod_set_flags)
```

Syntax

- `and in1, in2, out`
- `and! in1, in2, out`, to set `set flags` modifier.

Summary

Bitwise AND of two 256-bit numbers.

Semantic

- result is computed as a bitwise AND of two operands.

- flags are computed as follows:
  - `EQ` is set if $result = 0$.
  - `OF_LT` and `GT` are cleared

Reminder: flags are only set if `set_flags` modifier is set.

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

- operations with bit masks

Similar instructions

- `and`, `or` and `xor` are encoded as variants of the same `mach_instruction`.

```
  | step_And:
    ∀ mod_sf old_flags new_flags result,
      `(
          result = bitwise_and op1 op2 →
          new_flags = apply_set_flags mod_sf
                        old_flags
                        (bitwise_flags result) →

          step_and (OpAnd (mk_pv tag1 op1) (mk_pv tag2 op2) (IntValue result)
mod_sf) old_flags new_flags)
  .
  Generalizable No Variables.
End AndDefinition.
```

Library EraVM.sem.Or

```
Require SemanticCommon.

Import Common CoreSet Modifiers SemanticCommon PrimitiveValue.

Section OrDefinition.
  Open Scope ZMod_scope.
```

```
Generalizable Variables op tag.

Inductive step_or: instruction → flags_tsmallstep :=
```

Bitwise OR

Abstract Syntax

```
OpOr (in1: in_any) (in2: in_reg) (out1: out_any) (flags:mod_set_flags)
```

Syntax

- `or in1, in2, out`
- `or! in1, in2, out`, to set `set_flags` modifier.

Summary

Bitwise OR of two 256-bit numbers.

Semantic

- result is computed as a bitwise OR of two operands.
- flags are computed as follows:
  - `EQ` is set if $result = 0$.
  - `OF_LT` and `GT` are cleared

Reminder: flags are only set if `set_flags` modifier is set.

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

- operations with bit masks

Similar instructions

- `and`, `or` and `xor` are encoded as variants of the same `mach_instruction`.

```
  | step_Or:
    ∀ mod_sf old_flags new_flags result,
      `(
          result = bitwise_or op1 op2 →
          new_flags = apply_set_flags mod_sf
                        old_flags
                        (bitwise_flags result) →

          step_or (OpOr (mk_pv tag1 op1) (mk_pv tag2 op2) (IntValue result)
mod_sf) old_flags new_flags)
  .

  Generalizable No Variables.
End OrDefinition.
```

Library EraVM.sem.Xor

```
Require SemanticCommon.

Import Common CoreSet Modifiers SemanticCommon PrimitiveValue.

Section XorDefinition.
  Open Scope ZMod_scope.

  Generalizable Variables op tag.

  Inductive step_xor: instruction → flags_tsmallstep :=
```

Bitwise XOR

Abstract Syntax

```
OpXor (in1: in_any) (in2: in_reg) (out1: out_any) (flags:mod_set_flags)
```

Syntax

- `or in1, in2, out`
- `or! in1, in2, out`, to set `set flags` modifier.

Summary

Bitwise XOR of two 256-bit numbers.

Semantic

- $result$ is computed as a bitwise XOR of two operands.
- flags are computed as follows:
  - `EQ` is set if $result=0$.
  - `OF_LT` and `GT` are cleared

Reminder: flags are only set if `set_flags` modifier is set.

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

- operations with bit masks

Similar instructions

- `and`, `or` and `xor` are encoded as variants of the same instruction.

```
| step_Xor:
  ∀ mod_sf old_flags new_flags result,
    `(
        result = bitwise_xor op1 op2 →
        new_flags = apply_set_flags mod_sf
                      old_flags
                      (bitwise_flags result) →
```

```
          step_xor (OpXor (mk_pv tag1 op1) (mk_pv tag2 op2) (IntValue result)
mod_sf) old_flags new_flags)
  .

End XorDefinition.
```

Library EraVM.sem.Shl

```
Require sem.SemanticCommon.
Import Arith Core isa.CoreSet PrimitiveValue SemanticCommon spec.

Section ShlDefinition.
  Open Scope ZMod_scope.
  Generalizable Variables tag.
  Import operations operations.BitsNotations.
  Inductive step_shl: instruction → flags_tsmallstep :=
```

Shl

Abstract Syntax

```
OpShl (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
(flags:mod_set_flags)
```

Syntax

- `shl in1, in2, out`
- `shl.s in1, in2, out`, to set `swap` modifier.
- `shl! in1, in2, out`, to set `set flags` modifier.
- `shl.s! in1, in2, out`, to set both `swap` and `set flags` modifiers.

Summary

Bitwise left shift of `in1` by the number of binary digits specified by the lowest byte of `in2`. New binary digits (least significant bits in `out`) are zeros.

Semantic

Follows the scheme described in `binop_state_bitwise_effect_spec`.

- result is computed as `in1 << (in2 mod 256)`

- flags are computed as follows:
  - `EQ` is set if $result=0$.
  - other flags are reset

Reminder: flags are only set if `set_flags` modifier is set.

```
  | step_Shl:
    ∀ mod_sf result op shift w_shift old_flags new_flags,
      `(shift = toNat (low 8 w_shift) →
        result = shlBn op shift →
        step_shl (OpShl (mk_pv tag1 op) (mk_pv tag2 w_shift) (IntValue result)
mod_sf) old_flags new_flags)
  .
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

- Operations with bit masks.
- Fast arithmetic.

Similar instructions

- `shl`, `shr`, `rol` and `ror` are encoded as variants of the same instruction.

```
End ShlDefinition.
```

Library EraVM.sem.Shr

```
Require sem.SemanticCommon.
Import Arith Core isa.CoreSet PrimitiveValue SemanticCommon spec.

Section ShrDefinition.
  Open Scope ZMod_scope.
  Generalizable Variables tag.
  Import operations operations.BitsNotations.

  Inductive step_shr: instruction → flags_tsmallstep :=
```

Shr

Abstract Syntax

```
OpShr (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
(flags:mod_set_flags)
```

Syntax

- `shr in1, in2, out`
- `shr.s in1, in2, out`, to set `swap` modifier.
- `shr! in1, in2, out`, to set `set flags` modifier.
- `shr.s! in1, in2, out`, to set both `swap` and `set flags` modifiers.

Summary

Bitwise left shift of `in1` by the number of binary digits specified by the lowest byte of `in2`. New binary digits (most significant bits in `out`) are zeros.

Semantic

Follows the scheme described in `binop_state_bitwise_effect_spec`.

- result is computed as `in1 >> (in2 mod 256)`
- flags are computed as follows:
  - `EQ` is set if $result = 0$.
  - other flags are reset

Reminder: flags are only set if `set_flags` modifier is set.

```
  | step_Shr:
    ∀ mod_sf result op shift w_shift old_flags new_flags,
      `(shift = toNat (low 8 w_shift) →
        result = shrBn op shift →
        step_shr (OpShr (mk_pv tag1 op) (mk_pv tag2 w_shift) (IntValue result)
mod_sf) old_flags new_flags)
  .
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

- Operations with bit masks.
- Fast arithmetic.

Similar instructions

- `shl`, `shr`, `rol` and `ror` are encoded as variants of the same instruction.

```
End ShrDefinition.
```

Library EraVM.sem.Rol

```
Require sem.SemanticCommon.
Import Arith Core isa.CoreSet PrimitiveValue SemanticCommon spec.

Section RolDefinition.
  Open Scope ZMod_scope.
  Generalizable Variables tag.
  Inductive step_rol: instruction → flags_tsmallstep :=
```

Rol

Abstract Syntax

```
OpRol (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
(flags:mod_set_flags)
```

Syntax

- `rol in1, in2, out`
- `rol.s in1, in2, out`, to set `swap` modifier.
- `rol! in1, in2, out`, to set `set flags` modifier.
- `rol.s! in1, in2, out`, to set both `swap` and `set flags` modifiers.

Summary

Bitwise circular left shift of `in1` by the number of binary digits specified by the lowest byte of `in2`. New binary digits (least significant bits in `out`) are taken from the most significant bits of `in1`.

Semantic

- result is computed as `in1 <<< (in2 mod 256)`
- flags are computed as follows:
  - `EQ` is set if $result=0$.
  - other flags are reset

Reminder: flags are only set if `set_flags` modifier is set.

```
  | step_Rol:
    ∀ mod_sf result op shift w_shift old_flags new_flags,
      `(w_shift = widen word_bits (low 8 shift) →
      result = rolBn op (toNat w_shift) →
      step_rol (OpRol (mk_pv tag1 op) (mk_pv tag2 shift) (IntValue result)
mod_sf) old_flags new_flags)
  .
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

- Operations with bit masks.
- Fast arithmetic.

Similar instructions

- `shl`, `shr`, `rol` and `ror` are encoded as variants of the same instruction.

```
End RolDefinition.
```

Library EraVM.sem.Ror

```
Require sem.SemanticCommon.
Import Arith Core isa.CoreSet PrimitiveValue SemanticCommon spec.

Section RorDefinition.
  Open Scope ZMod_scope.
  Generalizable Variables tag.
  Inductive step_ror: instruction → flags_tsmallstep :=
```

Ror

Abstract Syntax

```
OpRor (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
(flags:mod_set_flags)
```

Syntax

- `ror in1, in2, out`
- `ror.s in1, in2, out`, to set `swap` modifier.
- `ror! in1, in2, out`, to set `set flags` modifier.
- `ror.s! in1, in2, out`, to set both `swap` and `set flags` modifiers.

Summary

Bitwise circular left shift of `in1` by the number of binary digits specified by the lowest byte of `in2`. New binary digits (most significant bits in `out`) are taken from the least significant bits of `in1`.

Semantic

- result is computed as `in1 >>> (in2 mod 256)`
- flags are computed as follows:
  - `EQ` is set if $result=0$.
  - other flags are reset

Reminder: flags are only set if `set_flags` modifier is set.

```
  | step_Ror:
    ∀ mod_sf result op shift w_shift old_flags new_flags,
      `(w_shift = widen word_bits (low 8 shift) →
      result = rorBn op (toNat w_shift) →
      step_ror (OpRor (mk_pv tag1 op) (mk_pv tag2 shift) (IntValue result)
mod_sf) old_flags new_flags)
  .
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- flags, if `set_flags` modifier is set.

Usage

- Operations with bit masks.
- Fast arithmetic.

Similar instructions

- `shl`, `shr`, `rol` and `ror` are encoded as variants of the same instruction.

```
End RorDefinition.
```

Library EraVM.sem.NearCall

```
Require SemanticCommon.

Import NearCallABI Addressing Common Core Flags CallStack GPR Ergs isa.CoreSet
TransientMemory PrimitiveValue State SemanticCommon.
```

```
Import ssreflect ssrfun ssrbool eqtype ssreflect.tuple.

Section NearCallDefinition.
  Open Scope ZMod_scope.
```

NearCall

Abstract Syntax

```
OpNearCall (in1: in_reg) (dest: imm_in) (handler: imm_in)
```

Syntax

- `call abi_reg, callee_address, exception_handler` as a fully expanded form.
- `call abi_reg, callee_address`
  - The assembler expands this variation to `call abi_reg, callee_address, DEFAULT_UNWIND_DEST`. Here:
    `DEFAULT_UNWIND_DEST` is a reserved system label; the linker will resolve it to the default exception handler.
- `call callee_address` is a simplified form.
  - Assembler expands this variation to `call r0, callee_address, DEFAULT_UNWIND_DEST`, where:
    `DEFAULT_UNWIND_DEST` is a reserved system label; linker will resolve it to the default exception handler.
    `R0` is a reserved read-only register that holds 0. This variation passes all ergs to the callee.

Summary

Reserves a portion of the currently available ergs for a new function instance and calls the code inside the current contract space.

Semantic

Reminder: the *callee* is the function that we call; the *caller* is the currently executing function where a call takes place. In other words, the caller calls the callee.

Step-by-step explanation:

1. Read the value of `abi_reg` and decode the following structure from `NearCallABI` from it. The `ergs_passed` field indicates the amount of ergs we intend to pass, but the actual amount of ergs passed gets decided at runtime (see step 2).

   ```
   Record params := { ergs_passed: u32; }.
   ```

   The actual amount of passed ergs is determined by `split_ergs_callee_caller` based

on:

- The ergs allocated for the caller frame.
- The value of `ergs_passed`.

```
  Definition split_ergs_callee_caller (ergs_passed caller_ergs:ergs) : ergs ×
ergs :=
    if ergs_passed == zero32 then (caller_ergs, zero32) else
      match caller_ergs - ergs_passed with
      | (false, remaining) ⇒ (ergs_passed, remaining)
      | (true, remaining) ⇒ (caller_ergs, zero32)
      end.
```

2. Explanation for `split_ergs_caller_callee`:

   - if `ergs_passed` = 0, pass all available ergs to the callee and set the `caller_ergs` to zero. Upon the callee's normal return, its unspent ergs are returned back to the caller.

   - otherwise, if `caller_ergs` $\geq$ `ergs_passed`, pass exactly `ergs_passed`. The `caller_ergs` is set to the unspent amount `ergs_passed - caller_ergs`.

   - otherwise, if the call is not affordable (`ergs_passed` $>$ `caller_ergs`), pass all available ergs to the callee.

   Function `split_ergs_callee_caller` returns a pair of erg values, where:

   - the first component is the amount of ergs actually passed to the callee;
   - the second component is the amount of ergs left to the caller.

   Note: after a normal return (not `panic`), the remaining ergs are returned to the caller.

3. Decrease the number of ergs in the caller frame.

4. Set up the new frame:

   - new PC is assigned the instruction's `callee_address` argument.
   - new exception handler is assigned the instruction's `handler_address` argument.
   - new SP is copied from the old frame as is.
   - the allocated ergs are determined by `split_ergs_caller_callee` in (2).

5. Clear flags.

```
  Inductive step_nearcall: @instruction bound → smallstep :=
  | step_NearCall_pass_some_ergs:
    ∀ (expt_handler call_addr: code_address)
      (passed_ergs callee_ergs caller_ergs: ergs)
      (new_caller:callstack) (new_frame:callstack_common) __flags
(cs:callstack) high224 regs ctx pages gs,
```

```
      (callee_ergs, caller_ergs) = split_ergs_callee_caller passed_ergs
(ergs_remaining cs) →

      new_caller = ergs_set caller_ergs cs →
      new_frame = mk_cf expt_handler (sp_get cs) call_addr callee_ergs
(gs_revertable gs) →

      step_nearcall
        (OpNearCall (Some (IntValue (high224, mk_params passed_ergs)))
call_addr expt_handler)
        {|
          gs_transient := {|
                           gs_flags := __flags;
                           gs_callstack := cs;
                           gs_regs := regs;
                           gs_context_u128 := ctx;
                           gs_pages := pages;
                           gs_status := NoPanic;
                         |};
          gs_global := gs;
        |}
        {|
          gs_transient := {|
                           gs_flags := flags_clear;
                           gs_callstack := InternalCall new_frame new_caller;
                           gs_regs := regs;
                           gs_context_u128 := ctx;
                           gs_pages := pages;
                           gs_status := NoPanic;
                         |};
          gs_global := gs;
        |}.
```

Affected parts of VM state

- Execution stack: a new frame is pushed on top of the execution stack, and the caller frame is changed.
  - Caller frame:
    PC of the caller frame is advanced by one, as in any instruction.
    Ergs are split between caller and callee frames. See `split_ergs_callee_caller`.
  - New (callee) frame:
    PC is set to `callee_address`
    SP is copied to the new frame as is.
    ergs are set to the actual amount passed. See `split_ergs_callee_caller`.
    exception handler
- Flags are always cleared.

Usage

- Set `ergs_passed=0` to pass all available ergs to callee.

- If the first argument is omitted, all available ergs will be passed to callee.

  Explanation: if the first argument is omitted, the assembler implicitly puts `r0` in its place. The reserved register `r0` always holds zero, therefore `ergs_passed` will be decoded into zero as well.
- No particular calling convention is enforced for near calls, so it can be decided by compiler.
- Can be used for internal system code, like bootloader. For example, wrap a pair of AA call + fee payment in any order in such `near_call`, and then rollback the entire frame atomically.

Similar instructions

- See `OpFarCall`, `OpMimicCall`, `OpDelegateCall`. They are used to call code of other contracts.

```
End NearCallDefinition.
```

Library EraVM.sem.NearRet

```
Require SemanticCommon.

Import Common Flags CallStack isa.CoreSet State SemanticCommon.

Section NearRetDefinition.
  Generalizable Variables __ regs pages ctx.
  Inductive step_nearret: @instruction bound → tsmallstep :=
```

NearRet (normal return, not panic/ revert)

Abstract Syntax

`OpNearRet`

Syntax

`ret.ok` aliased as `ret`

A normal return from a **near** call. Will pop up current callframe, give back unspent ergs and continue execution from the saved return address (from where the call had taken place).

Semantic

1. Pass all ergs from the current frame to the parent frame.
2. Drop current frame.
3. Clear flags.

```
  | step_NearRet:
    ∀ cf caller_stack new_caller pages,
      `(
          ergs_return_caller_and_drop (InternalCall cf caller_stack) new_caller
→

          step_nearret OpNearRet {|
                         gs_flags := __;
                         gs_callstack := InternalCall cf caller_stack;


                         gs_regs := regs;
                         gs_pages := pages;
                         gs_context_u128 := ctx;
                         gs_status := NoPanic;
                       |}
                       {|
                         gs_flags := flags_clear;
                         gs_callstack := new_caller;


                         gs_regs := regs;
                         gs_pages := pages;
                         gs_context_u128 := ctx;
                         gs_status := NoPanic;
                       |}
        )
.
```

Affected parts of VM state

- Flags are cleared.
- Execution stack:
  - Current frame is dropped.
  - Caller frame:
    Unspent ergs are given back to caller (but memory growth is paid first).

Usage

Normal return from functions.

```
  Generalizable No Variables.
End NearRetDefinition.
```

Library EraVM.sem.NearRevert

```
Require SemanticCommon.

Import Common Flags CallStack GPR TransientMemory isa.CoreSet State
SemanticCommon.

Section NearRevertDefinition.
  Generalizable Variables regs __.

  Inductive step_nearrevert: @instruction bound → smallstep :=
```

NearRevert (return with recoverable error)

Abstract Syntax

`OpNearRevert`

Syntax

`ret.revert` aliased as `revert`

An erroneous return from a **near** call, executes an exception handler. Will revert all changes in `global_state` produced in the current frame, pop up current frame, give back unspent ergs, and proceed to execute exception handler.

The assembler expands `revert` to `revert r1`, but `r1` is ignored by returns from near calls.

Semantic

1. Perform a `rollback`.
2. Retrieve an exception handler E from the current frame.
3. Pass all ergs from the topmost frame to the parent frame.
4. Drop topmost frame.
5. Clear flags
6. Proceed with executing E.

```
  | step_NearRevert:
    ∀ cf caller_stack new_caller new_cs _eh _sp _pc _ergs saved ctx gs new_gs
pages,
      `(
          let cs := InternalCall (mk_cf _eh _sp _pc _ergs saved) caller_stack
in
          let handler := active_exception_handler cs in

          ergs_return_caller_and_drop cs new_caller →
          rollback saved gs new_gs →
          new_cs = pc_set handler new_caller →
          step_nearrevert OpNearRevert
                          {|
                            gs_transient := {|
                                             gs_flags := __;
                                             gs_callstack := InternalCall cf
caller_stack;

                                             gs_regs := regs;
                                             gs_pages := pages;
                                             gs_context_u128 := ctx;
                                             gs_status := NoPanic;
                                           |};
                            gs_global := gs;
                          |}
                          {|
                            gs_transient := {|
                                             gs_flags := flags_clear;
                                             gs_callstack := new_cs;
                                             gs_regs := regs;
                                             gs_pages := pages;
                                             gs_context_u128 := ctx;
                                             gs_status := NoPanic;
                                           |};
                            gs_global := new_gs;
                          |}
      )
.
```

Affected parts of VM state

- Flags are cleared.
- Execution stack:
  - Current frame is dropped.
  - Caller frame:
    Unspent ergs are given back to caller (but memory growth is paid first).
    Program counter is overwritten with the exception handler address of the dead frame.

Usage

Return from a recoverable error, fail-safe.

```
  Generalizable No Variables.
End NearRevertDefinition.
```

Library EraVM.sem.NearRetTo

```
Require SemanticCommon.

Import Common Flags CallStack isa.CoreSet State SemanticCommon.

Section NearRetToDefinition.
  Generalizable Variables __ regs pages ctx.
  Inductive step_nearretto: @instruction bound → tsmallstep :=
```

NearRetTo (normal return to label, not panic/revert)

Abstract Syntax

- `OpNearRetTo (label: code_address)`

Syntax

- `ret label`

  A normal return from a **near** call. Will pop up current callframe, give back unspent ergs and continue execution from an explicitly provided label.

Semantic

1. Pass all ergs from the current frame to the parent frame.
2. Drop current frame.
3. Clear flags
4. Set PC to the label value.

```
  | step_NearRetTo:
    ∀ cf caller_stack new_caller label,
      `(
          ergs_return_caller_and_drop (InternalCall cf caller_stack) new_caller
→

          step_nearretto (OpNearRetTo label) {|
                          gs_flags := __;
                          gs_callstack := InternalCall cf caller_stack;
```

```
                gs_regs := regs;
                gs_pages := pages;
                gs_context_u128 := ctx;
                gs_status := NoPanic;
              |}
              {|
                gs_flags := flags_clear;
                gs_callstack := pc_set label new_caller;


                gs_regs := regs;
                gs_pages := pages;
                gs_context_u128 := ctx;
                gs_status := NoPanic;
              |}
      )
.
```

Affected parts of VM state

- Flags are cleared.
- Execution stack:
  - Current frame is dropped.
  - Caller frame:
    Unspent ergs are given back to caller (but memory growth is paid first).
    program counter is assigned the label.

Usage

A combination of return and jump.

```
  Generalizable No Variables.
End NearRetToDefinition.
```

Library EraVM.sem.NearRevertTo

```
Require SemanticCommon.

Import Common Flags CallStack GPR TransientMemory isa.CoreSet State
SemanticCommon.

Section NearRevert.
  Generalizable Variables regs pages __.
  Inductive step_nearrevertto: @instruction bound → smallstep :=
```

NearRevertTo (return with recoverable error)

Abstract Syntax

`OpNearRevertTo`

Syntax

`ret.revert label` aliased as `revert label`

An erroneous return from a **near** call to a specified label. Will revert all changes in `global_state` produced in the current frame, pop up current frame, give back unspent ergs, and proceed to execute exception handler.

The assembler expands `revert label` to `revert r1, label`, but `r1` is ignored by returns from near calls.

Semantic

1. Perform a `rollback`.
2. Pass all ergs from the topmost frame to the parent frame.
3. Drop topmost frame.
4. Clear flags
5. Proceed with executing `label`, i.e. replace program counter with the label's value.

```
  | step_NearRevert:
    ∀ cf caller_stack new_caller _eh _sp _pc _ergs saved ctx label gs new_gs
pages,
      `(
          let cs := InternalCall (mk_cf _eh _sp _pc _ergs saved) caller_stack
in
          let handler := active_exception_handler cs in

          ergs_return_caller_and_drop cs new_caller →
          rollback saved gs new_gs →
          step_nearrevertto (OpNearRevertTo label)
                            {|
                              gs_transient := {|
                                                gs_flags := __ ;
                                                gs_callstack := InternalCall cf
caller_stack;

                                                gs_regs := regs;
                                                gs_pages := pages;
                                                gs_context_u128 := ctx;
```

```
                                    gs_status := NoPanic;
                                  |};
                    gs_global := gs;
                  |}
                  {|
                    gs_transient := {|
                                    gs_flags := flags_clear;
                                    gs_callstack := pc_set label
new_caller;

                                    gs_regs := regs;
                                    gs_pages := pages;
                                    gs_context_u128 := ctx;
                                    gs_status := NoPanic;
                                  |};
                    gs_global := new_gs;
                  |}
        )
  .
```

Affected parts of VM state

- Flags are cleared.
- Execution stack:
  - Current frame is dropped.
  - Caller frame:
    Unspent ergs are given back to caller (but memory growth is paid first).
    Program counter is overwritten with the exception handler address of the dead frame.

Usage

Return from a recoverable error, fail-safe.

```
End NearRevert.
```

Library EraVM.sem.StepPanic

```
From RecordUpdate Require Import RecordSet.

Require Addressing CallStack Common GPR Flags isa.CoreSet State Steps VMPanic.
Import Addressing CallStack Common GPR Flags isa.CoreSet PrimitiveValue State
Steps VMPanic.
Import RecordSetNotations.

Section StepPanic.
```

Handling panics

EraVM handles the panic situation as follows:

- return from the current function signaling an error;
- execute exception handler;
- burn all ergs in current frame;
- set OF flag;
- restore depot and event queues to the state before external call (see `gs_revertable`).
- when returning from a far call, return no data.

Case 1: `panic` from near call

1. Perform a `rollback`.
2. Drop current frame with its ergs.
3. Set PC to the exception handler of the dropped frame.
4. Clear flags, and set OF.
5. Clears the context register.

```
Inductive step_panic reason: smallstep :=
| step_PanicLocal_nolabel:
  ∀ flags pages cf caller_stack regs gs gs' __,
    let handler := active_exception_handler (InternalCall cf caller_stack) in
    rollback (cf_saved_checkpoint cf) gs gs' →
    step_panic reason
                {|
                  gs_transient:= {|
                                  gs_flags := flags;
                                  gs_callstack := InternalCall cf
caller_stack;
                                  gs_regs := regs;
                                  gs_context_u128 := __;
                                  gs_status := Panic reason;


                                  gs_pages := pages;
                                |};

                  gs_global := gs;
                |}
                {|
                  gs_transient:= {|
                                  gs_flags := set_overflow flags_clear;
                                  gs_regs := regs;
                                  gs_callstack := pc_set
cf.(cf_exception_handler_location) caller_stack;
                                  gs_context_u128 := zero128;
                                  gs_status := NoPanic;

                                  gs_pages := pages;
                                |};
```

```
                gs_global := gs'
              |}
```

Case 2: `panic` from external call

Performs all the same actions as a panic from internal call: 1. Perform a `rollback`. 2. Drop current frame and its ergs 3. Clear flags and set OF. 4. Clear context register. 5. Set PC to the exception handler address of a dropped frame..

In addition to that: 6. Put an encoded zero-pointer into `R1` and tag `R1` as a pointer. All other registers are zeroed. Registers `R2`, `R3` and `R4` are reserved and may gain a special meaning in newer versions of EraVM.

```
| step_PanicExt:
  ∀ flags pages cf caller_stack __ regs gs gs' new_regs,
    let cs0 := ExternalCall cf (Some caller_stack) in
    rollback (cf_saved_checkpoint cf) gs gs' →
    new_regs = regs_state_zero <| r1 := PtrValue zero256 |> →
    step_panic reason
               {|
                 gs_transient := {|
                                   gs_flags := flags;
                                   gs_callstack := cs0;
                                   gs_regs := regs;
                                   gs_context_u128 := __;
                                   gs_status := Panic reason;


                                   gs_pages := pages;
                                 |};

                 gs_global := gs;
               |}
               {|
                 gs_transient := {|
                                   gs_flags := set_overflow flags_clear;
                                   gs_regs := new_regs;
                                   gs_callstack := pc_set
(active_exception_handler cs0) caller_stack;
                                   gs_context_u128 := zero128;
                                   gs_status := NoPanic;

                                   gs_pages := pages;
                                 |};

                 gs_global := gs'
               |}
  .
End StepPanic.
```

Library EraVM.sem.Panic

```
Require SemanticCommon VMPanic StepPanic.

Import isa.CoreSet SemanticCommon VMPanic StepPanic.

Inductive step_oppanic: @instruction bound → smallstep :=
```

Panic (irrecoverable error, not normal return/not return from recoverable error)

Return from a function/contract signaling an error; execute exception handler, burn all ergs in current frame, set OF flag, return nothing, perform `rollback`. See `Panics`.

Abstract Syntax

`OpPanic`

Syntax

`ret.panic` aliased as `panic`

An abnormal return from a **near** call. Will drop current callframe, burn all ergs and pass control to the current exception handler, setting OF flag.

Additionally, restore storage and event queues to the state before external call.

Semantic

Trigger panic with a reason `TriggeredExplicitly`. See `Panics`.

```
| step_trigger_panic:
  ∀ s s',
    step_panic TriggeredExplicitly s s' →
    step_oppanic OpPanic s s'.
```

Affected parts of VM state

- Flags are cleared, then OF is set.
- Context register is zeroed (only returns from far calls).
- Registers are cleared (only returns from far calls).
- Execution stack:
  - Current frame is dropped.
  - Caller frame:
    if a label is explicitly provided, and current frame is internal (near call), then caller's PC is overwritten with the label. Returns from external calls ignore label, even if it is explicitly provided.
    Unspent ergs are given back to caller (but memory growth is paid first).
- Storage changes are reverted.

Usage

- Abnormal returns from near/far calls when an irrecoverable error has happened. Use `revert` for recoverable errors.

Similar instructions

- `ret` returns to the caller instead of executing an exception handler, and does not burn ergs.
- `revert` acts similar to `panic` but does not burn ergs, returns data to the caller, and does not set an overflow flag.

Library EraVM.sem.NearPanicTo

```
From RecordUpdate Require Import RecordSet.
Require SemanticCommon StepPanic.

Import Common Flags CallStack GPR TransientMemory isa.CoreSet State
SemanticCommon VMPanic RecordSetNotations StepPanic isa.CoreSet.

Section NearPanicToDefinition.
Inductive step_panicto: instruction → smallstep :=
```

NearPanic (abnormal return, not return/revert)

Abstract Syntax

`OpNearPanicTo`

Syntax

`ret.panic label` aliased as `panic label`

An erroneous return from a **near** call to a specified label. Will revert all changes in `global_state` produced in the current frame, drop the current frame, give back unspent ergs, and proceed to execute exception handler.

The assembler expands `panic label` to `panic r1, label`, but `r1` is ignored by returns from near calls.

Semantic

1. Perform a `rollback`.
2. Drop topmost frame. Its ergs are burned (lost).
3. Set flag `OF_LT`, clear all other flags.
4. Proceed with executing `label`, i.e. replace program counter with the label's value.

```
| step_NearPanic:
    ∀ label s1 s2 s3,
      step_panic TriggeredExplicitly s1 s2 →
      s3 = s2 <| gs_transient ::= fun ts ⇒ ts <| gs_callstack ::= pc_set label
|> |> →
      step_panicto (@OpNearPanicTo bound label) s1 s3
.
```

Affected parts of VM state

- Flags are cleared.
- Execution stack:
  - Current frame is dropped.
  - Caller frame:
    Unspent ergs are given back to caller (but memory growth is paid first).
    Program counter is overwritten with the exception handler address of the dead frame.

Usage

Return from an irrecoverable error, fail-fast.

```
End NearPanicToDefinition.
```

Library EraVM.sem.Farcall

```
From RecordUpdate Require Import RecordSet.

Require
ABI
CallStack
Common
Decommitter
Ergs
MemoryManagement
MemoryOps
Pointer
SemanticCommon
State
TransientMemory
isa.CoreSet
.

Import
  BinIntDef.Z
    Bool
    List
    ListNotations
    ZArith.

Import
    ABI
      FarCallABI
      FatPointerABI
    CallStack
    Coder
    Common
    Core
    Decommitter
    Ergs
    Flags
    GPR
    KernelMode
    memory.Depot
    MemoryBase
    MemoryContext
    MemoryManagement
    MemoryOps
```

```
    PrimitiveValue
    RecordSetNotations
    SemanticCommon
    TransientMemory
    State
    VersionedHash
    isa.CoreSet
  Addressing
.

Import Addressing.Coercions.

Local Coercion Z.b2z: bool >-> Z.
Section FarCalls.
```

Far calls

Far calls are calls to the code outside the current contract space. This section describes three instructions to perform far calls:

- `OpFarCall`
- `OpDelegateCall`
- `OpMimicCall` (available only in kernel mode)

These instructions differ in the way they construct new frame.

The far call instructions have rich semantics; their full effect on the VM state is described through the following main predicates:

- `Semantics.step`
- `step`
- `fetch_operands`
- `farcall`

If you know about fetching operands for instructions and the instruction fetching described in `Semantics.step`, start investigating farcalls from the `farcall` predicate.

```
Section Parameters.
  Open Scope Z_scope.
  Open Scope ZMod_scope.
```

Global farcall parameters

1. Initial preallocated stack space.

   A far call creates a new context with a new stack page (and other pages, see `page_alloc`). The initial SP value after a far call is set to `INITIAL_SP_ON_FAR_CALL`.

Therefore, addresses in range from 0 inclusive to `INITIAL_SP_ON_FAR_CALL` exclusive can be used as a scratch space.

```
Definition INITIAL_SP_ON_FAR_CALL : stack_address := fromZ 1024.
```

2. Initial heap and auxheap pages bound.

   The heap and auxheap pages start with `NEW_FRAME_MEMORY_STIPEND` bound. Growing them beyond this bound costs ergs.

```
Definition NEW_FRAME_MEMORY_STIPEND : mem_address := fromZ 1024.

End Parameters.
```

3. Maximal fraction of ergs allowed to pass.

   It is not allowed to pass more than 63/64th of your remaining ergs to a far call.

```
Definition max_passable (remaining:ergs) : ergs := fromZ (toZ remaining × 63 /
64 ) %Z.
Inductive pass_allowed_ergs : (ergs × callstack )-> ergs × callstack → Prop :=
| pae_apply: ∀ cs1 cs2 pass_ergs_query,
    let pass_ergs_actual := min (max_passable (ergs_remaining cs1))
pass_ergs_query in
    pay pass_ergs_actual cs1 cs2 →
    pass_ergs_query ≠ zero32 →
    pass_allowed_ergs (pass_ergs_query,cs1) (pass_ergs_actual, cs2)
| pae_zero: ∀ cs1 cs2,
    let pass_ergs_actual := max_passable (ergs_remaining cs1) in
    pay pass_ergs_actual cs1 cs2 →
    pass_allowed_ergs (zero32, cs1) (pass_ergs_actual, cs2).
```

Helpers

Far call creates a new execution context with new pages for:

- code
- const (in the current implementation, const and code pages are the same page).
- stack
- heap
- auxheap

The initial bounds for the new heap and auxheap pages are set to NEW_FRAME_MEMORY_STIPEND.

```
Inductive alloc_pages_extframe: pages × mem_ctx → code_page → const_page →
pages × mem_ctx → Prop :=
| ape_alloc: ∀ code const (mm:pages) ctx code_id const_id stack_id heap_id
heap_aux_id,
    code_id = List.length mm →
    (const_id = code_id + 1)%nat →
    (stack_id = code_id + 2)%nat →
    (heap_id = code_id + 3)%nat →
    (heap_aux_id = code_id + 4)%nat →
    alloc_pages_extframe (mm,ctx) code const
      ( (heap_aux_id, (mk_page (DataPage (empty data_page_params)))::
         (heap_id, (mk_page (DataPage (empty _))))::
         (stack_id, (mk_page (StackPage (empty stack_page_params))))::
         (const_id,(mk_page (ConstPage const)))::
         (code_id, (mk_page (CodePage code)))::mm,
        {|
          ctx_code_page_id := code_id;
          ctx_const_page_id := const_id;
          ctx_stack_page_id := stack_id;
          ctx_heap_page_id := heap_id;
          ctx_auxheap_page_id := heap_aux_id;
          ctx_heap_bound := NEW_FRAME_MEMORY_STIPEND;
          ctx_auxheap_bound := NEW_FRAME_MEMORY_STIPEND;
        |} ).
```

```
Inductive alloc_mem_extframe: memory × mem_ctx → code_page → const_page →
memory × mem_ctx → Prop :=
| ame_apply: ∀ p c p' c' code const,
    alloc_pages_extframe (p,c) code const (p',c') →
    alloc_mem_extframe (mk_pages p,c) code const (mk_pages p',c').
```

Fetch code and pay the associated cost. If masking_allowed is true and there is no code associated with a given contract address, then the default AA code will be fetched. See code_fetch.

```
Inductive paid_code_fetch_result :=
| Fetched : callstack → code_page → const_page → paid_code_fetch_result
| CodeFetchInvalidVerisonedHashFormat (_:versioned_hash)
| CodeFetchUnaffordable (cost:ergs)
.

Inductive paid_code_fetch masking_allowed sid: depot → decommitter →
contract_address → callstack → paid_code_fetch_result →Prop :=
| cfp_fetched:
  ∀ depot (codes:decommitter) (dest_addr: contract_address) vhash dest_addr
new_code_page new_const_page code_length cost__decomm cs0 cs1,

    code_fetch _ depot codes.(cm_storage _) sid dest_addr masking_allowed
(vhash, (new_code_page, new_const_page), code_length) →
    decommitment_cost _ codes vhash code_length cost__decomm →
```

```
    pay cost__decomm cs0 cs1 →
    paid_code_fetch masking_allowed sid depot codes dest_addr cs0 (Fetched cs1
new_code_page new_const_page)
| cfp_unaffordable:
  ∀ depot (codes:decommitter) (dest_addr: contract_address) vhash dest_addr
new_code_page new_const_page code_length cost__decomm cs0 ,
    code_fetch _ depot codes.(cm_storage _) sid dest_addr masking_allowed
(vhash, (new_code_page, new_const_page), code_length) →
    decommitment_cost _ codes vhash code_length cost__decomm →
    affordable cs0 cost__decomm = false →
    paid_code_fetch masking_allowed sid depot codes dest_addr cs0
(CodeFetchUnaffordable cost__decomm)

| cfp_invalid_hash:
  ∀ depot (codes:decommitter) (dest_addr: contract_address) vhash dest_addr
new_code_page new_const_page code_length cs0 ,
    code_fetch _ depot codes.(cm_storage _) sid dest_addr masking_allowed
(vhash, (new_code_page, new_const_page), code_length) →
    marker_valid (extra_marker vhash) = false →
    paid_code_fetch masking_allowed sid depot codes dest_addr cs0
(CodeFetchInvalidVerisonedHashFormat vhash)
.
```

System calls

A system call is a far call that satisfies the following conditions:

- The destination is a kernel address.
- The field `is_system` of `FarCall.params` passed through an operand is set to 1.

Far call instructions

Summary

- Far calls are calls to the code outside the current contract space.
- Mimic calls are a kernel-only variation of far calls allowing to mimic a call from any contract by impersonating an arbitrary caller and putting an arbitrary address into the new callframe's `ecf_msg_sender` field.
- Delegate calls are a variation of far calls allowing to call a contract with the current storage space.

  Example: Suppose we have contracts A,B,C. Contract A called contract B normally, then contract B delegated to contract C. Then C's code will be executed in a context of B's storage, as if contract A called contract C. If contract C returns normally, the execution will proceed from the next

instruction of B after delegate call. In case of `revert` or `panic` in C, all the usual rules apply.

Abstract and concrete syntax

- `OpFarCall` `abi_params address handler is_static`
  - `farcall abi_reg, dest_addr`
  - `farcall abi_reg, dest_addr, handler`
  - `farcall.static abi_reg, dest_addr`
  - `farcall.static abi_reg, dest_addr, handler`
  - `farcall.shard abi_reg, dest_addr`
  - `farcall.shard abi_reg, dest_addr, handler`
- `OpDelegateCall` abi_params address handler is_static`
  - `delegatecall abi_reg, dest_addr`
  - `delegatecall abi_reg, dest_addr, handler`
  - `delegatecall.static abi_reg, dest_addr`
  - `delegatecall.static abi_reg, dest_addr, handler`
  - `delegatecall.shard abi_reg, dest_addr`
  - `delegatecall.shard abi_reg, dest_addr, handler`
- `OpMimicCall` `abi_params address handler is_static`
  - `mimic abi_reg, dest_addr`
  - `mimic abi_reg, dest_addr, handler`
  - `mimic.static abi_reg, dest_addr`
  - `mimic.static abi_reg, dest_addr, handler`
  - `mimic.shard abi_reg, dest_addr`
  - `mimic.shard abi_reg, dest_addr, handler`
- **static** modifier marks the new execution stack frame as ‘static’, preventing some instructions from being executed. Calls from a static calls are automatically marked static.
- **shard** modifier allows calling code from other shards. The shard ID will be taken from `abi_reg`.

Semantic

1. Decode the structure `params` from `abi_reg`:

```
Inductive fwd_memory :=
  ForwardFatPointer (p:fat_ptr)
| ForwardNewFatPointer (heap_var: data_page_type) (s:span).


Record params :=
  mk_params {
      fwd_memory: fwd_memory;
      ergs_passed: ergs;
      shard_id: shard_id;
```

```
  constructor_call: bool;
  to_system: bool;
}.
```

3. Decommit code of the callee contract (formalized by `paid_code_fetch`):
    - load the `versioned_hash` of the called code from the storage of a special contract located at `DEPLOYER_SYSTEM_CONTRACT_ADDRESS`.

      ```
      Inductive marker := CODE_AT_REST | YET_CONSTRUCTED | INVALID.

      Record versioned_hash := mk_vhash {
                                 code_length_in_words: u16;
                                 extra_marker: marker;
                                 partial_hash: BITS (28*bits_in_byte)%nat
                               }.
      ```

    - for non-system calls, if there is no code stored for a provided hash value, mask it into `VersionedHash.DEFAULT_AA_VHASH` and execute `VersionedHash.DEFAULT_AA_CODE`.
    - if the code with such hash has not been accessed in the current block, pay for decommitment.
4. Forward data to the new frame (formalized by `paid_forward_and_adjust_bounds`).
    - If `params.(fwd_memory)` is `ForwardExistingFatPointer p`, we are forwarding an existing fat pointer.

      ensure that `abi_reg` is tagged as a pointer.
      check the pointer validity;
      `fat_ptr_narrow` the pointer;

    - If `params.(fwd_memory)` is `ForwardNewFatPointer variant span`, a new `fat_ptr` is created. This pointer refers to the provided `span` of specified heap `variant`.

      Note the decoding of `ForwardNewFatPointer` in `fwd_memory_adapter` and especially `span_of`.
5. Allocate new pages for code, constants, stack, heap and auxheap (formalized by `alloc_pages_extframe`).
6. Reserve ergs for the new external frame (formalized by `pass_allowed_ergs`).
    - Maximum amount of ergs passed to an external call is 63/64 of ergs allocated for the caller.
    - Attempting to pass more ergs will result in only passing the maximum amount allowed.
    - Trying to pass 0 ergs will result in passing maximum amount of ergs allowed.
7. Clear the context register.
8. Clear flags.
9. Modify GPRs depending on the call being system or not (formalized by `regs_effect`):

- Effect of a non-system call:
  All registers are cleared.
  Register `R1` is assigned a fat pointer to forward data to the far call. See `paid_forward`.
- Effect of a system call:
  Register `R1` is assigned a fat pointer to forward data to the far call. See `paid_forward`.
  Register `R2` is assigned a bit-value:
    bit 1 indicates “this is a system call”
    bit 0 indicates “this is a constructor call”
  Registers `r3`, `r4`, …, `r15` are reserved; their pointer tags are cleared, but their values are unchanged.
  Registers `r14` and `r15` are cleared.

```
Definition regs_effect regs (is_system is_ctor:bool) ptr :=
  let far_call_r2 :=
    let is_system_bit := Z.shiftl is_system 1 in
    let is_ctor_bit := Z.shiftl is_ctor 0 in
    let bits := Z.lor is_system_bit is_ctor_bit in
    IntValue (fromZ bits) in
  match encode_fat_ptr_word zero128 ptr with
    | Some enc_ptr ⇒ Some
        (if is_system then
          regs
            <| r1 := PtrValue enc_ptr |>
            <| r2 := far_call_r2 |>
      (* In system calls, preserve values in r3-r13 but clear ptr tags *)
            <| r3 ::= clear_pointer_tag |>
            <| r4 ::= clear_pointer_tag |>
            <| r5 ::= clear_pointer_tag |>
            <| r6 ::= clear_pointer_tag |>
            <| r7 ::= clear_pointer_tag |>
            <| r8 ::= clear_pointer_tag |>
            <| r9 ::= clear_pointer_tag |>
            <| r10 ::= clear_pointer_tag |>
            <| r11 ::= clear_pointer_tag |>
            <| r12 ::= clear_pointer_tag |>
            <| r13 ::= clear_pointer_tag |>
            (* zero the rest *)
            <| r14 := IntValue word0 |>
            <| r15 := IntValue word0 |>
          else
            regs_state_zero <| r1 := PtrValue enc_ptr |> <| r2 := far_call_r2 |>
        )
    | _ ⇒ None
  end.
```

10. Form a new execution stack frame:
    - the call is static if the current call is static, or if `.is_static` modifier is applied to instruction;
    - set exception handler to `handler` address provided in the instruction;

- it is a checkpoint that saves all storage states;
- start PC at 0;
- start SP at `INITIAL_SP_ON_FAR_CALL`;
- `this_address`,`msg_sender` and `context` fields are affected by the `farcall_type` as follows:
  - Normal far call sets:
    - `this_address` <- destination address;
    - `msg_sender` <- caller address;
    - `context` <- value of context register `gs_context_u128`.
  - Delegate call sets:
    - `this_address` <- `this_address` of the current frame;
    - `msg_sender` <- `msg_sender` of the current frame;
    - `context` <- `context_u128` of the current frame.
  - Mimic call sets:
    - `this_address` <- destination address;
    - `msg_sender` <- value of `r15`;
    - `context` <- value of context register `gs_context_u128`.

```
Definition CALL_IMPLICIT_PARAMETER_REG := R15.
Inductive farcall_type : Set := Normal | Mimic | Delegate.
Definition select_this_address type (caller dest: contract_address) :=
  match type with
  | Normal ⇒ dest
  | Mimic ⇒ dest
  | Delegate ⇒ caller
  end.
```

```
Definition select_sender type (callers_caller caller : contract_address) regs
:=
  match type with
  | Normal ⇒ caller
  | Delegate ⇒ callers_caller
  | Mimic ⇒
      let r15_value := (fetch_gpr regs CALL_IMPLICIT_PARAMETER_REG).(value) in
      low contract_address_bits r15_value
  end.

Definition select_associated_contracts type regs (ac:associated_contracts)
(call_dest: contract_address): associated_contracts :=
  match ac with
  | mk_assoc_contracts this_address msg_sender code_address ⇒
      {|
        ecf_this_address := select_this_address type this_address call_dest;
        ecf_msg_sender := select_sender type ac.(ecf_msg_sender) this_address
regs;
        ecf_code_address := call_dest;
      |}
  end.

Definition select_ctx type (reg_ctx frame_ctx: u128) :=
  match type with
  | Normal | Mimic ⇒ reg_ctx
  | Delegate ⇒ frame_ctx
  end.
```

```
Definition new_code_shard_id (is_call_shard:bool)
  (provided current_shard:shard_id) : shard_id :=
  if is_call_shard then provided else current_shard.

Definition select_shards (type: farcall_type) (is_call_shard: bool) (provided:
shard_id) (ss: active_shards) : active_shards :=
  match ss with
  | mk_shards old_this _ code ⇒
      let new_caller := old_this in
      let new_code := new_code_shard_id is_call_shard provided new_caller in
      let new_this := match type with | Delegate ⇒ new_caller | _ ⇒ new_code
end in
      {|
        shard_this := new_this;
        shard_caller := new_caller;
        shard_code := new_code;
      |}
  end.

Section FarCallDefinitions.

  Import Pointer.
  Context (type:farcall_type) (is_static_call is_shard_provided:bool)
(dest:contract_address) (handler: code_address) (gs:global_state).

  Inductive farcall : @primitive_value FarCallABI.params → tsmallstep :=

  | farcall_fwd_existing_fatptr: ∀ flags old_regs old_pages cs0 cs1
new_caller_stack new_stack reg_context_u128 new_regs new_pages new_code_page
new_const_page new_mem_ctx (in_ptr narrowed_ptr: fat_ptr) abi_shard ergs_query
ergs_actual is_syscall_query,

      let caller_extframe := active_extframe cs0 in
      let mem_ctx0 := ecf_mem_ctx caller_extframe in
      let is_system := addr_is_kernel dest && is_syscall_query in
      let allow_masking := negb is_system in
      let callee_shard := if is_shard_provided then abi_shard else
current_shard cs0 in

      paid_code_fetch allow_masking callee_shard (gs_depot gs) (gs_contracts
gs) dest cs0 (Fetched cs1 new_code_page new_const_page) →

      (*!*)validate in_ptr = no_exceptions →
      (*!*)fat_ptr_narrow in_ptr narrowed_ptr →

      alloc_pages_extframe (old_pages, mem_ctx0) new_code_page new_const_page
(new_pages, new_mem_ctx) →
      pass_allowed_ergs (ergs_query,cs1) (ergs_actual, new_caller_stack) →

      new_stack = ExternalCall {|
                            ecf_associated := select_associated_contracts type
old_regs caller_extframe.(ecf_associated) dest;
                            ecf_context_u128_value := select_ctx type
reg_context_u128 caller_extframe.(ecf_context_u128_value);
                            ecf_shards := select_shards type is_shard_provided
abi_shard caller_extframe.(ecf_shards);
```

```
                            ecf_mem_ctx := new_mem_ctx;
                            ecf_is_static := ecf_is_static caller_extframe ||
is_static_call;
                            ecf_common := {|
                                           cf_exception_handler_location :=
handler;
                                           cf_sp := INITIAL_SP_ON_FAR_CALL;
                                           cf_pc := zero16;
                                           cf_ergs_remaining := ergs_actual;
                                           cf_saved_checkpoint :=
gs.(gs_revertable);
                                         |};
                          |} (Some new_caller_stack) →
      Some new_regs = regs_effect old_regs is_system false (NotNullPtr
narrowed_ptr) →
      farcall
        (PtrValue {|
FarCallABI.fwd_memory := ForwardExistingFatPointer (NotNullPtr in_ptr);
          ergs_passed := ergs_query;
FarCallABI.shard_id := abi_shard;
          constructor_call := false;
          to_system := is_syscall_query;
        |})
        {|
          gs_flags := flags;
          gs_regs := old_regs;
          gs_pages := old_pages;
          gs_callstack := cs0;
          gs_context_u128 := reg_context_u128;
          gs_status := NoPanic;
        |}
        {|
          gs_flags := flags_clear;
          gs_regs := new_regs;
          gs_pages := new_pages;
          gs_callstack := new_stack;
          gs_context_u128 := zero128;
          gs_status := NoPanic;
        |}

  | farcall_fwd_new_ptr: ∀ flags old_regs old_pages cs0 cs1 cs2 new_regs
new_caller_stack new_stack reg_context_u128 new_pages new_code_page
new_const_page new_mem_ctx abi_shard ergs_query ergs_actual is_syscall_query
out_ptr in_span page_type,

      let is_system := addr_is_kernel dest && is_syscall_query in
      let allow_masking := negb is_system in
      let callee_shard := if is_shard_provided then abi_shard else
current_shard cs0 in

      paid_code_fetch allow_masking callee_shard
        gs.(gs_revertable).(gs_depot) gs.(gs_contracts) dest cs0 (Fetched cs1
new_code_page new_const_page) →

      (*!*)paid_forward_new_fat_ptr page_type in_span cs0 (out_ptr, cs1) →

      let caller_extframe := active_extframe cs2 in
```

```
      let mem_ctx0 := caller_extframe.(ecf_mem_ctx) in
      alloc_pages_extframe (old_pages, mem_ctx0) new_code_page new_const_page
(new_pages, new_mem_ctx) →
      pass_allowed_ergs (ergs_query,cs2) (ergs_actual, new_caller_stack) →

      new_stack = ExternalCall {|
                            ecf_associated := select_associated_contracts type
old_regs caller_extframe.(ecf_associated) dest;
                            ecf_context_u128_value := select_ctx type
reg_context_u128 caller_extframe.(ecf_context_u128_value);
                            ecf_shards := select_shards type is_shard_provided
abi_shard caller_extframe.(ecf_shards);

                            ecf_mem_ctx := new_mem_ctx;
                            ecf_is_static := ecf_is_static caller_extframe ||
is_static_call;
                            ecf_common := {|
                                          cf_exception_handler_location :=
handler;
                                          cf_sp := INITIAL_SP_ON_FAR_CALL;
                                          cf_pc := zero16;
                                          cf_ergs_remaining := ergs_actual;
                                          cf_saved_checkpoint :=
gs.(gs_revertable);
                                        |};
                          |} (Some new_caller_stack) →
      Some new_regs = regs_effect old_regs is_system false (NotNullPtr out_ptr)
→
      farcall
        (IntValue {|
FarCallABI.fwd_memory := ForwardNewFatPointer page_type in_span;
          ergs_passed := ergs_query;
FarCallABI.shard_id := abi_shard;
          constructor_call := false;
          to_system := is_syscall_query;
(*!*) |})
        {|
          gs_flags := flags;
          gs_regs := old_regs;
          gs_pages := old_pages;
          gs_callstack := cs0;
          gs_context_u128 := reg_context_u128;
          gs_status := NoPanic;
        |}
        {|
          gs_flags := flags_clear;
          gs_regs := new_regs;
          gs_pages := new_pages;
          gs_callstack := new_stack;
          gs_context_u128 := zero128;
          gs_status := NoPanic;
        |}
```

Affected parts of VM state

- flags are cleared
- registers are affected as described by `regs_effect`.
- new pages appear as described by `alloc_pages_extframe`.
- context register is zeroed.
- execution stack is affected in a non-trivial way (see step 10 in description for `farcall`).

Comparison with near calls

- Far calls can not accept more than `max_passable` ergs, while near calls may accept all available ergs.
- Abnormal returns from far calls through `OpPanic` or `OpRevert` roll back all storage changes that occured during the contract execution.

This includes exceptional situations when an error occured and the current instruction is masked as `OpPanic`.

Usage

- Calling other contracts
- Calling precompiles Usually we call a system contract with assigned precompile. It prepares data for a precompile, performs precompile call, and returns the result.

Encoding

- In the encoding, `OpDelegateCall`, `OpFarCall`, and `OpMimicCall` share the same opcode.

Panics

1. Attempting to pass an existing `fat_ptr`, but the passed value is not tagged as a pointer.

```
  | farcall_fwd_existing_fatptr_notag: ∀ (ts1 ts2:transient_state) ___0 ___1
___2 ___3 ___4,

    ts2 = ts1 <| gs_status := Panic FarCallInputIsNotPointerWhenExpected |> →
    farcall
      (IntValue {|
          FarCallABI.fwd_memory := ForwardExistingFatPointer ___0;
          ergs_passed := ___1;
          FarCallABI.shard_id := ___2;
          constructor_call := ___3;
          to_system := ___4;
        |}) ts1 ts2
```

2. The hash for the contract code (stored in the storage of DEPLOYER_SYSTEM_CONTRACT_ADDRESS) is malformed.

```
  | farcall_malformed_decommitment_hash: ∀ cs0 abi_shard is_syscall_query ts1
ts2 ___1 ___2 ___3 ___4 _tag,
      let is_system := addr_is_kernel dest && is_syscall_query in
      let allow_masking := negb is_system in
      let callee_shard := if is_shard_provided then abi_shard else
current_shard cs0 in
      paid_code_fetch allow_masking callee_shard (gs_depot gs) (gs_contracts
gs) dest cs0 (CodeFetchInvalidVerisonedHashFormat ___4) →

      ts2 = ts1 <| gs_status := Panic FarCallInvalidCodeHashFormat |> →
      farcall
        (mk_pv _tag {|
          FarCallABI.fwd_memory := ___1;
          ergs_passed := ___2;
          FarCallABI.shard_id := abi_shard;
          constructor_call := ___3;
          to_system := is_syscall_query;
        |})
       ts1 ts2
```

3. Not enough ergs to pay for code decommitment.

```
  | farcall_decommitment_unaffordable: ∀ cs0 abi_shard is_syscall_query ts1 ts2
___1 ___2 ___3 ___4 _tag,
      let is_system := addr_is_kernel dest && is_syscall_query in
      let allow_masking := negb is_system in
      let callee_shard := if is_shard_provided then abi_shard else
current_shard cs0 in
      paid_code_fetch allow_masking callee_shard (gs_depot gs) (gs_contracts
gs) dest cs0 (CodeFetchUnaffordable ___1) →
      cs0 = gs_callstack ts1 →
      ts2 = ts1 <| gs_status := Panic FarCallNotEnoughErgsToDecommit |> →
      farcall
        (mk_pv _tag {|
          FarCallABI.fwd_memory := ___2;
          ergs_passed := ___3;
          FarCallABI.shard_id := abi_shard;
          constructor_call := ___4;
          to_system := is_syscall_query;
        |})
       ts1 ts2
```

4. Paid for decommitment; Returning a new fat pointer, but not enough ergs to pay for memory growth.

```
  | farcall_fwd_new_ptr_growth_unaffordable: ∀ cs0 cs1 ___1 ___2 abi_shard
ergs_query is_syscall_query in_span page_type bound growth_query ts1 ts2,
```

```
      let is_system := addr_is_kernel dest && is_syscall_query in
      let allow_masking := negb is_system in
      let callee_shard := if is_shard_provided then abi_shard else
current_shard cs0 in

      paid_code_fetch allow_masking callee_shard
        gs.(gs_revertable).(gs_depot) gs.(gs_contracts) dest cs0 (Fetched cs1
___1 ___2) →

      bound_of_span in_span page_type bound →
      growth_to_bound bound cs1 growth_query →
      affordable cs1 (cost_of_growth growth_query) = false →

      cs0 = gs_callstack ts1 →
      ts2 = ts1 <| gs_status := Panic FarCallNotEnoughErgsToGrowMemory |> →
      farcall
        (IntValue {|
          FarCallABI.fwd_memory := ForwardNewFatPointer page_type in_span;
          ergs_passed := ergs_query;
          FarCallABI.shard_id := abi_shard;
          constructor_call := false;
          to_system := is_syscall_query;
          |})
       ts1 ts2
  .
```

Not formalized

- system contracts + constructor calls + “call in now constructed system contract” exception

```
End FarCallDefinitions.

Inductive step_farcall : instruction → smallstep :=

| step_farcall_normal: ∀ handler pv_abi (dest:word) call_shard call_as_static
s1 s2 ts1 ts2 (__:bool),

    let dest_addr := low contract_address_bits dest in
    let handler_code_addr := low code_address_bits handler in
    farcall Normal call_as_static call_shard dest_addr handler_code_addr
s1.(gs_global) pv_abi ts1 ts2 →
    step_transient_only ts1 ts2 s1 s2 →
    step_farcall (OpFarCall (Some pv_abi) (mk_pv __ dest) handler call_shard
call_as_static) s1 s2
| step_farcall_mimic: ∀ handler pv_abi (dest:word) call_shard call_as_static s1
s2 ts1 ts2 (__:bool),

    let dest_addr := low contract_address_bits dest in
    let handler_code_addr := low code_address_bits handler in
    farcall Mimic call_as_static call_shard dest_addr handler_code_addr
s1.(gs_global) pv_abi ts1 ts2 →
    step_transient_only ts1 ts2 s1 s2 →
```

```
    step_farcall (OpMimicCall (Some pv_abi) (mk_pv __ dest) handler call_shard
call_as_static) s1 s2
| step_farcall_delegate: ∀ handler pv_abi (dest:word) call_shard call_as_static
s1 s2 ts1 ts2 (__:bool),

    let dest_addr := low contract_address_bits dest in
    let handler_code_addr := low code_address_bits handler in
    farcall Delegate call_as_static call_shard dest_addr handler_code_addr
s1.(gs_global) pv_abi ts1 ts2 →
    step_transient_only ts1 ts2 s1 s2 →
    step_farcall (OpDelegateCall (Some pv_abi) (mk_pv __ dest) handler
call_shard call_as_static) s1 s2
.
End FarCalls.
```

Library EraVM.sem.FarRet

```
From RecordUpdate Require Import RecordSet.

Require
ABI
Bool
CallStack
Coder
Common
Flags
GPR
MemoryContext
MemoryManagement
Pointer
PrimitiveValue
SemanticCommon
State.

Import
ABI
FatPointerABI
Bool
CallStack
Coder
Common
Flags
GPR
isa.CoreSet
MemoryContext
MemoryManagement
Pointer
PrimitiveValue
RecordSetNotations
SemanticCommon
State
StepPanic
.

Section FarRetDefinition.
```

```
Let reserve regs :=
 regs <| r2 := reserved |> <| r3 := reserved |> <| r4 := reserved |>.

Inductive step_farret: instruction → tsmallstep :=
```

Far return (normal return, not panic/revert)

Abstract Syntax

`OpFarRet (args: in_reg)`

Syntax

`ret in1`

A normal return from a **far** call. Will pop up current callframe, return unspent ergs to the caller, and continue execution from the saved return address (from where the call had taken place). The register `args` describes a span of memory passed to the external caller.

The assembler expands `ret` to `ret r1`; `r1` is ignored by returns from near calls.

Semantic

1. Fetch the value from register `args` and decode the value of type `fwd_memory`:

   ```
   Inductive fwd_memory :=
     ForwardFatPointer (p:fat_ptr)
   | ForwardNewFatPointer (heap_var: data_page_type) (s:span).
   ```

   The exact encoding is described by ABI.

2. Forward memory to the caller (see `paid_forward`):

   - If `args` is `ForwardFatPointer p`, an existing `fat_ptr` is forwarded:

     ensure that the register containing `args` is tagged as a pointer.
     ensure that the memory page of `p` does NOT refer to a page owned by an older frame.
     `fat_ptr_narrow p` so it starts at its current offset, and the offset is reset to zero.

     There is no payment because the existing fat pointer has already been paid for.

     Attention: **shrinking** and **narrowing** far pointers are different. See `fat_ptr_shrink` and `fat_ptr_narrow`.

- If `args` is `ForwardNewFatPointer heap_variant [start; limit)`, a new `fat_ptr` is created:

  let B be the bound of the `heap_variant` taken from `ctx_heap_bound` field of `ecf_mem_ctx` of the `active_extframe`.

  let I be the page id of the `heap_variant` taken from `ctx_heap_page_id` field of `ecf_mem_ctx` of the `active_extframe`.

  build a fat pointer P from the span as described in `paid_forward_heap_span`.

  $$P := (I, (start, limit), 0)$$

  if $start + limit > B$, pay for the growth difference (`growth_cost` $(start + limit - B)$).

  Note: it is not useful to readjust the current heap/auxheap bounds after paying for growth. The bounds are part of `mem_ctx` of the topmost frame, which is about to be discarded.

3. Return the remaining ergs from `cf_ergs_remaining` of the destroyed frame to the caller.
4. Clear flags.
5. Encode P and store it in register `r1`, setting its pointer tag.

   All other registers are zeroed. Registers `r2`, `r3`, and `r4` are reserved and may gain a special meaning in newer versions of EraVM.

6. Clear context register.

```
| step_RetExt_heapvar:
  ∀ pages cf caller_stack cs1 new_caller new_regs ___1 ___2 ___3 out_ptr
heap_type hspan params s1 s2 status enc_ptr,
    let cs0 := ExternalCall cf (Some caller_stack) in

    paid_forward_new_fat_ptr heap_type hspan cs0 (out_ptr, cs1) →
    ergs_return_caller_and_drop cs1 new_caller →
    params = FarRetABI.mk_params (ForwardNewFatPointer heap_type hspan) →
    Some enc_ptr = encode_fat_ptr_word zero128 (NotNullPtr out_ptr) →
    new_regs = (reserve regs_state_zero)
                            <| r1 := PtrValue enc_ptr |> →
    step_transient_only {|
                          gs_flags := ___1 ;
                          gs_callstack := cs0;
                          gs_regs := ___2;
                          gs_context_u128 := ___3;


                          gs_pages := pages;
                          gs_status := status;
                        |}
```

```
                        {|
                          gs_flags := flags_clear;
                          gs_callstack := new_caller;
                          gs_regs := new_regs;
                          gs_context_u128 := zero128;


                          gs_pages := pages;
                          gs_status := status;
                           |} s1 s2 →
    step_farret (OpFarRet (Some (IntValue params))) s1 s2

  | step_RetExt_ForwardFatPointer:
  ∀ pages cf caller_stack cs1 new_caller new_regs ___1 ___2 ___3 in_ptr out_ptr
page params s1 s2 status enc_ptr,
    let cs0 := ExternalCall cf (Some caller_stack) in

    in_ptr.(fp_page) = Some page →

    page_older page (get_mem_ctx cs0) = false→

    validate in_ptr = no_exceptions →

    fat_ptr_narrow in_ptr out_ptr →

    ergs_return_caller_and_drop cs1 new_caller →
    params = FarRetABI.mk_params (ForwardExistingFatPointer (NotNullPtr
in_ptr)) →
    Some enc_ptr = encode_fat_ptr_word zero128 (NotNullPtr out_ptr) →
    new_regs = (reserve regs_state_zero)
                          <| r1 := PtrValue enc_ptr |> →
    step_transient_only {|
                          gs_flags := ___1 ;
                          gs_callstack := cs0;
                          gs_regs := ___2;
                          gs_context_u128 := ___3;


                          gs_pages := pages;
                          gs_status := status;
                        |}
                        {|
                          gs_flags := flags_clear;
                          gs_callstack := new_caller;
                          gs_regs := new_regs;

                          gs_context_u128 := zero128;
                          gs_pages := pages;
                          gs_status := status;
                          |} s1 s2 →
    step_farret (OpFarRet (Some (PtrValue params))) s1 s2
```

Affected parts of VM state

- Flags are cleared.
- Context register is zeroed (only returns from far calls).
- Registers are cleared (only returns from far calls).
- Execution stack:
  - Current frame is dropped.
  - Caller frame:
    Unspent ergs are given back to caller (but memory growth is paid first).

Usage

Similar instructions

- `revert` executes the current frame's exception handler instead of returning to the caller.
- `panic` executes the current frame's exception handler instead of returning to the caller, and sets overflow flag.

Panics

1. Attempt to forward an existing fat pointer, but the value holding `RetABI` is not tagged as a pointer.

```
| step_RetExt_ForwardFatPointer_requires_ptrtag:
∀ cf caller_stack params ___1 ___2 (s1 s2:state),
  let cs0 := ExternalCall cf (Some caller_stack) in
  gs_callstack s1 = cs0 →
  params = FarRetABI.mk_params (ForwardExistingFatPointer ___1) →
  step_panic
    RetABIExistingFatPointerWithoutTag
    s1 s2 →
  step_farret (OpFarRet (Some (IntValue ___2))) s1 s2
```

2. Attempt to return a pointer created before the current callframe. It is forbidden to pass a pointer to a contract in a far call and return it back. Otherwise we could create a `fat_ptr` P to a heap page of contract A, pass it to a contract B, return it back to A, and then modify the contents on the heap page of A. This way we will also modify the memory `slice` associated with P.

In other words, this is a situation where:

- caller makes far call to some contract;
- callee does return-forward @calldataptr;

- caller modifies calldata corresponding heap region, that leads to modification of returndata.

```
  | step_RetExt_ForwardFatPointer_returning_older_pointer:
  ∀ cf caller_stack in_ptr page params (s1 s2:state) _tag,
    let cs0 := ExternalCall cf (Some caller_stack) in
    gs_callstack s1 = cs0 →

    in_ptr.(fp_page) = Some page →

    page_older page (get_mem_ctx cs0) = true →
    params = FarRetABI.mk_params (ForwardExistingFatPointer (NotNullPtr
in_ptr)) →
    step_panic
      RetABIReturnsPointerCreatedByCaller
      s1 s2 →
    step_farret (OpFarRet (Some (mk_pv _tag params))) s1 s2
```

3. Attempt to return a malformed pointer.

```
  | step_RetExt_ForwardFatPointer_returning_malformed_pointer:
  ∀ cf caller_stack _tag (in_ptr: fat_ptr) params (s1 s2:state) ,
    let cs0 := ExternalCall cf (Some caller_stack) in
    gs_callstack s1 = cs0 →
    validate in_ptr ≠ no_exceptions →

    params = FarRetABI.mk_params (ForwardExistingFatPointer (NotNullPtr
in_ptr)) →
    step_panic
      FatPointerMalformed
      s1 s2 →
    step_farret (OpFarRet (Some (mk_pv _tag params))) s1 s2
```

4. Attempt to return a new pointer but unable to pay for memory growth.

```
| step_RetExt_heapvar_growth_unaffordable:
  ∀ cf caller_stack _tag heap_type hspan params (s1 s2:state),
    let cs0 := ExternalCall cf (Some caller_stack) in
    gs_callstack s1 = cs0 →
    params = FarRetABI.mk_params (ForwardNewFatPointer heap_type hspan) →
    growth_to_span_unaffordable cs0 heap_type hspan →
    step_panic
      FatPointerCreationUnaffordable
      s1 s2 →
    step_farret (OpFarRet (Some (mk_pv _tag params))) s1 s2
  .

End FarRetDefinition.
```

Library EraVM.sem.FarRevert

```
From RecordUpdate Require Import RecordSet.

Require
ABI
Bool
CallStack
Coder
Common
Flags
GPR
MemoryContext
MemoryManagement
Pointer
PrimitiveValue
SemanticCommon
State.

Import
ABI
FatPointerABI
Bool
CallStack
Coder
Common
Flags
GPR
isa.CoreSet
MemoryContext
MemoryManagement
Pointer
PrimitiveValue
RecordSetNotations
SemanticCommon
State
StepPanic
.

Section FarRevertDefinition.

  Let reserve regs :=
   regs <| r2 := reserved |> <| r3 := reserved |> <| r4 := reserved |>.

  Inductive step_farrevert: instruction → smallstep :=
```

Far revert (return from recoverable error, not panic/ normal return)

Abstract Syntax

`OpFarRevert (args: in_reg)`

Syntax

`ret.revert in1` aliased as `revert in1`

An abnormal return from a far call. Will pop up current callframe, give back unspent ergs and execute a currently active exception handler. The register abi_reg describes a slice of memory passed to the external caller.

Restores storage to the state before external call.

The assembler expands `revert` to `revert r1`; `r1` is ignored by returns from near calls.

Semantic

1. Let E be the address of the `active_exception_handler`.
2. Perform a `rollback`.
3. Proceed with the same steps as `OpFarRetABI` (see `step_farret`).
4. Set PC to E.

```
  | step_RevertExt_heapvar:
    ∀ gs gs' pages cf caller_stack cs1 cs2 new_caller new_regs params out_ptr
heap_type hspan ___2 ___3 ___4 _tag ptr_enc,
      let cs0 := ExternalCall cf (Some caller_stack) in

      params = ForwardNewFatPointer heap_type hspan →

      paid_forward_new_fat_ptr heap_type hspan cs0 (out_ptr, cs1) →
      ergs_return_caller_and_drop cs1 cs2 →

      new_caller = pc_set (active_exception_handler cs0) cs2 →

      rollback cf.(cf_saved_checkpoint) gs gs' →
      Some ptr_enc = encode_fat_ptr_word zero128 (NotNullPtr out_ptr) →
      new_regs = reserve (regs_state_zero <| r1 := PtrValue ptr_enc |> )→
```

```
      step_farrevert (OpFarRevert (Some (mk_pv _tag (FarRetABI.mk_params
params))))
                     {|
                       gs_transient := {|
                                        gs_flags := ___2 ;
                                        gs_callstack := cs0;
                                        gs_regs := ___3;
                                        gs_context_u128 := ___4;

                                        gs_pages := pages;
                                        gs_status := NoPanic;
                                      |};
                       gs_global := gs;
                     |}
                     {|
                       gs_transient := {|
                                        gs_flags := flags_clear;
                                        gs_callstack := new_caller;
                                        gs_regs := new_regs;
                                        gs_context_u128 := zero128;

                                        gs_pages := pages;
                                        gs_status := NoPanic;
                                      |};
                       gs_global := gs';
                     |}
  | step_RevertExt_ForwardFatPointer:
    ∀ pages cf caller_stack cs1 cs2 new_caller new_regs ___2 ___3 ___4 in_ptr
out_ptr page params gs gs' ptr_enc,
      let cs0 := ExternalCall cf (Some caller_stack) in

      (* Panic if not a pointer *)

      params = ForwardExistingFatPointer (NotNullPtr in_ptr) →
      in_ptr.(fp_page) = Some page →

      MemoryContext.page_older page (get_mem_ctx cs0) = false →

      validate in_ptr = no_exceptions →

      fat_ptr_narrow in_ptr out_ptr →

      ergs_return_caller_and_drop cs1 cs2 →

      new_caller = pc_set (active_exception_handler cs0) cs2 →
      Some ptr_enc = encode_fat_ptr_word zero128 (NotNullPtr out_ptr) →
      new_regs = (reserve regs_state_zero) <| r1 := PtrValue ptr_enc |> →
      rollback cf.(cf_saved_checkpoint) gs gs' →
      step_farrevert (OpFarRevert (Some (PtrValue (FarRetABI.mk_params
params))))
                     {|
                       gs_transient := {|
                                        gs_flags := ___2;
                                        gs_callstack := cs0;
                                        gs_regs := ___3 ;
                                        gs_context_u128 := ___4;
```

```
                    gs_pages := pages;
                    gs_status := NoPanic;
                  |};

  gs_global := gs;
|}
{|
  gs_transient := {|
                    gs_flags := flags_clear;
                    gs_callstack := new_caller;
                    gs_regs := new_regs;
                    gs_context_u128 := zero128;

                    gs_pages := pages;
                    gs_status := NoPanic;
                  |};

  gs_global := gs';
|}
```

Affected parts of VM state

- Flags are cleared.
- Context register is zeroed (only returns from far calls).
- Registers are cleared (only returns from far calls).
- Execution stack:
  - Current frame is dropped.
  - Caller frame:
    if a label is explicitly provided, and current frame is internal (near call), then caller's PC is overwritten with the label. Returns from external calls ignore label, even if it is explicitly provided.
    Unspent ergs are given back to caller (but memory growth is paid first).
- Storage changes are reverted.

Usage

- Abnormal returns from near/far calls when a recoverable error happened. Use `panic` for irrecoverable errors.

Similar instructions

- `ret` returns to the caller instead of executing an exception handler.
- `panic` acts similar to `revert` but does not let pass any data to the caller and sets an overflow flag, and burns ergs in current frame.

Panics

1. Attempt to forward an existing fat pointer, but the value holding `RetABI` is not tagged as a

pointer.

```
| step_RevertExt_ForwardFatPointer_requires_ptrtag:
∀ cf caller_stack __ params (s1 s2:state),
  let cs0 := ExternalCall cf (Some caller_stack) in
  gs_callstack s1 = cs0 →
  params = FarRetABI.mk_params (ForwardExistingFatPointer __) →
  step_panic
    RetABIExistingFatPointerWithoutTag
    s1 s2 →
  step_farrevert (OpFarRevert (Some (IntValue params))) s1 s2
```

2. Attempt to return a pointer created before the current callframe. It is forbidden to pass a pointer to a contract in a far call and return it back. Otherwise we could create a `fat_ptr` P to a heap page of contract A, pass it to a contract B, return it back to A, and then modify the contents on the heap page of A. This way we will also modify the memory `slice` associated with P.

In other words, this is a situation where:

- caller makes far call to some contract;
- callee does return-forward @calldataptr;
- caller modifies calldata corresponding heap region, that leads to modification of returndata.

```
  | step_RevertExt_ForwardFatPointer_returning_older_pointer:
  ∀ cf caller_stack in_ptr page params (s1 s2:state) ,
    let cs0 := ExternalCall cf (Some caller_stack) in
    gs_callstack s1 = cs0 →

    in_ptr.(fp_page) = Some page →

    page_older page (get_mem_ctx cs0) = true →
    params = FarRetABI.mk_params (ForwardExistingFatPointer (NotNullPtr
in_ptr)) →
    step_panic
      RetABIReturnsPointerCreatedByCaller
      s1 s2 →
    step_farrevert (OpFarRevert (Some (PtrValue params))) s1 s2
```

3. Attempt to return a malformed pointer.

```
  | step_RevertExt_ForwardFatPointer_returning_malformed_pointer:
  ∀ cf caller_stack (in_ptr: fat_ptr) params (s1 s2:state) ,
    let cs0 := ExternalCall cf (Some caller_stack) in
    gs_callstack s1 = cs0 →

    validate in_ptr ≠ no_exceptions →
```

```
    params = FarRetABI.mk_params (ForwardExistingFatPointer (NotNullPtr
in_ptr)) →
    step_panic
      FatPointerMalformed
      s1 s2 →
    step_farrevert (OpFarRevert (Some (PtrValue params))) s1 s2
```

4. Attempt to return a new pointer but unable to pay for memory growth.

```
| step_RevertExt_heapvar_growth_unaffordable:
  ∀ cf caller_stack heap_type hspan params (s1 s2:state),
    let cs0 := ExternalCall cf (Some caller_stack) in
    gs_callstack s1 = cs0 →
    params = FarRetABI.mk_params (ForwardNewFatPointer heap_type hspan) →
    growth_to_span_unaffordable cs0 heap_type hspan →
    step_panic
      FatPointerCreationUnaffordable
      s1 s2 →
    step_farrevert (OpFarRevert (Some (IntValue params))) s1 s2.

End FarRevertDefinition.
```

Library EraVM.sem.LoadPtr

```
Require SemanticCommon Slice.

Import TransientMemory MemoryOps isa.CoreSet Pointer SemanticCommon
PrimitiveValue Slice State.

Section LoadPtrDefinition.

  Open Scope ZMod_scope.
  Inductive step_load_ptr : instruction → tsmallstep :=
```

LoadPointer

Abstract Syntax

```
OpLoadPointer (ptr: in_reg) (res: out_reg)
```

Syntax

- `uma.fat_ptr_read in1, out` aliased as `ld in1, out`

Summary

Read 32 consecutive bytes from address provided by a fat pointer `ptr` of active `heap` or `aux_heap` page as a 256-bit word, Big Endian. Reading bytes past the slice bound yields zero bytes.

Semantic

1. Let `in_ptr = (page, start, offset, length)` be a `fat_ptr` decoded from `in1`. Requires that `in1` is tagged as a pointer.

2. Validate that offset is in bounds: `offset < length`.

3. Read 32 consecutive bytes as a Big Endian 256-bit word from address `offset` in heap variant.

   Reading bytes past `start + length` returns zero bytes. For example, consider a pointer with:

   ```
   {|
   page   := _;
   start  := 0;
   length := 5;
   offset := 2
   |}
   ```

   Reading will produce a word with 3 most significant bytes read from memory fairly (addresses 2, 3, 4) and 29 zero bytes coming from attempted reads past `fp_start + fp_length` bound.

4. Store the word to `res`.

```
  | step_LoadPointer:
    ∀ result _flags _regs mem _cs _ctx addr selected_page (in_ptr:fat_ptr)
slice page_id high128,
      validate_in_bounds in_ptr = true →
      page_id = in_ptr.(fp_page) →
      page_has_id mem page_id (mk_page (DataPage selected_page)) →
      slice_page selected_page in_ptr slice →

      ptr_resolves_to in_ptr addr →
      mb_load_slice_result BigEndian slice addr result →

      step_load_ptr (OpLoadPointer (Some (PtrValue (high128, NotNullPtr
in_ptr))) (IntValue result))
                    (mk_transient_state _flags _regs mem _cs _ctx NoPanic)
```

```
(mk_transient_state _flags _regs mem _cs _ctx NoPanic)
```

Affected parts of VM state

- execution stack: PC, as by any instruction;
- GPRs, because `res` only resolves to a register.

Usage

- Read data from a read-only slice returned from a far call, or passed to a far call.
- One of few instructions that accept only reg or imm operand but do not have full addressing mode, therefore can't e.g. address stack. The full list is: `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc`.

Similar instructions

- `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc` are variants of the same instruction.

Panics

1. Argument is not a tagged pointer.

```
  | step_LoadPointer_not_tagged:
    ∀ ___ ____ (s1 s2:state),
      step_panic ExpectedFatPointer s1 s2 →
      step_load_ptr (OpLoadPointer (Some (IntValue ___)) ____) s1 s2
  .
End LoadPtrDefinition.
```

Library EraVM.sem.LoadPtrInc

```
Require SemanticCommon Slice.

Import Arith isa.CoreSet TransientMemory MemoryOps Pointer SemanticCommon
PrimitiveValue Slice State.

Section LoadPtrIncDefinition.
  Open Scope ZMod_scope.
```

LoadPointerInc

Abstract Syntax

```
OpLoadPointerInc (ptr: in_reg) (res: out_reg) (inc_ptr: out_reg)
```

Syntax

- `uma.fat_ptr_read.inc in1, out` aliased as `ld.inc in1, out`

Summary

Read 32 consecutive bytes from address `ptr` of active `heap` or `aux_heap` page as a 256-bit word, Big Endian. Reading bytes past the slice bound yields zero bytes.

Additionally, store a pointer to the next word to `inc_ptr` register.

Semantic

1. Let `in_ptr = (page, start, offset, length)` be a `fat_ptr` decoded from `in1`. Requires that `in1` is tagged as a pointer.

2. Validate that offset is in bounds: `offset < length`.

3. Read 32 consecutive bytes as a Big Endian 256-bit word from address `offset` in heap variant.

   Reading bytes past `start + length` returns zero bytes. For example, consider a pointer with:

   ```
   {|
   page   := _;
   start  := 0;
   length := 5;
   offset := 2
   |}
   ```

   Reading will produce a word with 3 most significant bytes read from memory fairly (addresses 2, 3, 4) and 29 zero bytes coming from attempted reads past `fp_start + fp_length` bound.

4. Store the word to `res`.

5. Store an encoded fat pointer to the next 32-byte word in heap variant in `inc_ptr`. Its fields are assigned as follows:

```
        page := in_ptr.(fp_page);
        start := in_ptr.(fp_page);
        length := in_ptr.(fp_length);
        offset := in_ptr.(fp_offset) + 32;
```

```
  Inductive step_load_ptr_inc : instruction → tsmallstep :=
  | step_LoadPointerInc:
    ∀ result addr selected_page (in_ptr:fat_ptr) out_ptr slice page_id s
high128,

      validate_in_bounds in_ptr = true →
      page_id = in_ptr.(fp_page) →

      page_has_id s.(gs_pages) page_id (mk_page (DataPage selected_page)) →
      slice_page selected_page in_ptr slice →

      ptr_resolves_to in_ptr addr →
      mb_load_slice_result BigEndian slice addr result →

      fat_ptr_inc in_ptr out_ptr →

      step_load_ptr_inc (OpLoadPointerInc (Some (PtrValue (high128, NotNullPtr
in_ptr))) (IntValue result) (Some (PtrValue (high128, NotNullPtr out_ptr)))) s
s
```

Affected parts of VM state

- execution stack: PC, as by any instruction;
- GPRs

Usage

- Read data from a read-only slice returned from a far call, or passed to a far call.
- One of few instructions that accept only reg or imm operand but do not have full addressing mode, therefore can't e.g. address stack. The full list is: `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc`.

Similar instructions

- `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc` are variants of the same instruction.

Panics

1. Argument is not a tagged pointer.

```
  | step_LoadPointerInc_not_tagged:
    ∀ ___ ____ _____ (s1 s2:state),
      step_panic ExpectedFatPointer s1 s2 →
      step_load_ptr_inc (OpLoadPointerInc (Some (IntValue ___)) ____ _____) s1
s2
```

2. Incremented pointer overflows.

```
  | step_LoadPointerInc_inc_overflow:
    ∀ in_ptr high128 ____ _____ (s1 s2:state),
      fat_ptr_inc_OF in_ptr = None →
      step_panic FatPtrIncOverflow s1 s2 →
      step_load_ptr_inc (OpLoadPointerInc (Some (PtrValue (high128, NotNullPtr
in_ptr))) ____ _____) s1 s2
.
End LoadPtrIncDefinition.
```

Library EraVM.sem.Load

```
Require SemanticCommon MemoryManagement.

Import Arith MemoryOps MemoryManagement isa.CoreSet Pointer SemanticCommon
PrimitiveValue State.

Section LoadDefinition.

  Open Scope ZMod_scope.

  Generalizable Variables cs flags regs mem.
  Inductive step_load: instruction → tsmallstep :=
```

Load

Abstract Syntax

```
OpLoad (ptr: in_regimm) (res: out_reg) (mem:data_page_type)
```

Syntax

- `uma.heap_read in1, out` aliased as `ld.1 in1, out`
- `uma.aux_heap_read in1, out` aliased as `ld.2 in1, out`

Summary

Decode the heap address from `in1`, load 32 consecutive bytes from the specified active heap variant.

Semantic

1. Decode a `heap_ptr` $addr$ from `ptr`.
2. Ensure reading 32 consecutive bytes is possible; for that, check if $addr < 2^{32} - 32$.
3. Let B be the selected heap variant bound. If $addr + 32 > B$, grow heap variant bound and pay for the growth. We are aiming at reading a 256-bit word starting from address $addr$ so the heap variant bound should contain all of it.
4. Read 32 consecutive bytes as a Big Endian 256-bit word from $addr$ in the heap variant, store result to `res`.

```
  | step_Load:
    ∀ new_cs heap_variant ctx result mem selected_page bound addr high224,
      `(
      addr ≤ MAX_OFFSET_TO_DEREF_LOW_U32 = true →

      heap_variant_page heap_variant cs1 mem selected_page →
      mb_load_result BigEndian selected_page addr result →

      word_upper_bound (mk_hptr addr) bound →
      bound_grow_pay (heap_variant, bound) cs0 new_cs →

      step_load (OpLoad (Some (IntValue (high224, mk_hptr addr))) (IntValue
result) heap_variant)
          {|
            gs_callstack := cs0;


            gs_regs := regs;
            gs_pages := mem;
            gs_flags := flags;
            gs_context_u128 := ctx;
           gs_status := NoPanic;
          |}
          {|
            gs_callstack := new_cs;
```

```
      gs_regs := regs;
      gs_pages := mem;
      gs_flags := flags;
      gs_context_u128 := ctx;
     gs_status := NoPanic;
    |}
  )
```

Affected parts of VM state

- execution stack:
  - PC, as by any instruction;
  - ergs allocated for the current function/contract instance, if the heap variant has to be grown;
  - heap variant bounds, if heap variant has to be grown.
- registers, because `res` only resolves to a register.

Usage

- Only `OpLoad` and `OpLoadInc` are capable of reading data from heap variants.
- One of few instructions that accept only reg or imm operand but do not have full addressing mode, therefore can't e.g. address stack. The full list is: `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc`.

Similar instructions

- `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc` are variants of the same `mach_instruction`.

Panics

1. Accessing an address greater than `MAX_OFFSET_TO_DEREF_LOW_U32`.

```
  | step_Load_offset_too_large:
    ∀ heap_variant ___ addr s1 s2 high224,
      `(
          addr > MAX_OFFSET_TO_DEREF_LOW_U32 = true →
          step_panic HeapPtrOffsetTooLarge s1 s2 →
          step_load (OpLoad (Some (IntValue (high224, mk_hptr addr))) ___
heap_variant) s1 s2
        )
```

2. Passed `fat_ptr` instead of `heap_ptr`.

```
  | step_Load_expects_intvalue:
    ∀ s1 s2 ___ ____ _____,
      `(
          step_panic ExpectedHeapPointer s1 s2 →
          step_load (OpLoad (Some (PtrValue ___)) ____ _____) s1 s2
        )
```

3. Accessing an address requires growing the bound of the corresponding heap variant, but the growth is unaffordable.

```
  | step_Load_growth_unaffordable:
    ∀ (s1 s2:state) cs ptr bound heap_variant ___ high224,
      `(
          word_upper_bound ptr bound →
          growth_to_bound_unaffordable cs (heap_variant, bound) →
          gs_callstack s1 = cs →
          step_panic HeapGrowthUnaffordable s1 s2 →
          step_load (OpLoad (Some (IntValue (high224, ptr ))) ___ heap_variant)
s1 s2
        )
End LoadDefinition.
```

Library EraVM.sem.LoadInc

```
Require SemanticCommon MemoryManagement.

Import Arith MemoryOps MemoryManagement isa.CoreSet Pointer SemanticCommon
PrimitiveValue State.

Section LoadIncDefinition.

  Open Scope ZMod_scope.
  Generalizable Variables cs flags regs mem.
  Inductive step_load_inc : instruction → tsmallstep :=
```

LoadInc

Abstract Syntax

`OpLoadInc (ptr: in_regimm) (res: out_reg) (mem:data_page_type) (inc_ptr: out_reg)`

Syntax

- `uma.inc.heap_read in1, out1, out2` aliased as `ld.1.inc in1, out1, out2`
- `uma.inc.aux_heap_read in1, out1, out2` aliased as `ld.2.inc in1, out1, out2`

Summary

Decode the heap address from `in1`, load 32 consecutive bytes from the specified active heap variant. Additionally, store a pointer to the next word to `inc_ptr` register.

Semantic

1. Decode a `heap_ptr` $addr$ from `ptr`.
2. Ensure reading 32 consecutive bytes is possible; for that, check if $addr < 2^{32} - 32$.
3. Let B be the selected heap variant bound. If $addr + 32 > B$, grow heap variant bound and pay for the growth. We are aiming at reading a 256-bit word starting from address $addr$ so the heap variant bound should contain all of it.
4. Read 32 consecutive bytes as a Big Endian 256-bit word from $addr$ in the heap variant, store result to `res`.
5. Store an encoded `heap_ptr` $addr + 32$ to the next 32-byte word in the heap variant in `inc_ptr`.

```
  | step_LoadInc:
    ∀ heap_variant result new_regs selected_page high224 ptr_inc bound new_cs
addr ctx,
      `(
      let hptr := mk_hptr addr in

      addr ≤ MAX_OFFSET_TO_DEREF_LOW_U32 = true →

      heap_variant_page heap_variant cs0 mem selected_page →
      mb_load_result BigEndian selected_page addr result →
```

```
      word_upper_bound hptr bound →
      bound_grow_pay (heap_variant, bound) cs0 new_cs →

      hp_inc hptr ptr_inc →

      step_load_inc (OpLoadInc (Some (IntValue (high224, hptr))) (IntValue
result) heap_variant (Some (IntValue (high224, ptr_inc))))
        (mk_transient_state flags regs mem cs0 ctx NoPanic)
        (mk_transient_state flags new_regs mem new_cs ctx NoPanic)
        )
```

Affected parts of VM state

- execution stack:
  - PC, as by any instruction;
  - allocated ergs if the heap variant has to be grown;
  - heap variant bounds, if heap variant has to be grown.
- GPRs, because `res` and `inc_ptr` only resolve to registers.

Usage

- Only `OpLoad` and `OpLoadInc` are capable of reading from heap variantheap.
- One of few instructions that accept only reg or imm operand but do not have full addressing mode, therefore can't e.g. address stack. The full list is: `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc`.

Similar instructions

- `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc` are variants of the same instruction.

Panics

1. Accessing an address greater than `MAX_OFFSET_TO_DEREF_LOW_U32`.

```
  | step_Load_offset_too_large:
    ∀ heap_variant ___ ____ addr s1 s2 high224,
      `(
          addr > MAX_OFFSET_TO_DEREF_LOW_U32 = true →
          step_panic HeapPtrOffsetTooLarge s1 s2 →
          step_load_inc (OpLoadInc (Some (IntValue (high224, mk_hptr addr)))
___ heap_variant ____) s1 s2
        )
```

2. Passed `fat_ptr` instead of `heap_ptr`.

```
  | step_Load_expects_intvalue:
    ∀ s1 s2 ___ ____ _____ ______,
      `(
         step_panic ExpectedHeapPointer s1 s2 →
         step_load_inc (OpLoadInc (Some (PtrValue ___)) ____ _____ ______) s1
s2
       )
```

3. Accessing an address requires growing the bound of the corresponding heap variant, but the growth is unaffordable.

```
  | step_Load_growth_unaffordable:
    ∀ (s1 s2:state) cs ptr bound heap_variant ___ ____ high224,
      `(
         word_upper_bound ptr bound →
         growth_to_bound_unaffordable cs (heap_variant, bound) →
         gs_callstack s1 = cs →
         step_panic HeapGrowthUnaffordable s1 s2 →
         step_load_inc (OpLoadInc (Some (IntValue (high224, ptr))) ___
heap_variant ____) s1 s2
       )
```

4. Incremented pointer overflows.

```
  | step_LoadInc_inc_overflow:
    ∀ (s1 s2:state) heap_variant result hptr ___ high224,
      `(
         hp_inc_OF hptr = None →
         step_panic HeapPtrIncOverflow s1 s2 →
         step_load_inc (OpLoadInc (Some (IntValue (high224, hptr))) (IntValue
result) heap_variant ___) s1 s2
       )
  .

End LoadIncDefinition.
```

Library EraVM.sem.Store

```
Require SemanticCommon MemoryManagement.
Import Core Common TransientMemory MemoryOps MemoryManagement isa.CoreSet State
  SemanticCommon Pointer PrimitiveValue.

Section StoreDefinition.
```

```
Open Scope ZMod_scope.

Inductive step_store: instruction → tsmallstep :=
```

Store

Abstract Syntax

`OpStore (ptr: in_regimm) (val: in_reg) (mem:data_page_type) (swap: mod_swap)`

Syntax

- `uma.heap_write in1, in2` aliased as `st.1.inc in1, out`
- `uma.aux_heap_write in1, in2` aliased as `st.2.inc in1, out`

Summary

Decode the heap address from `in1`, load 32 consecutive bytes from the specified active heap variant.

Semantic

1. Decode a `heap_ptr` $addr$ from `ptr`.
2. Ensure storing 32 consecutive bytes is possible; for that, check if $addr < 2^{32} - 32$.
3. Let B be the selected heap variant bound. If $addr + 32 > B$, grow heap variant bound and pay for the growth. We are aiming at reading a 256-bit word starting from address $addr$ so the heap variant bound should contain all of it.
4. Store 32 consecutive bytes as a Big Endian 256-bit word from `val` to $addr$ in the heap variant.

```
  | step_Store:
    ∀ high224 result flags new_cs heap_variant value new_mem selected_page
bound modified_page cs regs mem addr ctx,

      let selected_page_id := heap_variant_id heap_variant cs in

      addr ≤ MAX_OFFSET_TO_DEREF_LOW_U32 = true →

      heap_variant_page heap_variant cs mem selected_page →
```

```
      word_upper_bound (mk_hptr addr) bound →
      bound_grow_pay (heap_variant, bound) cs new_cs →

      mb_store_word_result BigEndian selected_page addr value modified_page →

      page_replace selected_page_id (mk_page (DataPage modified_page)) mem
new_mem →

      step_store (OpStore (Some (IntValue (high224, mk_hptr addr))) (IntValue
result) heap_variant)
           {|
             gs_callstack := cs;
             gs_pages := mem;


             gs_regs := regs;
             gs_flags := flags;
             gs_context_u128 := ctx;
             gs_status := NoPanic;
           |}
           {|
             gs_callstack := new_cs;
             gs_pages := new_mem;


             gs_regs := regs;
             gs_flags := flags;
             gs_context_u128 := ctx;
             gs_status := NoPanic;
           |}
```

Affected parts of VM state

- execution stack:
  - PC, as by any instruction;
  - allocated ergs if the heap variant has to be grown;
  - heap bounds, if heap variant has to be grown.
- GPRs, because `out` only resolves to a register.
- TransientMemory page

Usage

- Only `OpLoad` and `OpLoadInc` are capable of reading data from heap variant.
- One of few instructions that accept only reg or imm operand but do not have full addressing mode, therefore can't e.g. address stack. The full list is: `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc`.

Similar instructions

- `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc` are variants of the same instruction.

Panics

1. Accessing an address greater than `MAX_OFFSET_TO_DEREF_LOW_U32`.

```
  | step_Store_offset_too_large:
    ∀ heap_variant ___1 addr high224 s1 s2,
      `(
          addr > MAX_OFFSET_TO_DEREF_LOW_U32 = true →
          step_panic HeapPtrOffsetTooLarge s1 s2 →
          step_store (OpStore (Some (IntValue (high224, mk_hptr addr))) ___1
heap_variant) s1 s2
        )
```

2. Fat pointer provided where heap pointer is expected.

```
  | step_Store_expects_intvalue:
    ∀ s1 s2 ___1 ___2 ___3,
      `(
          step_panic ExpectedHeapPointer s1 s2 →
          step_store (OpStore (Some (PtrValue ___1)) ___2 ___3) s1 s2
        )
```

3. Accessing an address requires growing the bound of the corresponding heap variant, but the growth is unaffordable.

```
  | step_Store_growth_unaffordable:
    ∀ (s1 s2:state) high224 cs ptr bound heap_variant ___1,
      `(
          word_upper_bound ptr bound →
          growth_to_bound_unaffordable cs (heap_variant, bound) →
          gs_callstack s1 = cs →
          step_panic HeapGrowthUnaffordable s1 s2 →
          step_store (OpStore (Some (IntValue (high224, ptr))) ___1
heap_variant) s1 s2
        )
  .
End StoreDefinition.
```

Library EraVM.sem.StoreInc

```
Require SemanticCommon MemoryManagement.
Import Core Common TransientMemory MemoryOps MemoryManagement isa.CoreSet State
  SemanticCommon Pointer PrimitiveValue.

Section StoreIncDefinition.
 Open Scope ZMod_scope.

 Inductive step_storeinc: instruction → tsmallstep :=
```

StoreInc

Abstract Syntax

```
OpStoreInc (ptr: in_regimm) (val: in_reg) (mem:data_page_type) (inc_ptr:
out_reg) (swap: mod_swap)
```

Syntax

- `uma.inc.heap_write in1, in2` aliased as `st.1.inc in1, out`
- `uma.inc.aux_heap_write in1, in2` aliased as `st.2.inc in1, out`

Summary

Decode the heap address from `in1`, load 32 consecutive bytes from the specified active heap variant. Additionally, store a pointer to the next word to `inc_ptr` register.

Semantic

1. Decode a `heap_ptr` $addr$ from `ptr`.
2. Ensure storing 32 consecutive bytes is possible; for that, check if $addr < 2^{32} - 32$.
3. Let B be the selected heap variant bound. If $addr + 32 > B$, grow heap variant bound and pay for the growth. We are aiming at reading a 256-bit word starting from address $addr$ so the heap variant bound should contain all of it.
4. Store 32 consecutive bytes as a Big Endian 256-bit word from `val` to $addr$ in the heap variant.

5. Store an encoded `heap_ptr` $addr+32$ to the next 32-byte word in the heap variant in `inc_ptr`.

```
  | step_StoreInc:
    ∀ hptr flags new_cs heap_variant value new_mem selected_page bound
modified_page cs regs mem ___1 addr hptr_mod ctx high224,

      let selected_page_id := heap_variant_id heap_variant cs in

      hptr = mk_hptr addr →

      addr ≤ MAX_OFFSET_TO_DEREF_LOW_U32 = true →

      heap_variant_page heap_variant cs mem selected_page →

      word_upper_bound hptr bound →
      bound_grow_pay (heap_variant, bound) cs new_cs →

      mb_store_word_result BigEndian selected_page addr value modified_page →
      page_replace selected_page_id (mk_page (DataPage modified_page)) mem
new_mem →

      hp_inc hptr hptr_mod →

      step_storeinc (OpStoreInc (Some (IntValue (high224, hptr))) (mk_pv ___1
value) heap_variant (Some (IntValue (high224,hptr_mod))))
           {|
             gs_callstack := cs;
             gs_pages := mem;


             gs_regs := regs;
             gs_flags := flags;
             gs_context_u128 := ctx;
             gs_status := NoPanic;
           |}
           {|
             gs_callstack := new_cs;
             gs_pages := new_mem;


             gs_regs := regs;
             gs_flags := flags;
             gs_context_u128 := ctx;
             gs_status := NoPanic;
           |}
```

Affected parts of VM state

- execution stack:
  - PC, as by any instruction;
  - allocated ergs if the heap variant has to be grown;
  - heap variant bounds, if heap variant has to be grown.
- GPRs, because `res` and `inc_ptr` only resolve to registers.

Usage

- Only `OpStore` and `OpStoreInc` are capable of writing to heap variant.
- One of few instructions that accept only reg or imm operand but do not have full addressing mode, therefore can't e.g. address stack. The full list is: `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc`.

Similar instructions

- `OpLoad`, `OpLoadInc`, `OpStore`, `OpStoreInc`, `OpLoadPointer`, `OpLoadPointerInc` are variants of the same instruction.

Panics

1. Accessing an address greater than `MAX_OFFSET_TO_DEREF_LOW_U32`.

```
  | step_Store_offset_too_large:
    ∀ heap_variant ___1 ___2 addr s1 s2 high224,
      `(
          addr > MAX_OFFSET_TO_DEREF_LOW_U32 = true →
          step_panic HeapPtrOffsetTooLarge s1 s2 →
          step_storeinc (OpStoreInc (Some (IntValue (high224, mk_hptr addr)))
___1 heap_variant ___2) s1 s2
       )
```

2. Fat pointer provided where heap pointer is expected.

```
  | step_Store_expects_intvalue:
    ∀ s1 s2 ___1 ___2 ___3 ___4,
      `(
          step_panic ExpectedHeapPointer s1 s2 →
          step_storeinc (OpStoreInc (Some (PtrValue ___1)) ___2 ___3 ___4) s1
s2
       )
```

3. Accessing an address requires growing the bound of the corresponding heap variant, but the growth is unaffordable.

```
  | step_Store_growth_unaffordable:
    ∀ (s1 s2:state) cs hptr bound heap_variant ___1 ___2 high242,
      `(
          word_upper_bound hptr bound →
          growth_to_bound_unaffordable cs (heap_variant, bound) →
          gs_callstack s1 = cs →
          step_panic HeapGrowthUnaffordable s1 s2 →
          step_storeinc (OpStoreInc (Some (IntValue (high242, hptr))) ___1
heap_variant ___2) s1 s2
        )
```

4. Incrementing the pointer leads to overflow.

```
  | step_Store_inc_overflow:
    ∀ (s1 s2:state) hptr ___1 ___2 ___3 high242,
      `(
          hp_inc_OF hptr = None →
          step_panic HeapGrowthUnaffordable s1 s2 →
          step_storeinc (OpStoreInc (Some (IntValue (high242, hptr))) ___1 ___2
 3) s1 s2
___
        )
      .
End StoreIncDefinition.
```

Library EraVM.sem.PtrAdd

```
Require SemanticCommon.

Import Arith Core TransientMemory MemoryBase Pointer PrimitiveValue
SemanticCommon State isa.CoreSet spec.

Section PtrAddDefinition.
  Open Scope ZMod_scope.
```

PtrAdd

Abstract Syntax

```
OpPtrAdd (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
```

Syntax

- `ptr.add in1, in2, out`
- `ptr.add.s in1, in2, out`

Summary

Takes a fat pointer from `in1` and a 32-bit unsigned number from `in2`. Advances the fat pointer's offset by that number, and writes (`in2`{128…255} || incremented pointer) to `out`.

Semantic

1. Fetch input operands, swap them if `swap` modifier is set. Now operands are op_1 and op_2.
2. Ensure the op_1 is tagged as a pointer, and op_2 is not tagged as a pointer. Otherwise panic.
3. Decode fat pointer ptr_{in} from op_1
4. Let $diff$ be op_2 truncated to 32 bits:

   $$diff := op_2 \mod 2^{32}$$

   It is required that $op_2 < \texttt{MAX\_OFFSET\_FOR\_ADD\_SUB}$, otherwise VM panics.
5. Advance pointer offset of ptr_{in} by $diff$:

   $$ptr_{out} := ptr_{in}|_{offset := offset + diff}$$
6. Store the result, tagged as a pointer, to `out`. The most significant 128 bits of result are taken from `op1`, the least significant bits hold an encoded pointer:

   $$result := op_1\{255 \ldots 128\} \# \# \textbf{encode}(ptr_{out})$$

```
  Inductive step_ptradd : instruction → smallstep :=
  | step_PtrAdd:
    ∀ s ofs new_ofs pid high128 (arg_delta:word) (mem_delta: mem_address) span,

      arg_delta < MAX_OFFSET_FOR_ADD_SUB = true →
      mem_delta = low mem_address_bits arg_delta →
      (false, new_ofs) = ofs + mem_delta →

      step_ptradd (OpPtrAdd
              (Some (PtrValue (high128, NotNullPtr (mk_fat_ptr pid (mk_ptr span
ofs))))))
              (IntValue arg_delta)
              (Some (PtrValue (high128, NotNullPtr (mk_fat_ptr pid (mk_ptr span
```

```
new_ofs)))))
        s s
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- Flags are unaffected

Usage

- Manipulating fat pointers to pass slices of memory between functions.

Similar instructions

- Takes part in a group of pointer manipulating instructions:
  - `OpPtrAdd`
  - `OpPtrSub`
  - `OpPtrShrink`
  - `OpPtrPack`
- Instruction `OpPtrSub` effectively performs the same actions but the offset is negated.

Encoding

Instructions `OpPtrAdd`, `OpPtrSub`, `OpPtrPack` and `OpPtrShrink` are sharing an opcode.

Panics

1. First argument is not a pointer (after accounting for `swap`).

```
| step_PtrAdd_in1_not_ptr:
  ∀ s1 s2 ___ ____ _____,
    step_panic ExpectedFatPointer s1 s2 →
    step_ptradd (OpPtrAdd (Some (IntValue ___)) ____ _____) s1 s2
```

2. Second argument is a pointer (after accounting for `swap`).

```
| step_PtrAdd_in2_ptr:
```

```
    ∀ s1 s2 ___ ____ _____,
      step_panic ExpectedInteger s1 s2 →
      step_ptradd (OpPtrAdd (Some (PtrValue ___)) (PtrValue ____) _____) s1 s2
```

3. Second argument is larger than `MAX_OFFSET_FOR_ADD_SUB` (after accounting for `swap`).

```
  | step_PtrAdd_diff_too_large:
    ∀ s1 s2 (arg_delta:word) (mem_delta: mem_address) ___ ____,

      arg_delta ≥ MAX_OFFSET_FOR_ADD_SUB = true →
      step_panic FatPointerDeltaTooLarge s1 s2 →
      step_ptradd (OpPtrAdd (Some (PtrValue ___)) (IntValue arg_delta) ____) s1
s2
```

4. Addition overflows.

```
  | step_PtrAdd_overflow:
    ∀ s1 s2 ofs new_ofs pid (arg_delta:word) (mem_delta: mem_address) span
high128,
      arg_delta < MAX_OFFSET_FOR_ADD_SUB = true →
      mem_delta = low mem_address_bits arg_delta →
      (true, new_ofs) = ofs + mem_delta →

      step_panic FatPointerOverflow s1 s2 →
      step_ptradd (OpPtrAdd
             (Some (PtrValue (high128, NotNullPtr (mk_fat_ptr pid (mk_ptr span
ofs)))))
             (IntValue arg_delta)
             (Some (PtrValue (high128, NotNullPtr (mk_fat_ptr pid (mk_ptr span
new_ofs))))))
        s1 s2.
End PtrAddDefinition.
```

Library EraVM.sem.PtrSub

```
Require SemanticCommon.

Import
  Arith
Core
TransientMemory
MemoryBase
Pointer
PrimitiveValue
SemanticCommon
```

```
State
isa.CoreSet
.
Import spec.

Section PtrSubDefinition.
  Open Scope ZMod_scope.
```

PtrSub

Abstract Syntax

```
OpPtrSub (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
```

Syntax

- `ptr.sub in1, in2, out`
- `ptr.sub.s in1, in2, out`

Summary

Takes a fat pointer from `in1` and a 32-bit unsigned number from `in2`. Advances the fat pointer's offset by that number, and writes (`in2`{128…255} ll incremented pointer) to `out`.

Semantic

1. Fetch input operands, swap them if `swap` modifier is set. Now operands are op_1 and op_2.
2. Ensure the op_1 is tagged as a pointer, and op_2 is not tagged as a pointer. Otherwise panic.
3. Decode fat pointer ptr_{in} from op_1
4. Let $diff$ be op_2 truncated to 32 bits:

   $$diff := op_2 \mod 2^{32}$$

   It is required that $op_2 < \texttt{MAX\_OFFSET\_FOR\_ADD\_SUB}$, otherwise VM panics.
5. Advance pointer offset of ptr_{in} by $diff$:

   $$ptr_{out} := ptr_{in}|_{offset := offset - diff}$$
6. Store the result, tagged as a pointer, to `out`. The most significant 128 bits of result are taken

from `op1`, the least significant bits hold an encoded pointer:

$$result := op_1\{255..128\} \#\# \textbf{encode}(ptr_{out})$$

```
  Inductive step_ptrsub : instruction → smallstep :=
  | step_PtrSub:
    ∀ high128 s ofs new_ofs pid (arg_delta:word) (mem_delta: mem_address) span,

      arg_delta ≤ MAX_OFFSET_FOR_ADD_SUB = true→
      mem_delta = low mem_address_bits arg_delta →
      (false,new_ofs) = ofs - mem_delta →

      step_ptrsub (OpPtrSub
              (Some (PtrValue (high128, NotNullPtr (mk_fat_ptr pid (mk_ptr span
ofs))))))
              (IntValue arg_delta)
              (Some (PtrValue (high128, NotNullPtr (mk_fat_ptr pid (mk_ptr span
new_ofs))))))
        s s
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- Flags are unaffected

Usage

- Manipulating fat pointers to pass slices of memory between functions.

Similar instructions

- Takes part in a group of pointer manipulating instructions:
  - `OpPtrAdd`
  - `OpPtrSub`
  - `OpPtrShrink`
  - `OpPtrPack`
- Instruction `OpPtrSub` effectively performs the same actions but the offset is added, not subtracted.

Encoding

Instructions `OpPtrSub`, `OpPtrSub`, `OpPtrPack` and `OpPtrShrink` are sharing an opcode.

Panics

1. First argument is not a pointer (after accounting for `swap`).

```
| step_PtrSub_in1_not_ptr:
  ∀ s1 s2 ___ ____ _____,
    step_panic ExpectedFatPointer s1 s2 →
    step_ptrsub (OpPtrSub (Some (IntValue ___)) ____ _____) s1 s2
```

2. Second argument is a pointer (after accounting for `swap`).

```
| step_PtrSub_in2_ptr:
  ∀ s1 s2 __ ____ _____,
    step_panic ExpectedFatPointer s1 s2 →
    step_ptrsub (OpPtrSub __ (PtrValue ____) _____) s1 s2
```

3. Second argument is larger than `MAX_OFFSET_FOR_SUB_SUB` (after accounting for `swap`).

```
  | step_PtrSub_diff_too_large:
    ∀ s1 s2 (arg_delta:word) (mem_delta: mem_address) ___ ____,

      arg_delta ≥ MAX_OFFSET_FOR_ADD_SUB = true →
      step_panic FatPointerDeltaTooLarge s1 s2 →
      step_ptrsub (OpPtrSub (Some (PtrValue ___)) (IntValue arg_delta) ____) s1
s2
```

4. Subtraction underflows.

```
  | step_PtrSub_underflow:
    ∀ high128 s1 s2 ofs new_ofs pid (arg_delta:word) (mem_delta: mem_address)
span,

      arg_delta < MAX_OFFSET_FOR_ADD_SUB = true →
      mem_delta = low mem_address_bits arg_delta →
      (true, new_ofs) = ofs - mem_delta →
```

```
      step_panic FatPointerOverflow s1 s2 →
      step_ptrsub (OpPtrSub
                     (Some (PtrValue (high128, NotNullPtr (mk_fat_ptr pid
(mk_ptr span ofs))))))
                     (IntValue arg_delta)
                     (Some (PtrValue (high128, NotNullPtr (mk_fat_ptr pid
(mk_ptr span new_ofs)))))))

        s1 s2
  .
End PtrSubDefinition.
```

Library EraVM.sem.PtrShrink

```
Require SemanticCommon.

Import Common Core TransientMemory isa.CoreSet State
  Pointer PrimitiveValue SemanticCommon ZArith.

Section PtrShrinkDefinition.
  Open Scope ZMod_scope.

  Inductive step_ptrshrink: instruction → smallstep :=
```

PtrShrink

Attention: **shrinking** and **narrowing** far pointers are different. See `fat_ptr_shrink` and `fat_ptr_narrow`.

Abstract Syntax

`OpPtrShrink (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)`

Syntax

- `ptr.shrink in1, in2, ou1`
- `ptr.shrink.s in1, in2, ou1`

Summary

Shrink the fat pointer, decreasing its length.

Semantic

1. Fetch input operands, swap them if `swap` modifier is set. Now operands are op_1 and op_2.
2. Ensure the op_1 is tagged as a pointer, and op_2 is not tagged as a pointer. Otherwise panic.
3. Decode fat pointer ptr_{in} from op_1
4. Let $diff$ be op_2 truncated to 32 bits:

$$diff := op_2 \mod 2^{32}$$

5. Rewind pointer length of ptr_{in} by $diff$:

$$ptr_{out} := ptr_{in}|_{length:=length-diff}$$

6. Store the result, tagged as a pointer, to `out`:

$$result := op_1\{255..128\} \# \# \mathbf{encode}(ptr_{out})$$

```
| step_PtrShrink :
  ∀ s high128 ptr_in ptr_out delta ,
  let diff := low mem_address_bits delta in
  fat_ptr_shrink diff ptr_in ptr_out →


  step_ptrshrink (OpPtrShrink
                    (Some (PtrValue (high128, NotNullPtr ptr_in)))
                    (IntValue delta)
                    (Some (PtrValue (high128, NotNullPtr ptr_out))))
    s s
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- Flags are unaffected

Usage

- Manipulating fat pointers to pass slices of memory between functions.

Similar instructions

- Takes part in a group of pointer manipulating instructions:
  - `OpPtrAdd`
  - `OpPtrSub`
  - `OpPtrShrink`
  - `OpPtrPack`

Encoding

Instructions `OpPtrAdd`, `OpPtrSub`, `OpPtrPack` and `OpPtrShrink` are sharing an opcode.

Panics

1. First argument is not a pointer (after accounting for `swap`).

```
| step_PtrShrink_in1_not_ptr:
  ∀ s1 s2 ___2 ___3 ___4,
    step_panic ExpectedFatPointer s1 s2 →
    step_ptrshrink (OpPtrShrink (Some (IntValue ___2)) ___3 ___4) s1 s2
```

2. Second argument is a pointer (after accounting for `swap`).

```
| step_PtrShrink_in2_ptr:
  ∀ s1 s2 ___1 ___3 ___4,
    step_panic ExpectedFatPointer s1 s2 →
    step_ptrshrink (OpPtrShrink (Some ___1) (PtrValue ___3) ___4) s1 s2
```

3. Shrinking underflows.

```
| step_PtrShrink_underflow:
  ∀ s1 s2 high128 ptr_in ptr_out delta ,
    let diff := low mem_address_bits delta in
    fat_ptr_shrink_OF diff ptr_in = None →

    step_ptrshrink (OpPtrShrink
                      (Some (PtrValue (high128, NotNullPtr ptr_in)))
```

```
                        (IntValue delta)
                        (Some (PtrValue (high128, NotNullPtr ptr_out))))
        s1 s2
  .
End PtrShrinkDefinition.
```

Library EraVM.sem.PtrPack

```
Require SemanticCommon.

Import Common Core TransientMemory isa.CoreSet State
  SemanticCommon PrimitiveValue ZArith FatPointerABI.

Section PtrPackDefinition.
  Open Scope ZMod_scope.
  Inductive step_ptrpack : instruction → smallstep :=
```

PtrPack

Abstract Syntax

```
OpPtrPack (in1: in_any) (in2: in_reg) (out: out_any) (swap:mod_swap)
```

Syntax

- `ptr.pack in1, in2, ou1`
- `ptr.pack.s in1, in2, ou1`

Summary

Concatenates the lower 128 bit of `in1` and the higher 128 bits of `in2`, writes result to `out`. `in1` should hold a `fat_ptr`.

See Usage.

Semantic

1. Fetch input operands, swap them if `swap` modifier is set. Now operands are op_1 and op_2.
2. Ensure the op_1 is tagged as a pointer, and op_2 is not tagged as a pointer. Otherwise panic.
3. Ensure that the lower 128 bits of op_2 are zero. Otherwise panic.
4. Store the result, tagged as a pointer, to `out`:

$$result := op_2\{255...128\} \#\# op_1\{128...0\}$$

```
  | step_PtrPack :
    ∀ op1_high128 ptr (op2:word) (s:state) encoded result,
      low 128 op2 = zero128 →
      Some encoded = encode_fat_ptr ptr →
      result = (@high 128 128 op2) ## encoded →
      step_ptrpack (@OpPtrPack bound (Some (PtrValue (op1_high128, ptr)))
(IntValue op2) (IntValue result)) s s
```

Affected parts of VM state

- execution stack: PC, as by any instruction; SP, if `in1` uses `RelPop` addressing mode, or if `out` uses `RelPush` addressing mode.
- Current stack memory page, if `out` resolves to it.
- GPRs, if `out` resolves to a register.
- Flags are unaffected

Usage

- fat pointer in in_1 spans across bits in_1 $0...127$, and the bits in_1 $128...255$ are therefore available to put other data. This is used by for memory forwarding to far calls when we need to forward an existing fat pointer.

  To pass a fat pointer `P` to far call, it is necessary to encode an instance of `FarCallABI` with `fwd_memory := ForwardFatPointer P` into a `PtrValue`.

```
Module ABI.FarCall.

Record params :=
  mk_params {
      fwd_memory: fwd_memory;
      ergs_passed: ergs;
      shard_id: shard_id;
      constructor_call: bool;
      to_system: bool;
    }.
```

  The compound type of `FarCallABI` is serialized to a `word` in such a way that the pointer takes up the lower 128 bits of memory. This matches the layout of any fat pointer: serialized pointers occupy the lower 128 bit of a word.

  Therefore, encoding an instance of `FarCallABI` can be done as follows:

- take an existing `PtrValue P`
- form a value `A` encoding `ergs_passed`, `shard_id` and other fields of `FarCallABI` in A{128…255}.
- invoke `OpPtrPack P A B`. Now `B` stores an encoded instance of `FarCallABI` and can be passed to one of far call instructions.

Similar instructions

- Takes part in a group of pointer manipulating instructions:
  - `OpPtrAdd`
  - `OpPtrSub`
  - `OpPtrShrink`
  - `OpPtrPack`

Encoding

Instructions `OpPtrAdd`, `OpPtrSub`, `OpPtrPack` and `OpPtrShrink` are sharing an opcode.

Panics

1. First argument is not a pointer (after accounting for `swap`).

```
| step_PtrPack_in1_not_ptr:
  ∀ s1 s2 ___1 ___2 ___3,
    step_panic ExpectedFatPointer s1 s2 →
    step_ptrpack (OpPtrPack (Some (IntValue ___1)) ___2 ___3) s1 s2
```

2. Second argument is a pointer (after accounting for `swap`).

```
| step_PtrPack_in2_ptr:
  ∀ s1 s2 ___1 ___2 ___3,
    step_panic ExpectedFatPointer s1 s2 →
    step_ptrpack (OpPtrPack (Some ___1) (PtrValue ___2) ___3) s1 s2
```

3. Low 128 bits of the second operand are not zero (after accounting for `swap`).

```
| step_PtrPack_notzero:
  ∀ s1 s2 op2 ___1 ___2,

    low 128 op2 ≠ zero128 →
    step_panic PtrPackExpectsOp2Low128BitsZero s1 s2 →
    step_ptrpack (@OpPtrPack bound ___1 (IntValue op2) ___2) s1 s2
.
```

```
End PtrPackDefinition.
```

Library EraVM.sem.SStore

```
From RecordUpdate Require Import RecordSet.
Require SemanticCommon.

Import Common Core Predication Ergs isa.CoreSet CallStack Event memory.Depot
TransientMemory MemoryOps State
  PrimitiveValue SemanticCommon ZArith RecordSetNotations.

Section SStoreDefinition.

  Definition sstore_cost cs : ergs :=
    let pubdata := (net_pubdata cs) in
    ergs_of (pubdata × Z.of_nat bytes_in_word).

  Inductive step_sstore: instruction → smallstep :=
```

SStore

Abstract Syntax

`OpSStore (in1: in_reg) (in2: in_reg)`

Syntax

- `log.swrite in1, in2` aliased as `sstore in1, in2`

Summary

Store word in current storage by key.

Semantic

- Store word in current shard, and current contract's storage by key `key`.

  Current contract is identified by the field `ecf_this_address` of the active external frame.
- Pay for storage write.

```
| step_SStore:
```

```
∀ cs new_cs key new_depot write_value gs new_gs ts1 ts2 __ ,

  (* there are currently no refunds *)
  cs = gs_callstack ts1 →
  let fqa_storage := mk_fqa_key (current_storage_fqa cs) key in
  let old_depot := gs.(gs_revertable).(gs_depot) in
  storage_write old_depot fqa_storage write_value new_depot →
  global_state_new_depot new_depot gs new_gs →

  ts2 = ts1 <| gs_callstack := new_cs |> →
  pay (sstore_cost cs) cs new_cs →

  step_sstore (OpSLoad (mk_pv __ key) (IntValue write_value))
              {|
                gs_transient := ts1;
                gs_global := gs;
              |}
              {|
                gs_transient := ts2;
                gs_global := new_gs;
              |}
```

Affected parts of VM state

- execution stack:
  - PC, as by any instruction;
  - allocated ergs
- GPRs, because `res` only resolves to a register.
- Depot of current shard.

Usage

- Only `SStore` is capable to write data to storage.
- `SStore` is rolled back if the current frame ended by `OpPanic` or `OpRevert`.

Similar instructions

- `OpSLoad`, `OpSStore`, `OpEvent`, `OpToL1Message`, `OpPrecompileCall` share the same opcode.

Panics

1. Not enough ergs to pay for storage write.

```
| step_SStore_unaffordable:
  ∀ cs gs ts1 ts2 ___1 ___2,

    (* there are currently no refunds *)
    cs = gs_callstack ts1 →
```

```
      affordable cs (sstore_cost cs) = false →
      ts2 = ts1 <| gs_status := Panic StorageWriteUnaffordable |> →
      step_sstore (OpSLoad ___1 ___2)
                  {|
                    gs_transient := ts1;
                    gs_global := gs;
                  |}
                  {|
                    gs_transient := ts2;
                    gs_global := gs;
                  |}
.
End SStoreDefinition.
```

Library EraVM.sem.SLoad

```
Require SemanticCommon.
Import isa.CoreSet TransientMemory memory.Depot PrimitiveValue SemanticCommon
State.

Section SLoadDefinition.
  Generalizable Variable __.

  Inductive step_sload: instruction → smallstep :=
```

SLoad

Abstract Syntax

`OpSLoad (key: in_reg) (dest: out_reg)`

Syntax

- `log.sread in1, out` aliased as `sload in1, out`

Summary

Access word in current storage by key.

Semantic

1. Load word from current shard, current contract's storage by key `key`.

Current contract is identified by the field `ecf_this_address` of the active external frame.

2. Store the value to `dest`.

```
  | step_SLoad:
    ∀ read_value key (s:state) __,
      let fqa_storage := mk_fqa_key (current_storage_fqa (gs_callstack s)) key
in

      storage_read (gs_revertable s).(gs_depot) fqa_storage read_value →
      step_sload (OpSLoad (mk_pv __ key)(IntValue read_value)) s s
  .
```

Affected parts of VM state

- execution stack: PC, as by any instruction;
- GPRs, because `res` only resolves to a register.

Usage

- Only `SLoad` is capable of reading data from storage.

Similar instructions

- `OpSLoad`, `OpSStore`, `OpEvent`, `OpToL1Message`, `OpPrecompileCall` share the same opcode.

```
End SLoadDefinition.
```

Library EraVM.sem.OpEvent

```
Require SemanticCommon.

Import CallStack Event isa.CoreSet State PrimitiveValue SemanticCommon.
Import ssreflect ssrfun ssrbool ssreflect.eqtype ssreflect.tuple.

Section EventDefinition.
  Inductive step_event: instruction → smallstep :=
```

Event

Abstract Syntax

```
OpEvent (in1: in_reg) (in2: in_reg) (is_first: bool)
```

Syntax

- `log.event in1, in2` aliased as `event in1, in2`
- `log.event.first in1, in2` aliased as `event.i in1, in2`

Summary

Emit an event with provided key and value. See `event` for more details on events system.

Semantic

1. Fetch key and value from `in1` and `in2`.
2. If `is_first` is `true`, mark the event as the first in a chain of events.
3. Emit event.

```
| step_Event:
  ∀ xs is_first _tag1 _tag2
    key value gs new_gs,
    let regs := gs_regs xs in
    let pages := gs_pages xs in
    let xstack := gs_callstack xs in

    emit_event (EventQuery {|
                    ev_shard_id := current_shard xstack;
                    ev_is_first := is_first;
                    ev_tx_number_in_block := gs_tx_number_in_block gs;
                    ev_address := current_contract xstack;
                    ev_key := key;
                    ev_value := value;
                  |}) gs new_gs →

    step_event (OpEvent (mk_pv _tag1 key) (mk_pv _tag2 value) is_first)
               {|
                 gs_global := gs;
                 gs_transient := xs;
               |}
               {|
                 gs_global := new_gs;
                 gs_transient := xs;
```

```
|}.
```

Affected parts of VM state

- Event queue.

Usage TODO

Similar instructions

- `OpSLoad`, `OpSStore`, `OpEvent`, `OpToL1Message`, `OpPrecompileCall` are variants of the same `mach_instruction`.

```
End EventDefinition.
```

Library EraVM.sem.ToL1

```
From RecordUpdate Require Import RecordSet.
Require SemanticCommon.

Import Arith Common Ergs CallStack Event TransientMemory isa.CoreSet State
  PrimitiveValue SemanticCommon RecordSetNotations.
Import ssreflect.tuple ssreflect.eqtype.

Section ToL1Definition.

  Open Scope ZMod_scope.
```

ToL1Message

Abstract Syntax

```
OpToL1Message (in1: in_reg) (in2: in_reg) (is_first: bool)
```

Syntax

- `log.to_l1 in1, in2` aliased as `event in1, in2`
- `log.to_l1 event.first in1, in2` aliased as `event.i in1, in2`

Summary

Emit a message to L1 with provided key and value. See `event` for more details on events system.

Semantic

1. Fetch key and value from `key` and `value`.
2. If `is_first` is `true`, mark the event as the first in a chain of events.
3. Emit L1 message event.

```
  Inductive step_to_l1: instruction → smallstep :=

  | step_ToL1:
    ∀ cs new_cs is_first key value gs new_gs cost cost_truncated ts1 ts2 __
___1,

      cost = gs_current_ergs_per_pubdata_byte gs × ergs_of
L1_MESSAGE_PUBDATA_BYTES →
      cost < (fromZ (unsigned_max ergs_bits)) →
      cost_truncated = low ergs_bits cost →
      pay cost_truncated cs new_cs →
      emit_l1_msg {|
          ev_shard_id := current_shard cs;
          ev_is_first := is_first;
          ev_tx_number_in_block := gs_tx_number_in_block gs;
          ev_address := current_contract cs;
          ev_key := key;
          ev_value := value;
        |} gs new_gs →
      ts2 = ts1 <| gs_callstack := new_cs |> →
      step_to_l1 (OpToL1Message (mk_pv __ key) (mk_pv ___1 value) is_first)
           {|
             gs_transient := ts1;
             gs_global := gs;
           |}
           {|
             gs_transient := ts2;
             gs_global := new_gs;
           |}
.
```

Affected parts of VM state

- Event queue.

Usage

Communicating with L1.

Similar instructions

- `OpSLoad`, `OpSStore`, `OpEvent`, `OpToL1Message`, `OpPrecompileCall` share the same opcode.

```
End ToL1Definition.
```

Library EraVM.sem.PrecompileCall

```
From RecordUpdate Require Import RecordSet.

Require SemanticCommon Precompiles.

Import Addressing ABI Bool Common Coder Predication Ergs CallStack Event
TransientMemory MemoryOps isa.CoreSet State
  Addressing.Coercions PrimitiveValue SemanticCommon RecordSetNotations
PrecompileParametersABI.
```

```
Section PrecompileCallDefinition.
  Open Scope ZMod_scope.
  Import ssreflect.tuple ssreflect.eqtype.
```

PrecompileCall

Abstract Syntax

```
OpPrecompileCall (in1: in_reg) (in2: in_reg) (dst:out_reg)
```

Syntax

- `log.precompile in1, in2, out1`

Summary

A precompile call is a call to an extension of a virtual machine. The extension operates differently

depending on the currently executing contract's address. Only system contracts may have precompiles.

Semantic

Attempt to pay the extra cost:

$$cost_{extra} = in_2 \mod 2^{32}$$

- If the cost is affordable:
  - pay $cost_{extra}$
  - execute the precompile logic associated to the currently executing contract. This will emit a special event.
  - set out_1 to one.

```
  Inductive step_precompile: instruction → smallstep :=
  | step_PrecompileCall_affordable:
    ∀ flags pages cs regs result new_cs extra_ergs gs new_gs ctx ___1 new_xs
params,
      let heap_id := active_heap_id cs in


      let cost := low ergs_bits extra_ergs in
      pay cost cs new_cs →
      result = one256 →

      let xs := {|
                    gs_callstack := new_cs;

                    gs_regs := regs;
                    gs_pages := pages;
                    gs_flags := flags;
                    gs_context_u128 := ctx;
                    gs_status := NoPanic;
                  |} in
      Precompiles.precompile_processor (current_contract cs) params xs new_xs →

      emit_event (PrecompileQuery {|
                     q_contract_address := current_contract cs;
                     q_tx_number_in_block := gs_tx_number_in_block gs;
                     q_shard_id := current_shard cs;
                     q_key := params;
                   |}) gs new_gs →
      step_precompile (OpPrecompileCall (Some (IntValue params)) (mk_pv ___1
extra_ergs) (IntValue result))
                      {|
                        gs_transient := {|
                                        gs_regs := regs;
                                        gs_pages := pages;
                                        gs_callstack := cs;
                                        gs_flags := flags;
                                        gs_context_u128 := ctx;
```

```
                    gs_status := NoPanic;
                  |};
    gs_global := gs;

  |}
  {|
    gs_transient := new_xs;
    gs_global := new_gs;
  |}
```

- If the cost is unaffordable, do not pay anything beyound `base_cost` of this instruciton. Set out_1 to zero.

```
| step_PrecompileCall_unaffordable:
  ∀ flags pages cs regs extra_ergs ___1 ___2 s1 s2 result ctx,
    let cost := low ergs_bits extra_ergs in
    affordable cs cost = false →

    result = zero256 →
    step_transient_only
      {|
        gs_callstack := cs;

        gs_regs := regs;
        gs_flags := flags;
        gs_pages := pages;
        gs_context_u128 := ctx;
        gs_status := NoPanic;
      |}
      {|
        gs_callstack := ergs_reset cs;

        gs_regs := regs;
        gs_flags := flags;
        gs_pages := pages;
        gs_context_u128 := ctx;
        gs_status := NoPanic;
      |} s1 s2 →
      step_precompile (OpPrecompileCall ___2 (mk_pv ___1 extra_ergs) (IntValue
result)) s1 s2
.

End PrecompileCallDefinition.
```

Library EraVM.sem.Context

```
From RecordUpdate Require Import RecordSet.

Require SemanticCommon.

Import Addressing ABI Bool Coder Core Common Predication Ergs CallStack
```

```
TransientMemory MemoryOps isa.CoreSet State
  Addressing.Coercions PrimitiveValue SemanticCommon RecordSetNotations
ABI.MetaParametersABI.

Section ContextDefinitions.
```

ContextThis

Abstract Syntax

`OpContextThis (out: out_reg)`

Syntax

`context.this out`

Summary

Retrieves the address of the currently executed contract.

Semantic

- Fetch the address of the currently executed contract from the active external frame.
- Widen the address to `word_bits`, zero-extended, and write to register `out`.

Affected parts of VM state

- registers : `out` register is modified.

Usage

On `OpDelegateCall` this address is preserved to be one of the caller. See `select_this_address`.

Similar instructions

See `OpContextCaller`, `OpContextCodeAddress`.

Encoding

- A variant of `context` `mach_instruction`.

```
Inductive step_context: instruction → smallstep :=

| step_ContextThis:
  ∀ this_addr (this_addr_word:word) (s1 s2: state),

    this_addr = ecf_this_address (active_extframe (gs_callstack s1)) →
    this_addr_word = widen word_bits this_addr →


    step_context (OpContextThis (IntValue this_addr_word)) s1 s2
```

ContextCaller

Abstract Syntax

```
OpContextCaller (out: out_reg)
```

Syntax

```
context.caller out
```

Summary

Retrieves the address of the contract which has called the currently executed contract.

Semantic

- Fetch the address of the currently executed contract from the active external frame.
- Widen address is widened to `word_bits`, zero-extended, and written to register `out`.

Affected parts of VM state

- registers : `out` register is modified.

Usage

On `OpDelegateCall` this address is preserved to be the caller of the caller. See `select_sender`.

Similar instructions

See `OpContextThis`, `OpContextCodeAddress`.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextCaller:
  ∀ sender_addr sender_addr_word (s1 s2:state),
    sender_addr = (active_extframe (gs_callstack s1)).(ecf_msg_sender) →
    sender_addr_word = widen word_bits sender_addr →

    step_context (OpContextCaller (IntValue sender_addr_word)) s1 s2
```

ContextCodeAddress

Abstract Syntax

`OpContextCodeAddress (out: out_reg)`

Syntax

```
context.code_source out
```

Summary

Retrieves the address of the contract code that is actually being executed.

Semantic

- Fetch the contract address of the currently executed code from the active external frame.
- Widen the address to `word_bits`, zero-extended, and write to register `out`.

Affected parts of VM state

- registers : `out` register is modified.

Usage

- In the execution frame created by `OpDelegateCall` this will be the address of the contract that was called by `OpDelegateCall`.
- Necessary to implement Solidity's `immutable` under `OpDelegateCall`.

Similar instructions

See `OpContextThis`, `OpContextCaller`.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextCodeAddress:
  ∀ code_addr code_addr_word (s1 s2:state),
    code_addr = ecf_code_address (active_extframe (gs_callstack s1)) →
    code_addr_word = widen word_bits code_addr →
    step_context (OpContextCodeAddress (IntValue code_addr_word)) s1 s2
```

ContextErgsLeft

Abstract Syntax

`OpContextErgsLeft (out: out_reg)`

Syntax

```
context.ergs_left out
```

Summary

Retrieves the number of ergs allocated for the current frame.

Semantic

- Fetch the currently allocated ergs from the topmost frame, external or internal. The `ergs` belonging to the parent frames are not counted.
- Widen the ergs number to `word_bits`, zero-extended, and write to `out`.

Affected parts of VM state

- registers : `out` register is modified.

Usage

- Check if the number of ergs is sufficient for an expensive task.
- Should be used before `OpPrecompileCall`.

Similar instructions

The `context` instruction family.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextErgsLeft:
  ∀ ergs_left_word (s1 s2:state),
    ergs_left_word = widen word_bits (ergs_remaining (gs_callstack s1)) →
    step_context (OpContextErgsLeft (IntValue ergs_left_word)) s1 s2
```

ContextSP

Abstract Syntax

```
OpContextSp (out: out_reg)
```

Syntax

```
context.sp out
```

Summary

Retrieves current stack pointer.

Semantic

- Fetch the current SP from the topmost frame, external or internal.
- Widen the SP value to `word_bits`, zero-extended, and write to `out`.

Affected parts of VM state

- registers : `out` register is modified.

Usage

- Check if the number of ergs is sufficient for an expensive task.
- Should be used before `OpPrecompileCall`.

Similar instructions

The `context` instruction family.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextSP:
  ∀ sp_zero_padded (s1 s2 :state),
```

```
    sp_zero_padded = widen word_bits (sp_get (gs_callstack s1)) →
    step_context (OpContextSp (IntValue sp_zero_padded)) s1 s2
```

ContextGetContextU128

Abstract Syntax

`OpContextGetContextU128 (out: out_reg)`

Syntax

```
context.get_context_u128 out
```

Summary

Retrieves current captured context value from the active external frame.

Does not interact with the context register.

Semantic

- Fetch the current context value from the active external frame.
- Widen the context value from 128 bits to `word_bits`, zero-extended, and write to `out`.

Affected parts of VM state

- registers : `out` register is modified.

Usage

- See `gs_context_u128`, `ecf_context_u128_value`.

Similar instructions

- The `context` instruction family.
- Farcalls capture context. See `OpFarCall`, `OpMimicCall`, `OpDelegateCall`.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextGetContextU128:
  ∀ wcontext (s1 s2: state),
    wcontext = widen word_bits (gs_context_u128 s1) →
    step_context (OpContextGetContextU128 (IntValue wcontext)) s1 s2
```

ContextSetContextU128

- Only in kernel mode.
- Forbidden in static calls.

Abstract Syntax

```
OpContextSetContextU128 (in: in_reg)
```

Syntax

```
context.set_context_u128 out
```

Summary

Sets context register.

Does not interact with the captured context value in the active external frame.

Semantic

- Fetch the value from `out` and narrow it to 128 bits.
- Store the shrunk value in the context register `gs_context_u128`.

Affected parts of VM state

- registers : `out` register is modified.

Usage

- See `gs_context_u128`, `ecf_context_u128_value`.

Similar instructions

- The `context` instruction family.
- Farcalls capture context. See `OpFarCall`, `OpMimicCall`, `OpDelegateCall`.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextSetContextU128:
  ∀ (new_context256 :word) any_tag (new_context_u128:u128) xs1 xs2 s1 s2,
    new_context_u128 = low 128 new_context256→
    xs2 = xs1 <| gs_context_u128 := new_context_u128 |> →
    step_transient_only xs1 xs2 s1 s2 →

    step_context (OpContextSetContextU128 (mk_pv any_tag new_context256)) s1 s2
```

ContextMeta

VM internal state introspection.

Abstract Syntax

`OpContextMeta (out: out_reg)`

Syntax

```
context.meta out
```

Summary

Fetches

Semantic

- Stores the encoded value of `MetaParametersABI` in `out`. They follow the structure:

```
Record params := {
      ergs_per_pubdata_byte: ergs;
      heap_size: mem_address;
      aux_heap_size: mem_address;
      this_shard_id: shard_id;
      caller_shard_id: shard_id;
      code_shard_id: shard_id;
    }.
```

Affected parts of VM state

- registers : `out` register is modified.

Similar instructions

- The `context` instruction family.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextMeta:
  ∀ params cs shards
    (s1 s2: state),
    cs = gs_callstack s1 →
    shards = (active_extframe cs).(ecf_shards) →
    params = {|
               ergs_per_pubdata_byte := gs_current_ergs_per_pubdata_byte s1;
               heap_size := heap_bound cs;
               aux_heap_size := auxheap_bound cs;
               this_shard_id := shard_this shards;
               caller_shard_id := shard_caller shards;
               code_shard_id := shard_code shards;
             |} →
    step_context (OpContextMeta (Some (IntValue params))) s1 s2
```

ContextIncrementTxNumber

- Kernel only.
- Forbidden in static context.

Abstract Syntax

`OpContextIncrementTxNumber`

Syntax

```
context.inc_tx_num out
```

Summary

Increments the tx number counter in `gs_tx_number_in_block`.

Semantic

Affected parts of VM state

- only tx counter.

Usage

Utility in system contracts.

Similar instructions

- The `context` instruction family.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextIncTx:
  ∀ transient gs new_gs,
```

```
global_state_increment_tx tx_inc gs new_gs →
step_context OpContextIncrementTxNumber
     {|
       gs_transient := transient;
       gs_global := gs;
     |}
     {|
       gs_transient := transient;
       gs_global := new_gs;
     |}
```

SetErgsPerPubdataByte

- Kernel only.
- Forbidden in static context.

Abstract Syntax

`OpContextSetErgsPerPubdataByte (value:in_reg)`

Syntax

```
context.set_ergs_per_pubdata in
```

Summary

Sets a new value to `gs_current_ergs_per_pubdata_byte`.

Semantic

Affected parts of VM state

- only `gs_current_ergs_per_pubdata_byte`.

Usage

Utility in system contracts.

Similar instructions

- The `context` instruction family.

Encoding

- A variant of `context` `mach_instruction`.

```
| step_ContextSetErgsPerPubdata:
  ∀ gs new_gs any_tag new_val transient ,

    let new_ergs := low ergs_bits new_val in
    new_gs = gs <| gs_global ::= (fun s ⇒ s <| gs_current_ergs_per_pubdata_byte
:= new_ergs |> ) |> →

    step_context (OpContextSetErgsPerPubdataByte (mk_pv any_tag new_val))
        {|
          gs_transient := transient;
          gs_global := gs;
        |}
        {|
          gs_transient := transient;
          gs_global := new_gs;
        |}

.
End ContextDefinitions.
```

Library EraVM.Coder

```
Require Common.
Import ssreflect ssrfun.

Section Encoding.
```

Encoding

Application Binary Interfaces (`ABI`) require describing serialization and deserialization.

- Serialization encodes an instance of type `T` into a word of type `word`.
- Deserialization tries to decode an instance of type `T` from a word of type `word`.

```
  Context {U T:Type}.
```

The type `decoder` defines an embedding of a subset of words of type `word` to a type `T`. Decoding may fail if the input word is malformed.

```
  Definition decoder := U → option T.
```

Definition `encoder` defines an embedding of type `T` to a set of possible `word` values.

```
  Definition encoder := T → option U.

  Definition revertible (decode:decoder) (encode:encoder):= ∀ obj encoded,
encode obj = Some encoded → decode encoded = Some obj.
```

The record `coder` connects a specific decoder with the matching encoder, and proofs of their properties.

- `revertible decode encode` formalizes the following: if we encode an element `t` of type

```
  Record coder := mk_coder {
      decode: decoder;
      encode: encoder;
```

`decode` and `encode` should be mutual inverses in the following sense:

```
      _: revertible decode encode;
    }.
End Encoding.

Section Properties.

  Context {A B C: Type}.

  Section PropertyComposition.
    Context (encode1: @encoder B A) (decode1:@decoder B A) (encode2: @encoder C
B) (decode2: @decoder C B).

    Theorem revertible1_compose:
      revertible decode1 encode1 →
      revertible decode2 encode2 →
      let encode3 : A → option C := pcomp encode2 encode1 in
      let decode3 : C → option A := pcomp decode1 decode2 in
      revertible decode3 encode3.
```

```
  End PropertyComposition.

  Definition coder_compose (c2: @coder C B) (c1: @coder B A) : @coder C A.
  Defined.
End Properties.
```

Library EraVM.ABI

```
Require Coder Ergs Pointer GPR MemoryManagement TransientMemory lib.BitsExt.

Import ssreflect ssreflect.ssrfun ssreflect.ssrbool ssreflect.eqtype
ssreflect.tuple zmodp.
Import Core Common Coder Bool GPR Ergs MemoryManagement TransientMemory
Pointer.
```

Application binary interface (ABI)

This section details the serialization and deserialization formats for compound instruction arguments.

The description from Rust VM implementation is described here: https://github.com/matter-labs/zkevm_opcode_defs/blob/v1.4.1/src/definitions/abi

```
Require Export
  ABI.FatPointerABI
  ABI.MetaParametersABI
  ABI.PrecompileParametersABI
  ABI.NearCallABI
  ABI.FarRetABI
  ABI.FarCallABI
.
```

Library EraVM.ABI.NearCallABI

```
Require Coder Ergs TransientMemory.

Import ssreflect.
Import Types Core Coder Ergs TransientMemory.

Section NearCallABI.
```

Near call may only accept one parameter: the amount of ergs allocated to it.

```
Record params: Type :=
```

```
  mk_params {
      ergs_passed: u32; (* in low 32 bits *)
    }.

Definition encode : params → option u32 := fun p ⇒ Some p.(ergs_passed).
Definition decode : u32 → option params := fun u ⇒ Some (mk_params u).

Definition encode_word (p:params) (high224: u224) : option word :=
  option_map (fun encoded ⇒ high224 ## encoded) (encode p).

Definition decode_word (w:word) : option (u224 × params) :=
  option_map (fun decoded ⇒ (@high 224 32 w, decoded)) (decode (low 32 w)).

Definition coder : @coder u32 params.
Defined.
End NearCallABI.
```

Library EraVM.ABI.FatPointerABI

```
Require Coder Pointer lib.BitsExt.

Import ssreflect ssreflect.ssrfun ssreflect.eqtype ssreflect.tuple.
Import Core Common Coder Pointer lib.BitsExt.

Section FatPointerABI.
```

Record `fat_ptr_layout` displays the memory layout of a 128-bit fat pointer.

```
Record fat_ptr_layout := mk_fat_ptr_layout {
                           length: u32;
                           start: u32;
                           page: u32;
                           offset: u32;
                         }.
Definition null_fat_ptr_layout := mk_fat_ptr_layout zero32 zero32 zero32
zero32.
Section LayoutCoder.
```

Functions `encode_layout` and `decode_layout` formalize the encoding of a `fat_ptr_nullable` to `fat_ptr_layout`.

```
  Definition encode_layout : @encoder fat_ptr_layout fat_ptr_nullable :=
    fun fp : fat_ptr_nullable ⇒
      match fp with
      | NullPtr ⇒ Some null_fat_ptr_layout
      | NotNullPtr (mk_fat_ptr p (mk_ptr (mk_span s l) ofs)) ⇒
          if (p == 0) || (p ≥ 2^32) then None else
            Some (mk_fat_ptr_layout l s (# p) ofs)
      end.
```

```
  Definition decode_layout : @decoder fat_ptr_layout fat_ptr_nullable :=
    fun w: fat_ptr_layout ⇒
      match w with
      | mk_fat_ptr_layout length start page offset ⇒
          if page == zero32 then Some NullPtr else
            Some (NotNullPtr {|
                      fp_page := toNat page;
                      fp_ptr := {|
                                  p_span := mk_span start length;
                                  p_offset := offset;
                                |}
                    |})
      end.

  Theorem layout_coding_revertible: revertible decode_layout encode_layout.

  Definition fat_ptr_layout_coder : @Coder.coder fat_ptr_layout
fat_ptr_nullable
    := mk_coder decode_layout encode_layout layout_coding_revertible.

End LayoutCoder.

Section BinaryCoder.
  #[local] Open Scope ZMod_scope.

  Definition encode_bin: @encoder u128 fat_ptr_layout :=
    fun fp: fat_ptr_layout ⇒
      match fp with
      | mk_fat_ptr_layout length start page offset ⇒
          Some (length ## start ## page ## offset)
      end.

  Definition decode_bin : @decoder u128 fat_ptr_layout :=
    fun w: u128 ⇒
      let length := w { 3×32, 4×32 } in
      let start := w { 2×32, 3×32 } in
      let page := w { 32, 2×32} in
      let offset := w { 0, 32 } in
      Some (mk_fat_ptr_layout length start page offset).

  Lemma fat_ptr_bin_revertible: revertible decode_bin encode_bin.

  Definition binary_coder: @Coder.coder u128 fat_ptr_layout :=
    mk_coder decode_bin encode_bin fat_ptr_bin_revertible.
End BinaryCoder.

Section ComposedCoder.
  Definition ABI : @Coder.coder u128 fat_ptr_nullable :=
    coder_compose binary_coder fat_ptr_layout_coder.

  Definition decode_fat_ptr (w:u128) : option fat_ptr_nullable := ABI.(decode)
w.

  Definition decode_fat_ptr_word (w:word) : option (u128 × fat_ptr_nullable) :=
    let (high128, low128) := split2 128 128 w in
    match decode_fat_ptr low128 with
    | Some fpn ⇒ Some (high128, fpn)
```

```
    | _ ⇒ None
    end
  .

  Definition decode_heap_ptr (w:word) : option (u224 × heap_ptr) :=
    let (msbs, ofs) := split2 _ 32 w in
    Some (msbs, mk_hptr ofs).

  Definition encode_heap_ptr (h:heap_ptr) : option u32 :=
    Some (hp_addr h)
  .

  Definition encode_heap_ptr_word (high224: u224) (h:heap_ptr) : option word :=
    match encode_heap_ptr h with
    | Some hpenc ⇒ Some (high224 ## hpenc)
    | _ ⇒ None
    end
  .
  Definition encode_fat_ptr (fp: fat_ptr_nullable) : option u128 :=
ABI.(encode) fp.
  Definition encode_fat_ptr_word (high_bytes: u128) (fp: fat_ptr_nullable) :
option word :=
    match encode_fat_ptr fp with
    | Some enc ⇒ Some (high_bytes ## enc)
    | _ ⇒ None
    end.
End ComposedCoder.
End FatPointerABI.
```

Library EraVM.ABI.ForwardPageTypesABI

```
Require Coder Ergs MemoryManagement Pointer TransientMemory lib.BitsExt
ABI.FatPointerABI.

Import ssreflect ssreflect.ssrfun ssreflect.eqtype ssreflect.tuple.
Import Arith Core Common Coder Ergs MemoryManagement Pointer TransientMemory
lib.BitsExt FatPointerABI.

Module FarCallForwardPageType.
  Definition UseHeap : u8 := # 0.
  Definition ForwardFatPointer : u8 := # 1.
  Definition UseAuxHeap : u8 := # 2.
  (* other u8 values are mapped to UseHeap. *)
End FarCallForwardPageType.

Definition data_page_type_to_u8 (t:data_page_type) : u8 :=
  match t with
  | Heap ⇒ FarCallForwardPageType.UseHeap
  | AuxHeap ⇒FarCallForwardPageType.UseAuxHeap
  end.

Definition span_of (fp: fat_ptr_layout) : option span :=
```

```
  if fp.(offset) == zero32
  then Some (mk_span fp.(start) fp.(length))
  else None
.

Definition fwd_memory_adapter (fwd_type: u8) (raw_fat_ptr_layout:
fat_ptr_layout) : option MemoryManagement.fwd_memory:=
  if fwd_type == FarCallForwardPageType.ForwardFatPointer then
    option_map ForwardExistingFatPointer
      (FatPointerABI.decode_layout raw_fat_ptr_layout)
  else
    match span_of raw_fat_ptr_layout with
    | Some span ⇒
        Some (
            if fwd_type == FarCallForwardPageType.UseAuxHeap then
              ForwardNewFatPointer AuxHeap span
            else (* Heap or a default option *)
              ForwardNewFatPointer Heap span
          )
    | None ⇒ None
    end
.
```

Library EraVM.ABI.FarCallABI

```
Require Coder Ergs memory.Depot MemoryManagement Pointer lib.BitsExt
ABI.FatPointerABI ABI.ForwardPageTypesABI.

Import ssreflect ssreflect.ssrfun ssreflect.eqtype ssreflect.tuple.
Import Arith Core Common Coder Ergs memory.Depot MemoryManagement Pointer
TransientMemory lib.BitsExt FatPointerABI ForwardPageTypesABI.

Section FarCallABI.
```

This record describes all the parameters that `FarCalls` can use.

```
Record params :=
  mk_params {
      fwd_memory: fwd_memory;
      ergs_passed: ergs;
      shard_id: shard_id;
      constructor_call: bool;
      to_system: bool;
    }.
```

This record describes the layout of 256-bit word that encodes these parameters.

```
Record params_layout :=
  mk_params_layout {
```

```
      raw_to_system_bool: u8;
      raw_constructor_call_bool: u8;
      raw_shard_id: u8;
      raw_memory_forwarding_type_enum : u8;
      raw_ergs_passed: u32;
      (*raw_reserved: u64; *)
      raw_fat_ptr_layout: fat_ptr_layout;
    }.

Section BinaryCoder.
  Open Scope ZMod_scope.
  Definition encode_bin : @encoder word params_layout :=
    fun params =>
      match params with
       | mk_params_layout to_system8 constructor_call8 shard_id8
memory_forwarding_type8 ergs_passed32 (* reserved64 *) fat_ptr_layout128 =>
          match FatPointerABI.encode_bin fat_ptr_layout128 with
          | Some ptr_encoded128 => Some (
                                        to_system8 ## constructor_call8 ##
shard_id8 ## memory_forwarding_type8 ##
                                          ergs_passed32 ##
                                          zero64 ##
                                          ptr_encoded128)
          | None => None
          end
      end
  .

  Definition decode_bin : @decoder word params_layout :=
    fun w =>
      let to_system : u8 := w { 256-8, 256 } in
      let constructor_call : u8 := w { 256 - 2×8, 256 - 8 } in
      let shard_id : u8 := w { 256 - 3×8, 256 - 2×8 } in
      let memory_forwarding_type : u8 := w { 256 - 4×8, 256 - 3×8 } in
      let ergs_passed : u32 := w { 256 - 2×32, 256 - 32 } in
      let ptr : option fat_ptr_layout := FatPointerABI.decode_bin (low 128 w)
in
      match ptr with
      | Some ptr => Some (mk_params_layout to_system constructor_call shard_id
memory_forwarding_type ergs_passed ptr )
      | None => None
      end.

  Theorem binary_coder_revertible: revertible decode_bin encode_bin.

  Definition coder_binary : @coder word params_layout :=
    mk_coder decode_bin encode_bin binary_coder_revertible.

End BinaryCoder.

Section LayoutCoder.

  Definition encode_layout: @encoder params_layout params :=
fun params =>
  match params with
  | mk_params (ForwardExistingFatPointer fptr) ergs_passed shard_id
constructor_call to_system =>
```

```
      match FatPointerABI.encode_layout fptr with
        | Some ptr_layout ⇒
          Some
            {|
              raw_to_system_bool:= if to_system then # 1 else # 0;
              raw_constructor_call_bool := if constructor_call then # 1 else
# 0;
              raw_shard_id := shard_id;
              raw_memory_forwarding_type_enum :=
FarCallForwardPageType.ForwardFatPointer;
              raw_ergs_passed:= ergs_passed;
              (*raw_reserved: u64; *)
              raw_fat_ptr_layout:= ptr_layout;
            |}
        | None ⇒ None
        end
    | mk_params (ForwardNewFatPointer heap_variant span) ergs_passed shard_id
constructor_call to_system ⇒
        Some
          {|
            raw_to_system_bool:= if to_system then # 1 else # 0;
            raw_constructor_call_bool := if constructor_call then # 1 else # 0;
            raw_shard_id := shard_id;
            raw_memory_forwarding_type_enum := data_page_type_to_u8 heap_variant;
            raw_ergs_passed := ergs_passed;
            (*raw_reserved: u64; *)
            raw_fat_ptr_layout:= {|
                                  start := span.(s_start);
                                  length := span.(s_length);
                                  page := zero32;
                                  offset := zero32;
                                |};
          |}
    end.

    Definition decode_layout: @decoder params_layout params :=
      fun layout ⇒
        match layout with
        | mk_params_layout raw_to_system_bool raw_constructor_call_bool
raw_shard_id raw_memory_forwarding_type_enum raw_ergs_passed raw_fat_ptr_layout
⇒
          match fwd_memory_adapter raw_memory_forwarding_type_enum
raw_fat_ptr_layout with
          | Some fwd_memory_value ⇒ Some (
                                        {|
                                          fwd_memory := fwd_memory_value;
                                          ergs_passed := raw_ergs_passed;
                                          shard_id := raw_shard_id;
                                          constructor_call :=
raw_constructor_call_bool != zero8;
                                          to_system := raw_to_system_bool !=
zero8;
                                        |})
          | None ⇒ None
          end
        end
    .
```

```
  Theorem layout_coding_revertible: revertible decode_layout encode_layout.

  Definition coder_layout := mk_coder decode_layout encode_layout
layout_coding_revertible.

End LayoutCoder.

Definition coder := coder_compose coder_binary coder_layout.
End FarCallABI.
```

Library EraVM.ABI.FarRetABI

```
Require Coder Ergs MemoryManagement Pointer lib.BitsExt ABI.FatPointerABI
ABI.ForwardPageTypesABI.

Import ssreflect ssreflect.ssrfun ssreflect.eqtype ssreflect.tuple.
Import Arith Core Common Coder Ergs MemoryManagement Pointer TransientMemory
lib.BitsExt FatPointerABI ForwardPageTypesABI.
```

This record describes all the parameters that far returns can use.

```
Section FarRetABI.
  Record params :=
  mk_params {
      fwd_memory: fwd_memory;
    }.
```

This record describes the layout of 256-bit word that encodes these parameters.

```
Record params_layout :=
  mk_params_layout {
      (* reserved 3 bytes *)
      raw_memory_forwarding_type_enum : u8;
      (* last 16 bytes contain the pointer *)
      raw_fat_ptr_layout: fat_ptr_layout;
    }.

Section BinaryCoder.
  Open Scope ZMod_scope.
  Definition encode_bin : @encoder word params_layout :=
    fun params =>
      match params with
       | mk_params_layout memory_forwarding_type8 fat_ptr_layout128 =>
          match FatPointerABI.encode_bin fat_ptr_layout128 with
          | Some ptr_encoded128 => Some (zero8 ## zero8 ## zero8 ##
memory_forwarding_type8 ##
                                         zero32 ##
                                         zero64 ##
```

```
                                        ptr_encoded128)
          | None ⇒ None
          end
      end
  .

  Definition decode_bin : @decoder word params_layout :=
    fun w ⇒
      let memory_forwarding_type : u8 := w { 256 - 4×8, 256 - 3×8 } in
      let ptr : option fat_ptr_layout := FatPointerABI.decode_bin (low 128 w)
in
      match ptr with
      | Some ptr ⇒ Some (mk_params_layout memory_forwarding_type ptr)
      | None ⇒ None
      end.

  Theorem binary_coder_revertible: revertible decode_bin encode_bin.

  Definition coder_binary : @coder word params_layout :=
    mk_coder decode_bin encode_bin binary_coder_revertible.

End BinaryCoder.

Section LayoutCoder.

 Definition encode_layout: @encoder params_layout params :=
fun params ⇒
  match params with
  | mk_params (ForwardExistingFatPointer fptr) ⇒
      match FatPointerABI.encode_layout fptr with
      | Some ptr_layout ⇒
            Some
              {|
                raw_memory_forwarding_type_enum :=
FarCallForwardPageType.ForwardFatPointer;
                raw_fat_ptr_layout:= ptr_layout;
              |}
      | None ⇒ None
      end
  | mk_params (ForwardNewFatPointer heap_variant span) ⇒
      Some
        {|
          raw_memory_forwarding_type_enum := data_page_type_to_u8 heap_variant;
          raw_fat_ptr_layout:= {|
                                 start := span.(s_start);
                                 length := span.(s_length);
                                 page := zero32;
                                 offset := zero32;
                               |};
        |}
  end.

  Definition decode_layout: @decoder params_layout params :=
    fun layout ⇒
      match layout with
      | mk_params_layout raw_memory_forwarding_type_enum raw_fat_ptr_layout ⇒
          match fwd_memory_adapter raw_memory_forwarding_type_enum
```

```
raw_fat_ptr_layout with
          | Some fwd_memory_value ⇒ Some (mk_params fwd_memory_value)
          | None ⇒ None
          end
      end
  .
  Theorem layout_coding_revertible: revertible decode_layout encode_layout.

  Definition coder_layout := mk_coder decode_layout encode_layout
layout_coding_revertible.

End LayoutCoder.

Definition coder := coder_compose coder_binary coder_layout.
End FarRetABI.
```

Library EraVM.ABI.MetaParametersABI

```
Require Coder Ergs memory.Depot MemoryManagement Pointer TransientMemory
lib.BitsExt.

Import ssreflect ssreflect.ssrfun ssreflect.eqtype ssreflect.tuple.
Import Arith Core Common Coder Ergs memory.Depot TransientMemory
MemoryManagement Pointer lib.BitsExt.

Section MetaParametersABI.
```

Meta parameters are returned by `OpContextMeta`. This record shows the data that is contained in it.

```
Record params :=
  mk_params {
      ergs_per_pubdata_byte: ergs;
      heap_size: mem_address;
      aux_heap_size: mem_address;
      this_shard_id: shard_id;
      caller_shard_id: shard_id;
      code_shard_id: shard_id;
    }.
```

This record describes the exact memory layout in a 256-bit word, which holds parameters.

```
Record params_layout :=
  mk_params_layout {
      (* reserved: u8 *)
      l_code_shard_id : u8;
      l_caller_shard_id : u8;
      l_this_shard_id : u8;
```

```
    (* reserved: u96 *)
     l_aux_heap_size : u32;
     l_heap_size : u32;
     (* reserved: u32 *)
     l_ergs_per_pubdata_byte: u32;
  }.

Section BinaryCoder.
  #[local] Open Scope ZMod_scope.
  Context (zero96 : BITS 96 := # 0).

  Definition encode_bin : @encoder word params_layout :=
    fun layout =>
      match layout with
      | mk_params_layout
          l_code_shard_id8
          l_caller_shard_id8
          l_this_shard_id8
          l_aux_heap_size32
          l_heap_size32
          l_ergs_per_pubdata_byte32
        => Some (
              zero8 ## l_code_shard_id8 ## l_caller_shard_id8 ##
l_this_shard_id8 ## zero96 ## l_aux_heap_size32 ## l_heap_size32 ## zero32 ##
l_ergs_per_pubdata_byte32)
      end.

  Definition decode_bin: @decoder word params_layout :=
    fun w =>
      Some {|
          (* reserved: u8 *)
          l_code_shard_id := w { 256 - 2×8 , 256 - 8} ;
          l_caller_shard_id := w {256 - 3 × 8, 256 - 2 × 8};
          l_this_shard_id := w {256 - 4 × 8, 256 - 3 × 8};
          (* reserved: u96 *)
          l_aux_heap_size := w { 3 × 32, 4 × 32 };
          l_heap_size := w { 2 × 32, 3 × 32 };
          (* reserved: u32 *)
          l_ergs_per_pubdata_byte:= w { 0, 32 };
        |}.

  Theorem binary_coder_revertible: revertible decode_bin encode_bin.

  Definition binary_coder := mk_coder decode_bin encode_bin
binary_coder_revertible.
End BinaryCoder.

Section LayoutCoder.

Definition decode_layout : @decoder params_layout params :=
  fun layout =>
    match layout with
    | mk_params_layout l_code_shard_id l_caller_shard_id l_this_shard_id
l_aux_heap_size l_heap_size
       l_ergs_per_pubdata_byte => Some
                                    {|
                                      ergs_per_pubdata_byte :=
```

```
l_ergs_per_pubdata_byte;
                                    heap_size := l_heap_size;
                                    aux_heap_size := l_aux_heap_size;
                                    this_shard_id := l_this_shard_id;
                                    caller_shard_id := l_caller_shard_id;
                                    code_shard_id := l_code_shard_id;
                                  |}
    end.

Definition encode_layout : @encoder params_layout params :=
  fun params =>
    match params with
    | mk_params ergs_per_pubdata_byte heap_size aux_heap_size this_shard_id
caller_shard_id code_shard_id => Some
                                  {|
                                    l_ergs_per_pubdata_byte :=
ergs_per_pubdata_byte;
                                    l_heap_size := heap_size;
                                    l_aux_heap_size := aux_heap_size;
                                    l_this_shard_id := this_shard_id;
                                    l_caller_shard_id := caller_shard_id;
                                    l_code_shard_id := code_shard_id;
                                  |}
    end.

Theorem layout_coder_revertible: revertible decode_layout encode_layout.

  Definition layout_coder := mk_coder decode_layout encode_layout
layout_coder_revertible.
End LayoutCoder.

Definition coder := coder_compose binary_coder layout_coder.
End MetaParametersABI.
```

Library EraVM.ABI.PrecompileParametersABI

```
Require Coder TransientMemory lib.BitsExt.

Import ssreflect ssreflect.ssrfun ssreflect.eqtype ssreflect.tuple.
Import Core Common Coder TransientMemory lib.BitsExt.

Section PrecompileParametersABI.

  Record params :=
  mk_params
    {
      input_memory_offset: mem_address;
      input_memory_length: mem_address;
      output_memory_offset: mem_address;
      output_memory_length: mem_address;
      per_precompile_interpreted: u64;
      memory_page_to_read: page_id;
```

```
      memory_page_to_write: page_id;
      precompile_interpreted_data: u64;
    }.
```

equality

```
Definition params_eqn (x y: params) : bool :=
  (x.(input_memory_offset) == y.(input_memory_offset)) &&
    (x.(input_memory_length) == y.(input_memory_length)) &&
    (x.(output_memory_offset) == y.(output_memory_offset)) &&
    (x.(output_memory_length) == y.(output_memory_length)) &&
    (x.(per_precompile_interpreted) == y.(per_precompile_interpreted)) &&
    (x.(memory_page_to_read) == y.(memory_page_to_read)) &&
    (x.(memory_page_to_write) == y.(memory_page_to_write)) &&
    (x.(precompile_interpreted_data) == y.(precompile_interpreted_data)).

Lemma params_eqnP : Equality.axiom params_eqn.
Proof.
  move => x y.
  unfold params_eqn.
  destruct x,y =>//=.
  repeat match goal with
           [ |- context [(?x == ?y)] ] =>
             let Hf := fresh "H" in
             destruct (x == y) eqn: Hf; try rewrite Hf
         end =>//=; constructor; auto;
    repeat match goal with [ H: (?x == ?y) = true |- _] => move: H => /eqP →
end;
    try injection; intros; subst;
    repeat match goal with [ H: (?x == ?x) = false |- _] => rewrite eq_refl in H
end =>//=.
  - constructor.
Qed.

Canonical params_eqMixin := EqMixin params_eqnP.
Canonical params_eqType := Eval hnf in EqType _ params_eqMixin.
Axiom ABI: @coder word params.
End PrecompileParametersABI.
```

Library EraVM.encoding.EncodingUtils

```
Require Common Predication isa.Modifiers isa.GeneratedMachISA.

Import ZBits ZArith.
Import Common GPR GeneratedMachISA Modifiers Predication.

Section EncodingTools.
```

Encoding parts of instructions

The encoding of `asm_instruction` is described in two stages:

1. `asm_to_mach` describes how fields `asm_instruction` are mapped to the EraVM machine instruction `mach_instruction`; this describes the instruction layout.
2. `encode_mach_instruction` describes the binary encoding of `mach_instruction`.

EraVM instructions `mach_instruction` are 64 bits wide. Starting from the least significant bit, they are made of the following parts:

- `mach_opcode` (11 bits)
- 2 reserved bits
- `predicate` (3 bits)
- `reg_name`s (4 x 4 bits)
- immediate `u16` values (2 x 16 bits, big endian)

The precise format is formalized by `encode_mach_instruction`.

Their encoding is independent and happens as follows:

1. The `mach_opcode` packs together the information about

- the instruction meaning, e.g. is it addition, or call, or something else;
- its source and destination addressing modes. For example, in the encoding of `add r1, r0, r3` the meaning of the register `R1` in the field `op_src0` is “use the value from register r1 as the first operand”. But in the encoding of `add stack=[r1], r0, r3` the meaning of the register `R1` in the field `op_src0` is “use the value on the stack page by address R1 as the first operand”. This difference is reflected in the `mach_opcode` field: the addressing modes are encoded by `encode_src_mode`/`encode_src_special_mode` (for selected instructions)/`encode_dst_mode`) and mixed in `mach_opcode` by `encode_opcode`.

```
Definition encode_src_mode (sm:src_mode) : Z :=
  match sm with
  | SrcReg ⇒ 0
  | SrcSpRelativePop ⇒ 1
  | SrcSpRelative ⇒ 2
  | SrcStackAbsolute ⇒ 3
  | SrcImm ⇒ 4
  | SrcCodeAddr ⇒ 5
  end
.

Definition encode_src_special_mode (sm:src_special_mode) : Z :=
  match sm with
  | SrcSpecialReg ⇒ 0
  | SrcSpecialImm ⇒ 10
  end
.

Definition encode_dst_mode (sm:dst_mode) : Z :=
  match sm with
```

```
  | DstReg ⇒ 0
  | DstSpRelativePush ⇒ 1
  | DstSpRelative ⇒ 2
  | DstStackAbsolute ⇒ 3
  end
.
```

The definitions `encode_set_flags` and `encode_swap` are encoding the `mod_set_flags` and `mod_swap` modifier values as one-bit numbers.

```
Definition encode_set_flags (m: mod_set_flags) : Z :=
  match m with
  | SetFlags ⇒ 1
  | PreserveFlags ⇒ 0
  end.
Definition encode_swap(m: mod_swap) : Z :=
  match m with
  | Swap ⇒ 1
  | NoSwap =>0
  end
.
```

2. [encode_predicate] maps eight different predicates to their encodings as 3-bit binary numbers.

```
Definition encode_predicate (p:predicate) : BITS 3 :=
  # match p with
    | IfAlways ⇒ 0
    | IfGT ⇒ 1
    | IfLT ⇒ 2
    | IfEQ ⇒ 3
    | IfGE ⇒ 4
    | IfLE ⇒ 5
    | IfNotEQ ⇒ 6
    | IfGTOrLT ⇒ 7
    end
.
```

3. Registers are encoded by their indices as 4-bit numbers, e.g. register `R3` is encoded as $3 = 0011_2$. The meaning of two source and two destination registers depends on the instruction and/or addressing modes.

```
Definition encode_reg (name:reg_name) : BITS 4 :=
  # (reg_idx name)
.

Definition encode_reg_opt (name:option reg_name) : BITS 4 :=
```

```
  match name with
  | Some name => encode_reg name
  | None => # 0
  end
.
```

4. Immediate 16-bit values are encoded as-is.

For example, the instruction `sub r0, r1, r2` will be encoded as `000000000210004B`, which, in binary form, is:

```
0000000000000000 0010 0001 0000 000 00 00001001011
```

Let's break it down:

```
| 0000000000000000 -> imm1
| 0000000000000000 -> imm0
| 0000 -> Dst1: R0 (4 bits, ignored)
| 0010 -> Dst0: R2 (4 bits)
| 0001 -> Src1: R1 (4 bits)
| 0000 -> Src0: R0 (4 bits)
| 000 -> Predicate: IfAlways (3 bits)
| 00 -> Reserved (2 bits)
|  00001001011 -> Opcode (11 bits)
```

Another example: `sub stack=[r1+15], r2, stack+=[r3+63]` is encoded as: `003f000f0321007d`, which, in binary form, is:

```
0000 0000 0011 1111 0000 0000 0000 1111 0000 0011 0010 0001 0000 0000 0111 1101
```

Let's break it down:

```
| 0000 0000 0011 1111  -> imm1: 63
| 0000 0000 0000 1111 -> imm0: 15
| 0000 -> dst1: R0 (4 bits, ignored)
| 0011 -> dst0: R3 (4 bits)
| 0010 -> src1: R2 (4 bits)
| 0001 ->  src0: R1 (4 bits)
| 000 -> Predicate: IfAlways (3 bits)
| 00 -> Reserved (2 bits)
| 000 0111 1101 -> Opcode (11 bits)
```

Be careful with the big-endian byte order in multibyte numbers.

```
End EncodingTools.
```

Library EraVM.encoding.GeneratedEncodeOpcode

```
(* GENERATED FILE, DO NOT EDIT MANUALLY. *)
Require isa.Modifiers encoding.EncodingUtils.
```

```
Import ZArith.
Import GeneratedMachISA Modifiers encoding.EncodingUtils.

Section OpcodeEncoderDefinition.
Coercion encode_dst_mode : dst_mode >-> Z.
Coercion encode_src_mode : src_mode >-> Z.
Coercion encode_src_special_mode : src_special_mode >-> Z.
Coercion encode_swap: mod_swap >-> Z.
Coercion encode_set_flags : mod_set_flags >-> Z.
Coercion Z.b2z: bool >-> Z.

Definition encode_opcode (op:mach_opcode) : Z :=
   match op with
| OpInvalid ⇒ 0
| OpNoOp src dst ⇒ 1 + 4 × src + 1 × dst
| OpAdd src dst set_flags ⇒ 25 + 8 × src + 2 × dst + set_flags
| OpSub src dst swap set_flags ⇒ 73 + 16 × src + 4 × dst + 2 × set_flags + swap
| OpMul src dst set_flags ⇒ 169 + 8 × src + 2 × dst + set_flags
| OpDiv src dst swap set_flags ⇒ 217 + 16 × src + 4 × dst + 2 × set_flags +
swap
| OpJump src ⇒ 313 + 1 × src
| OpXor src dst set_flags ⇒ 319 + 8 × src + 2 × dst + set_flags
| OpAnd src dst set_flags ⇒ 367 + 8 × src + 2 × dst + set_flags
| OpOr src dst set_flags ⇒ 415 + 8 × src + 2 × dst + set_flags
| OpShl src dst swap set_flags ⇒ 463 + 16 × src + 4 × dst + 2 × set_flags +
swap
| OpShr src dst swap set_flags ⇒ 559 + 16 × src + 4 × dst + 2 × set_flags +
swap
| OpRol src dst swap set_flags ⇒ 655 + 16 × src + 4 × dst + 2 × set_flags +
swap
| OpRor src dst swap set_flags ⇒ 751 + 16 × src + 4 × dst + 2 × set_flags +
swap
| OpPtrAdd src dst swap ⇒ 847 + 8 × src + 2 × dst + swap
| OpPtrSub src dst swap ⇒ 895 + 8 × src + 2 × dst + swap
| OpPtrPack src dst swap ⇒ 943 + 8 × src + 2 × dst + swap
| OpPtrShrink src dst swap ⇒ 991 + 8 × src + 2 × dst + swap
| OpCall ⇒ 1039
| OpContextThis ⇒ 1040
| OpContextCaller ⇒ 1041
| OpContextCodeAddress ⇒ 1042
| OpContextMeta ⇒ 1043
| OpContextErgsLeft ⇒ 1044
| OpContextSp ⇒ 1045
| OpContextGetContextU128 ⇒ 1046
| OpContextSetContextU128 ⇒ 1047
| OpContextSetErgsPerPubdataByte ⇒ 1048
| OpContextIncrementTxNumber ⇒ 1049
| OpSload ⇒ 1050
| OpSstore ⇒ 1051
| OpLogToL1 is_first ⇒ 1052 + is_first
| OpLogEvent is_first ⇒ 1054 + is_first
| OpLogPrecompile ⇒ 1056
| OpFarcall is_shard is_static ⇒ 1057 + 2 × is_static + is_shard
| OpDelegate is_shard is_static ⇒ 1061 + 2 × is_static + is_shard
| OpMimic is_shard is_static ⇒ 1065 + 2 × is_static + is_shard
| OpRet to_label ⇒ 1069 + to_label
```

```
| OpRevert to_label ⇒ 1071 + to_label
| OpPanic to_label ⇒ 1073 + to_label
| OpLoadHeap src inc ⇒ 1075 + 10 × src + inc
| OpStoreHeap src inc ⇒ 1077 + 10 × src + inc
| OpLoadAuxHeap src inc ⇒ 1079 + 10 × src + inc
| OpStoreAuxHeap src inc ⇒ 1081 + 10 × src + inc
| OpLoadPtr inc ⇒ 1083 + inc
  end
.

End OpcodeEncoderDefinition.
```

Library EraVM.encoding.Encoder

```
From RecordUpdate Require Import RecordSet.
Require
  isa.Modifiers
  isa.GeneratedMachISA
  isa.AssemblyToMach
  encoding.EncodingUtils
  encoding.GeneratedEncodeOpcode.

Section MachEncoder.
Import ZArith.
Import Assembly Common GeneratedMachISA GeneratedEncodeOpcode
isa.AssemblyToMach Predication EncodingUtils.
```

Encoding machine instructions

For an overview of instruction sets and different layers of instructions definitions, used in this specification, refer to `InstructionSets`.

The binary encoding is defined for `mach_instruction`, which is a representation of an EraVM instruction aware of its encoding and layout. Once the `asm_instruction` has been transformed into `mach_instruction` via `asm_to_mach`, it is trivial to put all of `mach_instruction`'s fields in binary form via `encode_mach_instruction`.

Reminder: `##` notation stands for concatenating binary strings; `a ## b` signifies that `b` holds less significant bits and `a` is prepended to it, forming a more significant part.

```
Definition encode_mach_instruction (i:@mach_instruction GPR.reg_name u16) :
BITS 64 :=
  match i with
  | mk_ins op_code op_predicate op_src0 op_src1 op_dst0 op_dst1 op_imm0 op_imm1
⇒
     let reserved2 := @fromNat 2 0 in
          op_imm1
       ## op_imm0
       ## encode_reg op_dst1
```

```
        ## encode_reg op_dst0
        ## encode_reg op_src1
        ## encode_reg op_src0
        ## encode_predicate op_predicate
        ## reserved2
        ## @fromZ 11 (encode_opcode op_code)
  end
.

Definition encode_asm (i:predicated asm_instruction) : option (BITS 64) :=
  option_map encode_mach_instruction (asm_to_mach i)
.
End MachEncoder.
```

Library EraVM.Bootloader

```
Require memory.Depot.
Import memory.Depot.

Section Bootloader.
  Import ZArith spec.
  Open Scope Z.
```

Bootloader

Bootloader is a system contract in charge of block construction (**sources**).

Formally, bootloader is assigned an address `BOOTLOADER_SYSTEM_CONTRACT_ADDRESS`, but on execution start EraVM decommits its code directly by its `versioned_hash`.

```
  Definition BOOTLOADER_SYSTEM_CONTRACT_ADDRESS : contract_address := fromZ
((2^15) + 1).
```

Using the bootloader `versioned_hash`, EraVM queries the bootloader code from decommitter and starts executing it.

The heap page of the bootloader is different from other pages: it acts as an interface between server and EraVM. Server is able to modify the contents of this page at will. Server gradually fills it with transaction data, formatted according to an implementation-defined convention.

The bootloader then acts roughly as the following code (not an actual implementation):

```
contract Bootloader {
    function executeBlock(
        address operatorAddress,
      Transaction[2] memory transactions
    ) {
```

```
     for(uint i = 0; i < transactions.length; i++) {
        validateTransaction(transactions[i]);
        chargeFee(operatorAddress, transactions[i]);
        executeTransaction(transactions[i]);
     }
  }

    function validateTransaction(Transaction memory tx) {
    // validation logic
    }
  function chargeFee(address operatorAddress, Transaction memory tx) {
    // charge fee
    }
  function executeTransaction(Transaction memory tx) {
    // execution logic
  }
}
```

The bootloader is therefore responsible for:

- validating transactions;
- executing transactions to form a new block;
- setting some of the transaction- or block-wide transaction parameters (e.g. blockhash, tx.origin).

Server makes a snapshot of EraVM state after completing every transaction. When the bootloader encounters a malformed transaction, it fails, and the server restarts EraVM from the most recent snapshot, skipping this transaction. If a transaction is well-formed, EraVM may still panic while handling it outside the bootloader code. This is a normal situation and is handled by EraVM in a regular way, through panics. See e.g. `OpPanic`.

The exact code of the bootloader is a part of a protocol; its `versioned_hash` is included in the block header.

```
End Bootloader.
```

Library EraVM.Precompiles

```
Require memory.Depot ABI State.

Import ABI memory.Depot State.

Section Precompiles.
```

Precompiles

Precompiles are extensions of VM bound to one of the system contracts. When this contract executes an instruction `OpPrecompileCall`, VM executes an algorithm specific to this contract.

This requires preparing data for the precompiled algorithm in a special, algorithm-dependent way.

Precompiles are able to change `data_page`s.

Precompiles may fail.

Precompiles are not revertable; their functioning is not affected by rollbacks.

Currently we formalize precompiles as a black box.

```
  Parameter precompile_processor : contract_address →
PrecompileParametersABI.params → transient_state → transient_state → Prop.
End Precompiles.
```

Library EraVM.VersionedHash

```
Require Coder PrimitiveValue TransientMemory.

Section VersionedHash.
  Import ZArith ssrbool eqtype ssreflect ssrfun ssrbool ssreflect.eqtype
ssreflect.tuple zmodp.
  Import Coder Common TransientMemory.

  Context {word: Type}.
  Context {ins_type: Type} (invalid_ins: ins_type).
```

Versioned hash

`versioned_hash` is a hash augmented with additional information. It is used as a key to identify the contract code for decommitter.

Additional information includes:

- `VERSION_BYTE` (currently 1)
- `marker` (is the contract being constructed or already constructed?)

The hash itself is described by `partial_hash`; it is computed as SHA256 hash modulo $2^{28\times 8}$.

```
  Definition VERSION_BYTE: u8 := fromZ 1%Z.

  Inductive marker := CODE_AT_REST | YET_CONSTRUCTED | INVALID.
```

decidable equality

```
  Scheme Equality for marker.
  Lemma marker_eqP : Equality.axiom marker_beq.
  Proof. by move ⇒ [] []; constructor. Qed.
```

```
  Canonical marker_eqMixin := EqMixin marker_eqP.
  Canonical marker_eqType := Eval hnf in EqType marker marker_eqMixin.
  Definition marker_valid (m: marker) :=
    match m with
    | INVALID ⇒ false
    | _ ⇒ true
    end.

  Record versioned_hash := mk_vhash {
                             code_length_in_words: u16;
                             extra_marker: marker;
                             partial_hash: BITS (28×bits_in_byte)%nat
                           }.
  Axiom hash_coder: @Coder.coder word versioned_hash.
```

EraVM accepts `DEFAULT_AA_VHASH` as a parameter. See also `Parameters`.

```
  Parameter DEFAULT_AA_VHASH: versioned_hash.

  Open Scope ZMod_scope.
```

equality on versioned hashes

```
  Definition eqn (x y:versioned_hash) : bool :=
    match x,y with
    | mk_vhash l1 em1 ph1 , mk_vhash l2 em2 ph2 ⇒
        (l1 == l2) && (em1 == em2) && (ph1 == ph2)
    end.

  Lemma eqnP : Equality.axiom eqn.
  Proof.
    move ⇒ [l1 em1 ph1] [l2 em2 ph2].
    simpl.
    destruct (l1 == l2) eqn: H1;
      destruct (em1 == em2) eqn: H2;
      destruct (ph1 == ph2) eqn: H3;
      try move: (eqP H1) ⇒ ->; try move: (eqP H2) ⇒ ->; try move: (eqP H3) ⇒
->; constructor =>//.
    - injection. move ⇒ ?; subst. by rewrite eq_refl in H3.
    - injection. move ⇒ ?; subst. by rewrite eq_refl in H2.
    - injection. move ⇒ ?; subst. by rewrite eq_refl in H3.
    - injection. move ⇒ ?; subst. by rewrite eq_refl in H1.
    - injection. move ⇒ ?; subst. by rewrite eq_refl in H3.
    - injection. move ⇒ ?; subst. by rewrite eq_refl in H2.
    - injection. move ⇒ ?; subst. by rewrite eq_refl in H3.
  Qed.

  Canonical vh_eqMixin := EqMixin eqnP.
  Canonical vh_eqType := Eval hnf in EqType _ vh_eqMixin.

End VersionedHash.
```

Library EraVM.Decommitter

```
Require Common ABI lib.Decidability History memory.Depot MemoryOps
VersionedHash.

Import Coder Core History VersionedHash Common Decidability Ergs MemoryBase
memory.Depot TransientMemory ZArith ABI bits.

Section Decommitter.
  Open Scope Z.
  Open Scope ZMod_scope.
```

Decommitter

Decommitter is a module external to EraVM, a key-value storage where:

- key is `versioned_hash`
- value is the contract code

Decommitting refers to querying contract code from the decommitter and filling new `code_page` and `const_page` with it. The rest of the code page is filled with invalid instructions; the rest of const page is filled with zeros.

The mapping `code hash location` between contract addresses and the hashes of their codes is implemented by the storage of the contract `DEPLOYER_SYSTEM_CONTRACT_ADDRESS`.

Note: storage with the same contract address may differ between shards.

```
  Definition DEPLOYER_SYSTEM_CONTRACT_ADDRESS : contract_address := fromZ
((2^15) + 2).

  Definition code_hash_location (for_contract: contract_address)
(sid:Depot.shard_id): fqa_key :=
    mk_fqa_key (mk_fqa_storage sid DEPLOYER_SYSTEM_CONTRACT_ADDRESS) (widen
word_bits for_contract).

  Context {ins_type: Type} (invalid_ins: ins_type) (code_page := code_page
invalid_ins)
    (empty_code : code_page := @mk_code_page _ invalid_ins (empty _))
    (empty_const: const_page := (empty _ )).

  Definition code_storage_params := {|
                                      addressable_block := code_page ×
const_page;
                                      address_bits := 256;
                                      default_value := (empty_code,
empty_const);
                                      writable := false;
                                    |}.
```

```
  Definition code_storage: Type := mem_parameterized code_storage_params.
Import VersionedHash.

  Record decommitter :=
    mk_code_mgr {
        cm_storage: code_storage;
        cm_accessed: history vh_eqType;
      }.
```

The versioned hash is called **cold** if it was not accessed during construction of the current block. Otherwise, it is **warm**. See `is_first_access`.

```
  Definition is_first_access cm vh := negb (contains _ (cm_accessed cm) vh).
```

Decommitting code by a cold versioned hash costs (`ERGS_PER_CODE_WORD_DECOMMITTMENT` * (block size in words)) ergs. Decommitting warm code is free.

```
  Inductive decommitment_cost (cm:decommitter) vhash (code_length_in_words:
code_length): ergs → Prop :=
  |dc_fresh: ∀ bigcost cost,
      is_first_access cm vhash = true→
      bigcost = ergs_of Ergs.ERGS_PER_CODE_WORD_DECOMMITTMENT × (zeroExtend
(ergs_bits - code_address_bits) code_length_in_words) →
      (toZ bigcost ≤ unsigned_max ergs_bits)%Z →
      cost = low ergs_bits bigcost →
      decommitment_cost cm vhash code_length_in_words cost
  |dc_not_fresh:
    is_first_access cm vhash = false →
    decommitment_cost cm vhash code_length_in_words zero32.

  Inductive code_fetch_hash (d:depot) (cs: code_storage) (sid: Depot.shard_id)
(contract_addr: contract_address) :
    option (versioned_hash × code_length) → Prop :=
  |cfh_found: ∀ hash_enc code_length_in_words extra_marker partial_hash,
      storage_read d (code_hash_location contract_addr sid) hash_enc →
      hash_enc ≠ zero256 →
      marker_valid extra_marker = true →
      hash_coder.(Coder.decode) hash_enc = Some (mk_vhash code_length_in_words
extra_marker partial_hash) →
      code_fetch_hash d cs sid contract_addr (Some (mk_vhash
code_length_in_words extra_marker partial_hash, code_length_in_words))

  | cfh_not_found:
    storage_read d (code_hash_location contract_addr sid) zero256 →
    code_fetch_hash d cs sid contract_addr None.
```

When decommitter does not have code for the requested hash, VM may allow masking, and request the

code with default versioned hash `DEFAULT_AA_VHASH` instead. It is expected that:

- `DEFAULT_AA_VHASH` is well-formed (see `marker_valid`), and
- decommitter has code matching `DEFAULT_AA_VHASH`.

VM does not mask the code for system contracts; see `sem.FarCall.step`.

`DEFAULT_AA_CODE` is the decommitter's answer to the query `DEFAULT_AA_VHASH`.

```
  Parameter DEFAULT_AA_CODE: (code_page× const_page).

  Inductive code_fetch (d:depot) (cs: code_storage) (sid: Depot.shard_id)
(contract_addr: contract_address) :
    bool → (versioned_hash × (code_page × const_page) × code_length) → Prop :=
  | cfnm_no_masking: ∀ vhash (code_storage:code_storage) code_length_in_words
pages0 masking,
      code_fetch_hash d cs sid contract_addr (Some (vhash,
code_length_in_words)) →
      load_result code_storage_params (widen 256 (partial_hash vhash))
code_storage pages0 →

      code_fetch d cs sid contract_addr masking (vhash, pages0,
code_length_in_words)
  | cfnm_masking: ∀ (code_storage:code_storage) code_length_in_words,
      code_fetch_hash d cs sid contract_addr None →
      code_fetch d cs sid contract_addr true (DEFAULT_AA_VHASH,
DEFAULT_AA_CODE,code_length_in_words).

End Decommitter.
```

Library EraVM.Arith

```
From mathcomp Require ssreflect ssrfun ssrbool eqtype tuple zmodp.
Require Types.
Import Types ssreflect.tuple ssreflect.eqtype ssrbool.
```

Common project-independent definitions

```
Section Operations.
```

```
  Import operations ZArith.

  Record udiv_result {n} := mk_divrem { div: BITS n; rem: BITS n }.

  Definition uadd_of {n: nat} (a: BITS n) (b:BITS n) : bool × BITS n := adcB
false a b.
  Definition uadd_wrap {n: nat} (a: BITS n) (b:BITS n) : BITS n := snd (uadd_of
a b).
  Definition uinc_of {n: nat} (a: BITS n) : bool × BITS n := uadd_of a (fromZ
1).
  Definition uinc_by_32_of {n: nat} (a: BITS n) : bool × BITS n := uadd_of a
(fromZ 32).

  Definition usub_uf {n: nat} (a: BITS n) (b:BITS n) : bool × BITS n := sbbB
false a b.
  Definition umul {n: nat} (a: BITS n) (b:BITS n) : BITS (n+n) := fullmulB a b.

  Definition udiv {n: nat} (a: BITS (S n)) (b:BITS (S n)) : udiv_result :=
    let za := toZ a in
    let zb := toZ b in
    @mk_divrem (S n) (fromZ (BinInt.Z.div za zb)) (fromZ (BinInt.Z.rem za zb)).

  Definition lt_unsigned {n:nat} (a b: BITS n) := ltB a b.
  Definition gt_unsigned {n:nat} (a b: BITS n) := lt_unsigned b a.
  Definition le_unsigned {n:nat} (a b: BITS n) := lt_unsigned a b || (a == b).
  Definition ge_unsigned {n:nat} (a b: BITS n) := gt_unsigned a b || (a == b).

  Definition bitwise_xor {n} := @xorB n.
  Definition bitwise_or {n} := @orB n.
  Definition bitwise_and {n} := @andB n.

  Definition characteristic (n:nat): Z := 2 ^ (Z.of_nat n).
  Definition unsigned_max (n:nat) :Z := characteristic n - 1.
  Definition unsigned_min :Z := 0.

  Definition min {n} (a b: BITS n) : BITS n := if lt_unsigned a b then a else
b.
  Definition max {n} (a b: BITS n) : BITS n := if gt_unsigned a b then a else
b.

  Definition widen {s} (result_size: nat) (val: BITS s): BITS (s + (result_size
- s)) := @zeroExtend (result_size - s)%nat s val.

  Definition rolBn {n} (p: BITS (S n)) (k: nat): BITS (S n) := Nat.iter k rolB
p.
  Definition rorBn {n} (p: BITS (S n)) (k: nat): BITS (S n) := Nat.iter k rorB
p.

  Definition subrange_len {skip} (from:nat) (len:nat) (w:BITS ((from + len) +
skip)) :=
    (@high len from (@low skip (ssrnat.addn from len) w)).

  Definition subrange {skip} (from:nat) (to:nat) (w:BITS (from + (to-from) +
skip)) :=
    @subrange_len skip from (to-from) w.

End Operations.
```

```
Declare Scope ZMod_scope.

Infix "+" := (uadd_of) : ZMod_scope.
Infix "-" := (usub_uf) : ZMod_scope.
Infix "×" := (umul) : ZMod_scope.
Infix "<" := (lt_unsigned ) : ZMod_scope.
Infix ">" := (gt_unsigned) : ZMod_scope.
Infix "≤" := (le_unsigned) : ZMod_scope.
Infix "≥" := (ge_unsigned) : ZMod_scope.
(* Equality is already provided by eqType of ssreflect.  *)

Notation "w { from , to }" := (@subrange _ from to w) (at level 10) :
ZMod_scope .

Bind Scope ZMod_scope with BITS.
```