Introduce the ~~zk~~VM

The ~~zk~~VM part for Engineers

github@dyxushuai

What is a VM?

In computing, a **virtual machine** (**VM**) is the virtualization or emulation of a computer system. Virtual machines are based on computer architectures and provide the functionality of a physical computer.

keynotes: Emulate a physical computer

Goal: runing programs like a computer

OS/Application

What is a VM in ZK?

- Same goal, runing programs like a computer.
- Get the execution trace for proving system.

The program lifecycle

- Expressing ideas using high-level programming languages.
- Lowering these expressions into assembly.
- Executing and verifying the result.

Engineers only care about the experience

- Morden high-level language: Rust, Go, C++ etc.
- The mature ecosystem: Ide, Package manager, Building,Testing,Debugging etc.
- A robust and well-developed third-party libraries in work field: Web2, Web3 etc.

ISA: the bridge of program&computer(VM)

In computer science, an **instruction set architecture** (**ISA**) is an abstract model that generally defines how software controls the CPU in a computer or a family of computers.[1] A device or program that executes instructions described by that ISA, such as a central processing unit (CPU), is called an *implementation* of that ISA.

Technology selection: ISA

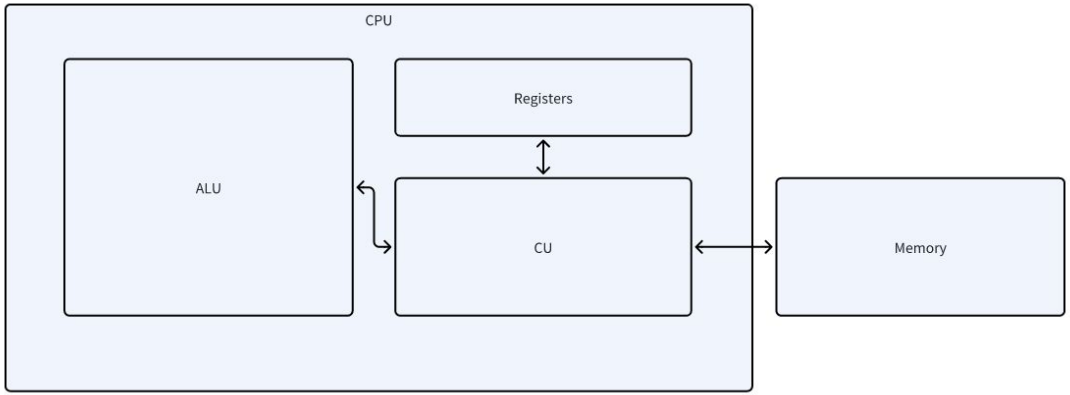
ZK10: Analysis of zkVM Designs - Wei Dai & Terry Chung

- For engineers:
  - Popular and widely supported ISA.
    - We can use any language we prefer.
- For ZK: Performance! Performance! Performance!
  - Hardware acceleration: SIMD and GPU friendly.
    - 32 bit data type for both SIMD and GPU(flops: FP32 >> FP64)
    - Tons of address operations(usize) and memory limitation in zkVM
    - Cache friendly for both CPU and GPU
  - Using small finite field.
    - Less than 32 bit maximum value.
  - The fewer the instructions, always better.

RISCV32/MIPS32 etc.

How to implement?

The computer simple architecture

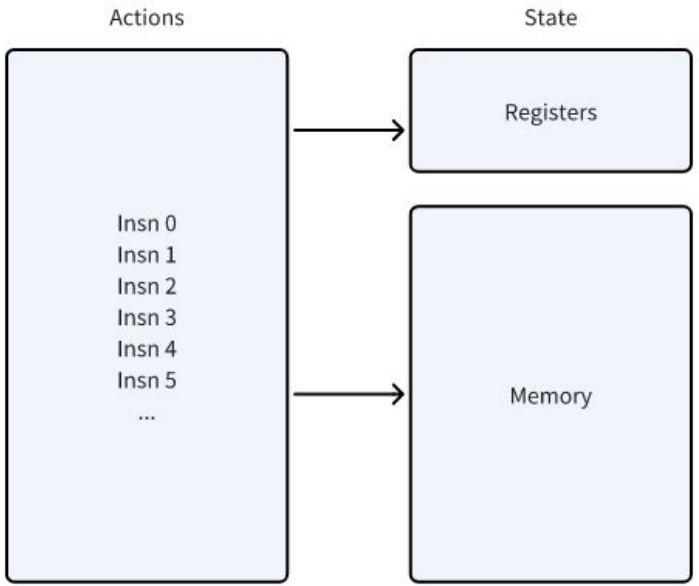


- Control Unit: Fetch/Decode/Dispatch the Instructions
- Arithmetic logic unit: Execute the Instructions
- Register: The data for Instructions
- Memory: The data for Instructions/Programs

We don’t need a “real computer” in the VM

High-level abstraction: Be like a finite-state machine

- State: Memory+Registers
- Actions: Instruction



SP1 source code analysis: State

```rust
pub struct Executor<'a> {
    /// The program.
    pub program: Arc<Program>,
    /// The state of the execution.
    pub state: ExecutionState,
    /// The collected records, split by cpu cycles.
    pub records: Vec<ExecutionRecord>,
}

```

```rust
pub struct ExecutionState {
    /// The program counter.
    pub pc: u32,
    /// The memory which instructions operate over. Values contain the memory value and
    /// + timestamp that each memory address was accessed.
    pub memory: PagedMemory<MemoryRecord>,
    /// The clock increments by 4 (possibly more in syscalls) for each instruction that
    /// executed in this shard.
    pub clk: u32,
    /// A stream of input values (global to the entire program).
    pub input_stream: Vec<Vec<u8>>,
    /// A ptr to the current position in the input stream incremented by `HINT_READ` opc
    pub input_stream_ptr: usize,
    /// A stream of public values from the program (global to entire program).
    pub public_values_stream: Vec<u8>,
    /// A ptr to the current position in the public values stream, incremented when read
    /// `public_values_stream`.
    pub public_values_stream_ptr: usize,
}
```

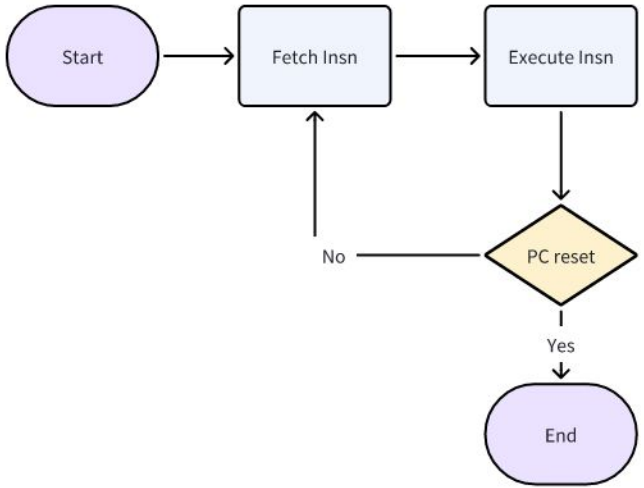
- The ELF format Program
- PC
- Memory/Register
- Clock
- Witness input
- Public value

SP1 source code analysis: Actions

```rust
/// Executes the program, returning whether the program
/// has finished.
pub fn execute(&mut self) -> Result<bool, ExecutionError>;

```

- Fetch and Decode Insn
- Execute Insn
- Check PC == 0



SP1 source code analysis: Actions

```rust
// method execute
loop {
    if self.execute_cycle()? {
        break;
    }
}
```

- Long loop until program exit
- Category for different purposes
- Changing the State

```rust
// method execute_cycle
let instruction = self.fetch();
self.execute_instruction(&instruction)?;
```

```rust

// method execute_instruction
 match instruction.opcode {
      ... => self.execute_alu(...)
      ... => self.execute_load(...)
      ... => self.execute_store(...)
      ... => self.execute_branch(...)
      ... => execute jump
      ... => execute syscall
 }
```

Same design as ZKM

```rust
pub struct State {
    pub memory: Box<Memory>,
    pub registers: [u32; 32],
    pub pc: u32,
    pub input_stream: Vec<Vec<u8>>,
    pub input_stream_ptr: usize,
    pub public_values_stream: Vec<u8>,
    pub public_values_stream_ptr: usize,
}
```

State

```rust
// method split_prog_into_segs
loop {
    if instrumented_state.state.exited {
        break;
    }
    let cycles = instrumented_state.step();
}
```

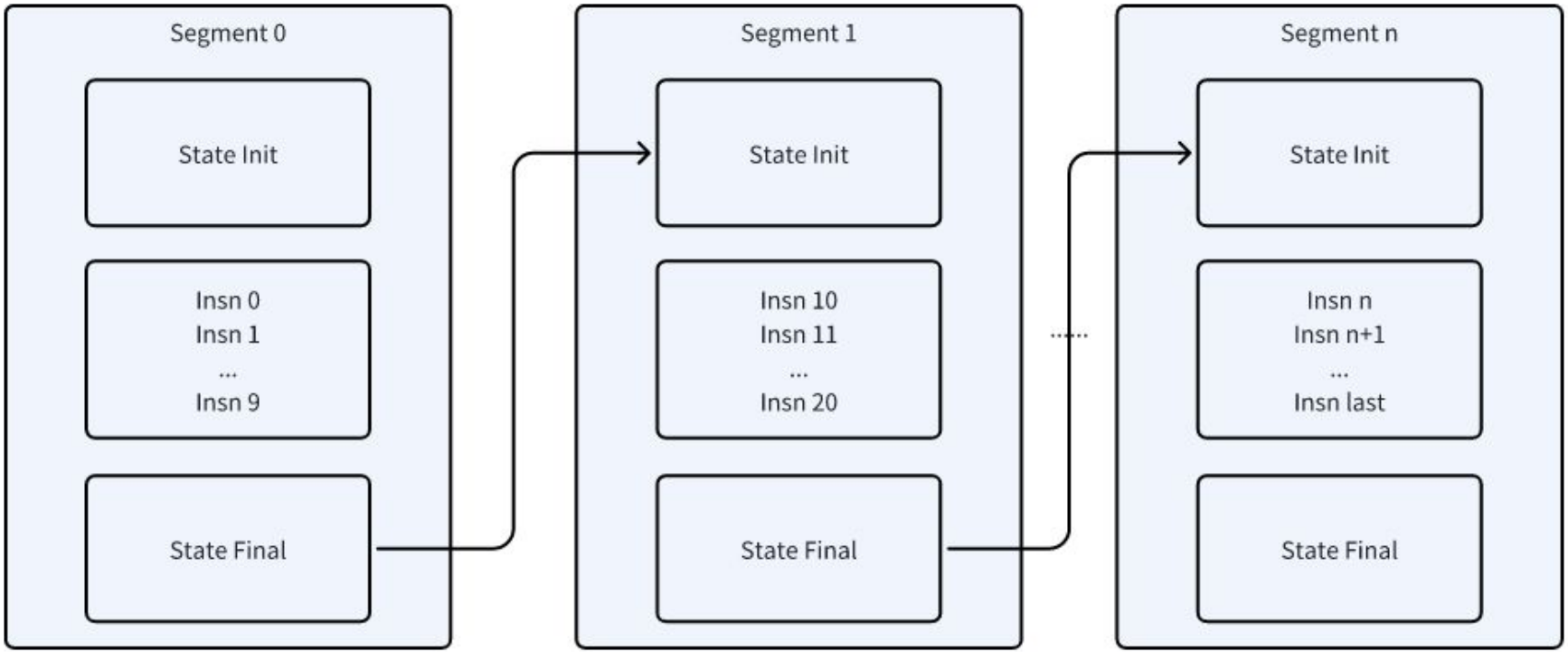
Actions

A deeper dive into

Precompiles: performance boost

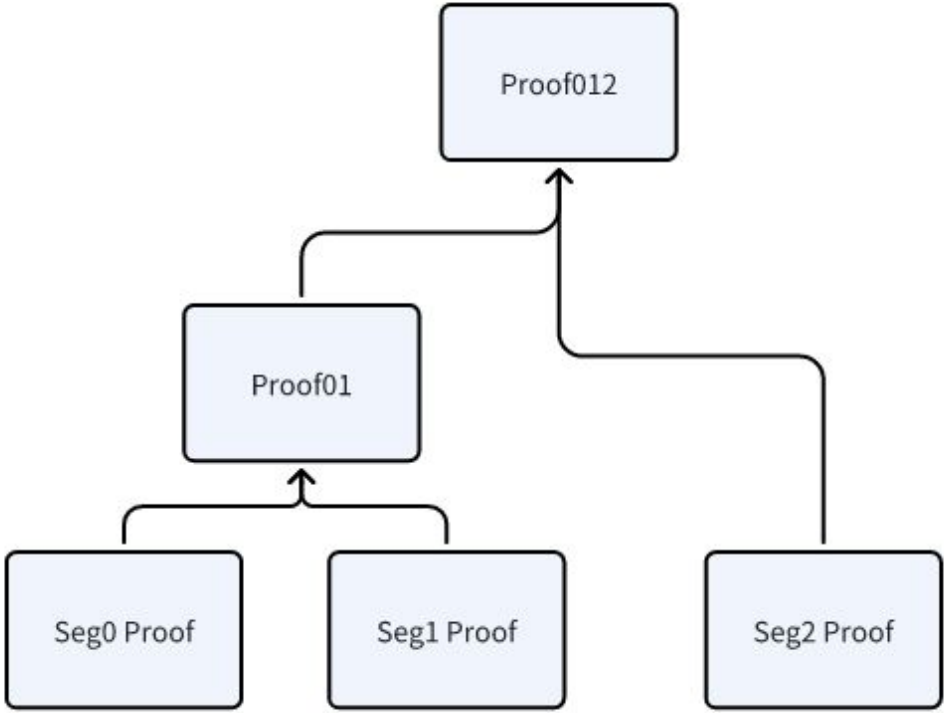
- Precompile universal and stable actions: elliptic curve arithmetic, hashing etc. [1]
- How to?
  - Custom instructions:
    - Pros:
      - Almost language support custom assembly, like `asm!` in Rust.
        - `asm!("myinsn", args)`
      - Less patches for language, only for third-party library you wanted.
    - Cons:
      - Unfriendly to third-party VMs and toolchains.
  - Custom syscalls(ABI):
    - Pros:
      - Without break the spec of ISA.
    - Cons:
      - More patches for language
        - New target tier, like `zkvm` in Rust.
        - New ABI for new tier

Continuation/Segment



Aggregation: Recursive proving

- Verify two proofs recursively each time.



And more …

- IO stuff
  - Witness input/Public values.
  - Standard IO: stdin, stdout, stderr.
    - Debug
    - Trace
    - …
- Memory Pager
  - Improve Merkle proof efficiency.
  - Improve the read/write of VM.

Summary

“Zero-knowledge” for Users

What's next?

Looking Ahead

- zkVM based toolchains: custom ABI(IO)
  - Debugger: lldb, gdb issue
  - Profiler: CPU/Memory profiling
  - …
- Auto precompiles:
  - “Inline Recusive”
    - Find and verify the proof of instruction sequence in custom program. (what the zkm do manually
    - We can even share these general time-consuming algorithm proofs online
  - Circuit Compiler, JIT like
    - Zirgen Circuit Compiler
- Standard VM for zkVMs
  - Standardized syscalls(precompiles).
  - Standardized execution trace.
  - We can share the same patches for the upstream.
  - We can share the zkVM based toolchains.