ZK-WASM: A WASM ZK Emulator

https://jhc.sjtu.edu.cn/~hongfeifu/manuscriptb.pdf
https://github.com/DelphinusLab/zkWasm

XIN GAO, HONGFEI FU, HENG ZHANG, JUNYU ZHANG, and GUOQIANG LI,

Delphinus Lab., Australia and Shanghai Jiao Tong University, China

Introduction

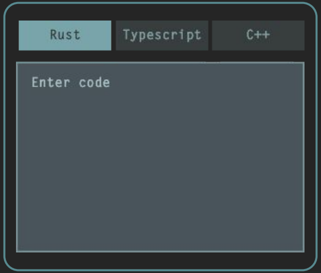
ZK experts:

Write their code in zk arithmetic circuits.

Want to leverage zk but **has limited expertise** in writing zk circuits in their existing or newly created projects.

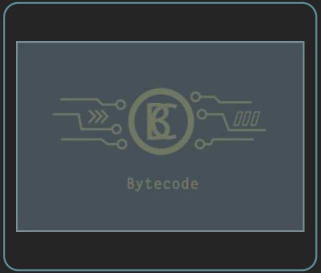
Existing project (written in assembly script, rust, c), **has no technique expertise** but want to leverage zk in their projects

Workflow of using ZKWASM Virtual Machine



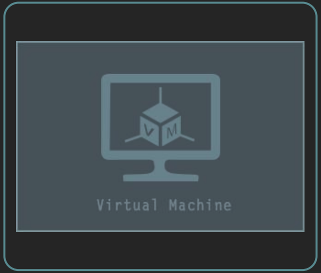
1. Edit: Preparing code

Any language that can be compiled into WASM



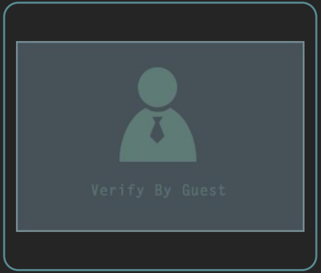
2. Compile: Image Setup

Preparing the Code Image Circuit, Init Memory Circuit, Global Data Circuit, etc



3. Run:

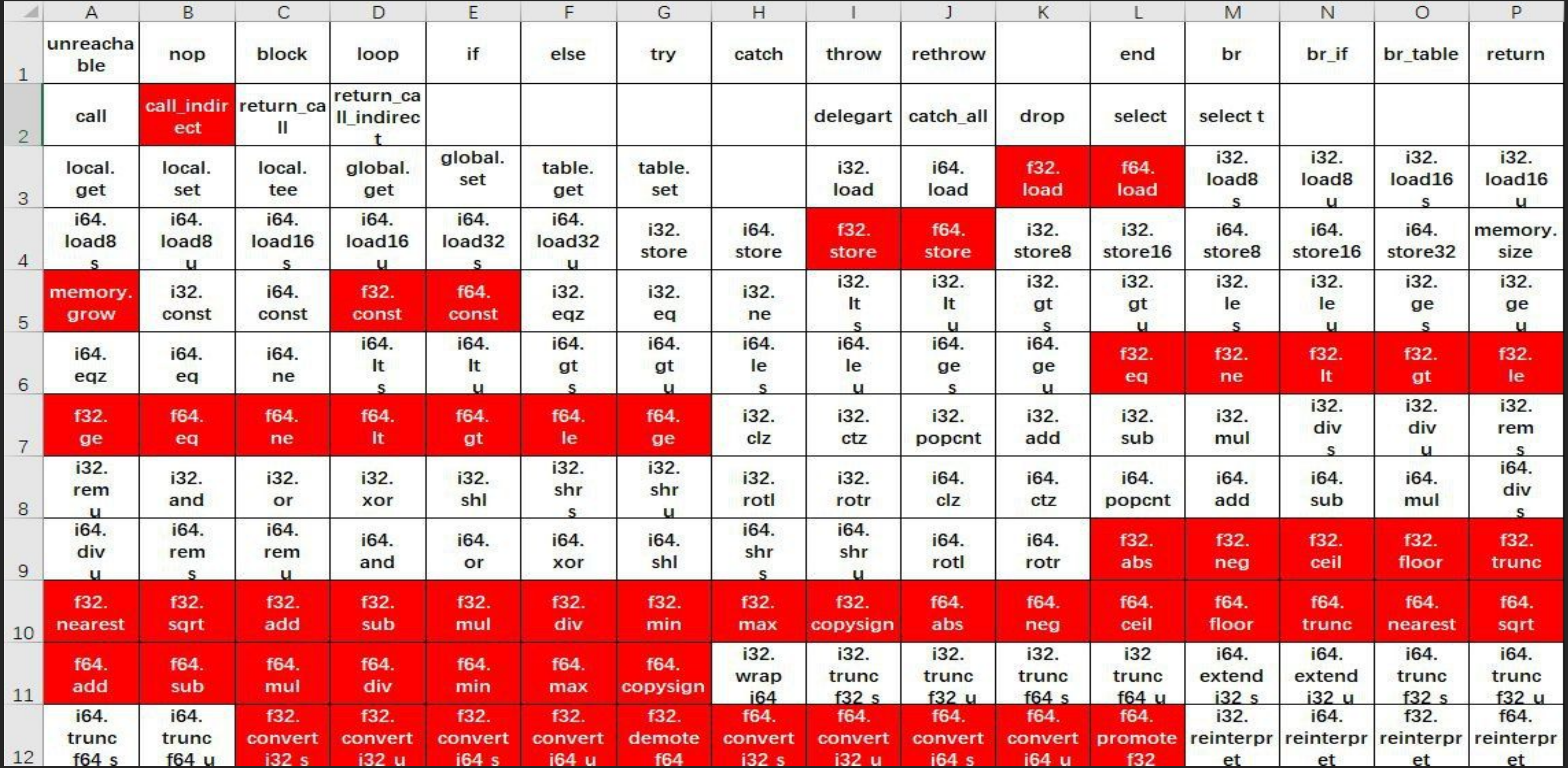
- Execution Trace Generation
- Synthesis Circuits



4. Proof Generation

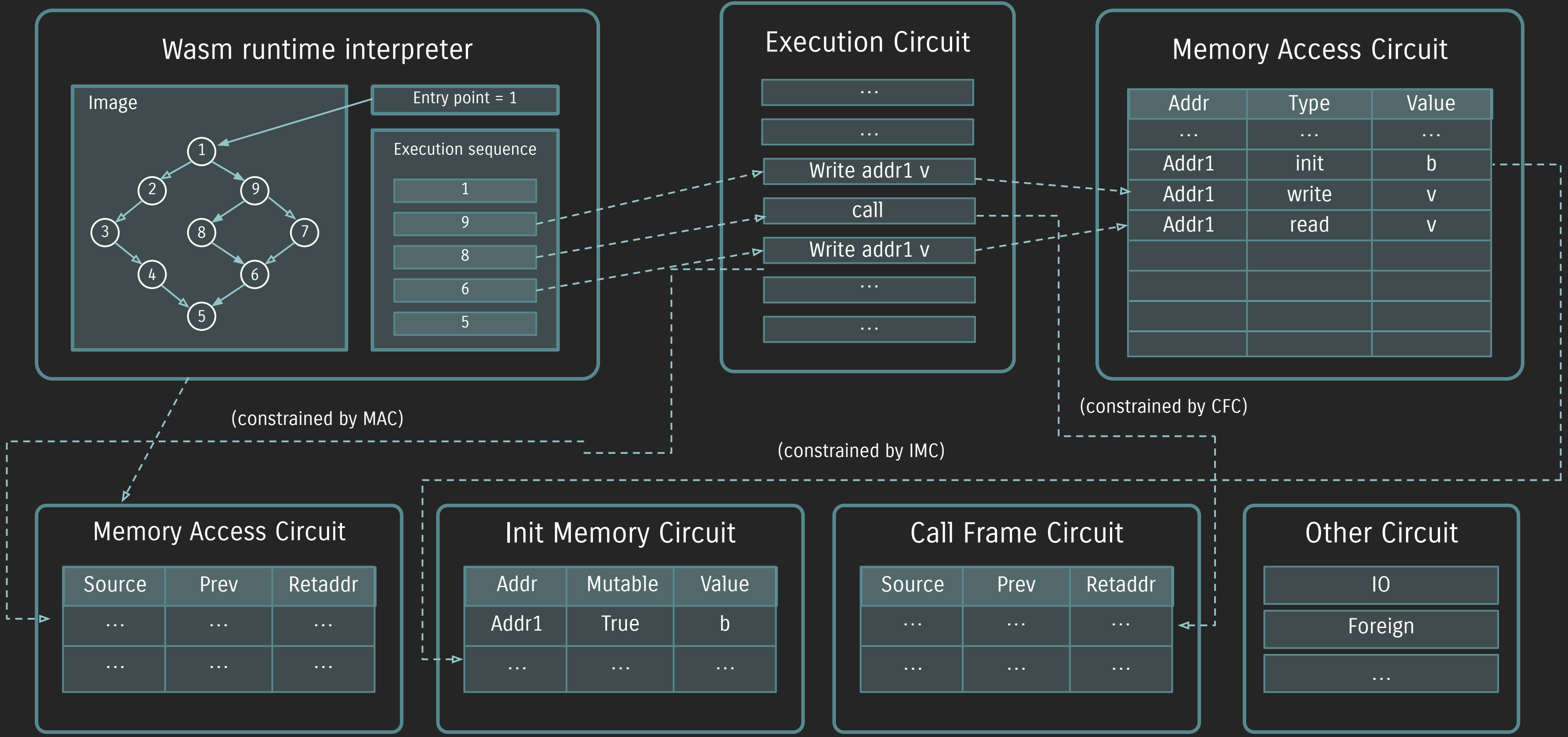
- Execution Partition
- Proof generation and batching

Instruction Set



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	unreachable	nop	block	loop	if	else	try	catch	throw	rethrow		end	br	br_if	br_table	return
2	call	call_indirect	return_call	return_call_indirect					delegart	catch_all	drop	select	select t			
3	local. get	local. set	local. tee	global. get	global. set	table. get	table. set		i32. load	i64. load	f32. load	f64. load	i32. load8 s	i32. load8 u	i32. load16 s	i32. load16 u
4	i64. load8 s	i64. load8 u	i64. load16 s	i64. load16 u	i64. load32 s	i64. load32 u	i32. store	i64. store	f32. store	f64. store	i32. store8	i32. store16	i64. store8	i64. store16	i64. store32	memory. size
5	memory. grow	i32. const	i64. const	f32. const	f64. const	i32. eqz	i32. eq	i32. ne	i32. lt s	i32. lt u	i32. gt s	i32. gt u	i32. le s	i32. le u	i32. ge s	i32. ge u
6	i64. eqz	i64. eq	i64. ne	i64. lt s	i64. lt u	i64. gt s	i64. gt u	i64. le s	i64. le u	i64. ge s	i64. ge u	f32. eq	f32. ne	f32. lt	f32. gt	f32. le
7	f32. ge	f64. eq	f64. ne	f64. lt	f64. gt	f64. le	f64. ge	i32. clz	i32. ctz	i32. popcnt	i32. add	i32. sub	i32. mul	i32. div s	i32. div u	i32. rem s
8	i32. rem u	i32. and	i32. or	i32. xor	i32. shl	i32. shr s	i32. shr u	i32. rotl	i32. rotr	i64. clz	i64. ctz	i64. popcnt	i64. add	i64. sub	i64. mul	i64. div s
9	i64. div u	i64. rem s	i64. rem u	i64. and	i64. or	i64. xor	i64. shl	i64. shr s	i64. shr u	i64. rotl	i64. rotr	f32. abs	f32. neg	f32. ceil	f32. floor	f32. trunc
10	f32. nearest	f32. sqrt	f32. add	f32. sub	f32. mul	f32. div	f32. min	f32. max	f32. copysign	f64. abs	f64. neg	f64. ceil	f64. floor	f64. trunc	f64. nearest	f64. sqrt
11	f64. add	f64. sub	f64. mul	f64. div	f64. min	f64. max	f64. copysign	i32. wrap i64	i32. trunc f32 s	i32. trunc f32 u	i32. trunc f64 s	i32 trunc f64 u	i64. extend i32 s	i64. extend i32 u	i64. trunc f32 s	i64. trunc f32 u
12	i64. trunc f64 s	i64. trunc f64 u	f32. convert i32 s	f32. convert i32 u	f32. convert i64 s	f32. convert i64 u	f32. demote f64	f64. convert i32 s	f64. convert i32 u	f64. convert i64 s	f64. convert i64 u	f64. promote f32	i32. reinterpret	i64. reinterpret	f32. reinterpret	f64. reinterpret

Circuit Architecture



Memory Access Circuit

Addr	Type	Value
...	...	...
Addr1	init	b
Addr1	write	v
Addr1	read	v

Memory Access Circuit

Source	Prev	Retaddr
...	...	...
...	...	...

Init Memory Circuit

Addr	Mutable	Value
Addr1	True	b
...	...	...

Call Frame Circuit

Source	Prev	Retaddr
...	...	...
...	...	...

Setup Circuit

(Code Image Circuit)

Setup circuits are filled by the ZKWASM compiler component and its purpose is to provide lookup tables T_C, T_H, T_G that encode code section, initial memory & global and module section.

Code Section. Instructions are grouped in a tree like hierarchy. Each instruction can be indexed by *moid* (modular id), *mmid* (memory block instance id), *fid* (function id) and *iid* (offset of the instruction in a particular function).

Data Section: Stores initial memory, and can be represented as a map from (*mmid*,*offset*) to (*value*,*isMutable*).

Global Data & Import Info: Information about import tree.

Code Table

moid	mmid	fid	iid	Opcode

Initial Memory Table

ltype	mmid	offset	value	isMustable
Heap	mmid$_0$	1	0x01	true
Heap	mmid$_1$	1	0x01	true
Global	mmid$_2$	1	0x01	true
Global	mmid$_3$	1	0x01	false

Import Table

root	target	source

Execution Circuit (Overview)

Execution Trace Circuits are used to constraint the execution trace emulated from WASMI (WASM interpreter). Each trace element is related to an instruction in the code table and has a predefined semantic based on the opcode. The semantics of a WASM opcode is defined based on its parameters derived from the stack and the micro operations it contains.

Execution Table

start	opcode	bit cell	state	aux	address	sp $\in T_F$	u64cell
ture	op	b_0	s_0	aux	$iaddr_0$	sp	w_0
0	mop_0	b_1	s_1	aux_0	$iaddr_0$	sp	w_1
0	mop_1	b_2	s_2	aux_1	$iaddr_1$	sp	w_2
0	mop_2	b_3	s_3	aux_2	$iaddr_2$	sp	w_3
…	…	…	…	…	…	…	…
ture	op_1	b	s	aux	$iaddr_1$	sp	w_0
…	…	…	…	…	…	…	…

$$mop1 = \begin{cases} w_i = load(ltype,addr) \text{ where } addr \in \{p_1,p_2,\cdots,p_k,w_0,w_1,\cdots,w_{i-1}\} \\ write(ltype,addr,v) \text{ where } addr,v \in \{p_1,p_2,\cdots,p_k,w_0,w_1,\cdots,w_{i-1}\} \\ w = arith(p,p,\cdots,p,w,w); \\ FALLTHOUGH; \\ GOTO(iaddr); \\ if\ b then\{mop_{i+1},\cdots mop_j\}else\{mop_{j+1},\cdots\}. \end{cases}$$

Instruction Circuit (numeric)

Numeric Instructions are the majority of instructions in WASM. In general, the semantics of numeric instructions contain three parts, parameters preparation, arithmetic calculation, result writeback and FALLTHROUGH.

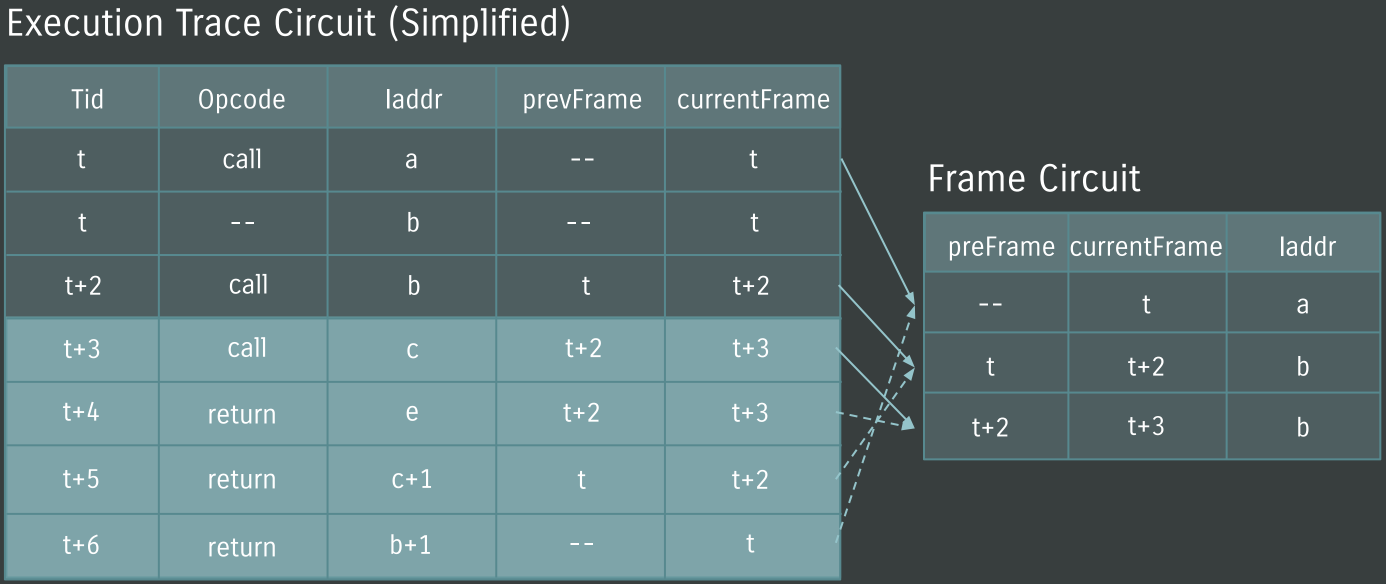
Add Circuit In Execution Trace Table

start	opcode	bit cell	state	aux	address $\in$ *TI*	sp $\in$ *TF*	u64cell	extra
ture	arithop	nil	tid	nil	$iaddr_0$	sp	$param_0$	nil
0	nil	nil	nil	nil	nil	nil	…	nil
0	nil	nil	nil	nil	nil	nil	$param_n$	nil
0	nil	nil	nil	nil	nil	nil	result	nil
ture	otherop	–	tid+1	nil	$iaddr_1$	sp’	w'_0	nil

$$\text{Carith} = \begin{cases} arith(param\ ,\ param\ ,\ ...,\ param\ ) - result = 0 \\ Plookup(T\ ,(stack,read,sp-k,tid,k,param\ )=0 \qquad Plookup(TM,(stack, \\ write,sp',tid,N,result))=0 \\ iaddr0+1-iaddr1=0 \\ sp-sp'-N+1=0 \end{cases}$$

Call frame circuit

```
call a {
  instructions;
  call b {
     instructions;
     call c {
          instructions;
          return;
     }
     return;
  }
  return;
}
```



Execution Trace Circuit (Simplified)

Tid	Opcode	Iaddr	prevFrame	currentFrame
t	call	a	--	t
t	--	b	--	t
t+2	call	b	t	t+2
t+3	call	c	t+2	t+3
t+4	return	e	t+2	t+3
t+5	return	c+1	t	t+2
t+6	return	b+1	--	t

Frame Circuit

preFrame	currentFrame	Iaddr
--	t	a
t	t+2	b
t+2	t+3	b

Instruction Circuit (call)

Call (Return) Circuit. Call instruction will first add a new frame table entry $(tid, frameId, iaddr_0)$ into the frame circuits T_F and then load calling parameters onto the stack and go to the $iaddr_1$ for next instruction.

Circuit Layout Of Call

start	opcode	bit cell	state	aux	address $\in T_I$	sp $\in T_F$	u64cell	extra
ture	call(targetIaddr)	nil	tid	nil	$iaddr_0$	sp	$param_0$	nil
0	nil	nil	pFrameId	nil	nil	nil	···	nil
0	nil	nil	nil	nil	nil	nil	$param_N$	nil
ture	otherop	–	tid+1	nil	$iaddr_1$	sp’	nil	nil
0	nil	nil	nFrameId	nil	nil	nil	nil	nil

$$C_{call} = \begin{cases} Plookup(T\ ,(stack,write,sp+i,tid,i,param))=0 \\ Plookup(T\ ,(tid,pFrameId,iaddr\ ))=0 \\ iaddr_1 - targetIaddr = 0 \\ sp'-sp-N=0 \\ nFrameId-tid=0. \end{cases}$$

Instruction Circuit (return)

- We need to find the correct return address of the current frame and set the frame state to the previous frame.
- The entries in TF are used to help the return instruction to find the correct calling frame and previous frame.

Circuit Layout Of Return

star t	opcode	bit cell	state	aux	address $\in TI$	sp $\in T$F	u64cell	extra
ture	return	nil	tid	nil	$iaddr_0$	sp	nil	nil
0	nil	nil	pFrameId	nil	nil	nil	nil	nil
tur e	otherop	–	tid+1	nil	$iaddr_1$	sp’	nil	nil
0	nil	nil	nFrameId	nil	nil	nil	nil	nil

$$C_{return} = \begin{cases} Plookup(T_F, (pFrameId, nFrameId, iaddr_1 - 1)) = 0 \\ sp' - sp = 0 \end{cases}$$

Execution Circuit (branch)

```
def branchop :=
 \\ parameters preparation
 param1 = read(stack sp);
 param2 = read(stack (sp-1));
  ...
 paramN = read(stack (sp-N+1));

 \\ calculate branch address
 iaddr1 = select(param1, param2,
 ..., paramN);

 \\branch to target address
 GOTO iaddr2;
```

Circuit Layout Of Branch

start	opcode	bit cell	state	aux	address $\in \mathcal{TI}$	sp $\in \mathcal{TF}$	u64cell	extra
ture	branchop	nil	tid	nil	$iaddr_0$	sp	$param_0$	nil
0	nil	nil	pFrameId	nil	nil	nil	$\cdots$	nil
0	nil	nil	nil	nil	nil	nil	$param_N$	nil
ture	otherop	–	tid+1	nil	$iaddr_1$	sp'	nil	nil
0	nil	nil	nFrameId	nil	nil	nil	nil	nil

$$\mathcal{C}branch = \begin{cases} Plookup(TM, (stack, write, sp+i, tid, i, param)) = 0 \\ \mathcal{C}_{branch} = iaddr_1 - select\ (param_0, param_1, \cdots) = 0 \\ nFrameId - pFrameId = 0. \end{cases}$$

Access Log Circuit

Access log circuit is a unique table corresponding to a valid memory access log sequence. In WASM specification, an access log is used for three different types that are memory access, stack access and global access.

Each access log can be one of the three types: Init, Read or Write and all logs are sorted by ($addres$, (tid, $tmid$)) where $address$ is indexed by ($mmid$,$offset$), tid is the transition index of the execution log and $tmid$ is the index of the access micro-op in that instruction.

Access Log Circuit

section	ltype	mmid	offset	tid	emid	value
Memory | stack | global	Init | Read | Write					

Foreign Instruction

Inline if semantic is simple

External circuits if semantic is complicated but pure

Row Reduce By Using Customized Instructions

original	original rows	customized	Optimized rows
$\lambda x,y,\ (x\&y)\|(complete(x)\&z)$	1	$ch(x,y)$	1
$\lambda x,y,z,z\|(x\&(y\|z)$	1	$maj(x,y,z)$	1
$\lambda x,rotr32(x,2)\|rotr32(x,13)\|rotr32(x,22)$	1	$lsigma0(x)$	1
$\lambda x,rotr32(x,6)\|rotr32(x,11)\|rotr32(x,25)$	1	$lsigma1(x)$	1

Program Sharding and Batching (Sub Sequence)

Execution Trace Circuit (Whole Trace)

Tid	Opcode	Iaddr	State	Aux	Address	···
t	op1	a	--	--	--	
t+1	--	b	--	--	--	
t+2		c	--	--	--	
t+3		d				
t+4		e				
t+5		f				
t+6		g				

Sub Trace 1

Tid	Opcode	Iaddr	State	Aux	Address	
t	op1	a	--	--	--	
t+1	op2	b	--	--	--	
t+2	op3	c	--	--	--	
t+3	pad	d	--	--	--	

Sub Trace 2

Start	Opcode	Iaddr	State	Aux	Address	
t+2	pad	c	--	--	--	
t+3	op4	d	--	--	--	
t+4	op5	e	--			

Pad op

Program Sharding and Batching (Memory Consistency)

Memory Access Circuit (Whole Trace)

Iaddr	Type	Tid	Mid	Value
Addr1	Init	a	--	V1
Addr1	Read	b	--	V1
Addr1	Write	c	--	V2
Addr1	Read	d	--	V2
Addr1	Read	e	--	V2
Addr2	Init	f	--	V2

Sub Access Circuit 1

Iaddr	Type	Tid	Mid	Value
Addr1	Init	a	--	V1
Addr1	Read	b	--	V1
Addr1	Write	c	--	V2

Sub Access Circuit 2

Iaddr	Type	Tid	Mid	Value
Addr1	Read	d	--	V2
Addr1	Read	e	--	V2
Addr2	Init	f	--	V2

Access Glue Circuit

Iaddr	Type	Tid	Mid	Value
Addr1	Write	c	--	V2
Addr1	Read	d	--	V2
--	--	--	--	--

Performance Benchmark

Benchmark For Fibonacci In ZKWASM

circuit size	trace size	call depth	proof time	verify time	Memory swap
18	9037	13	44.200s	22 ms	false
19	23677	15	87.200s	24 ms	false
20	38317	16	173.200s	22 ms	false
21	100333	18	342.200s	24 ms	false
22	162349	19	803.200s	25 ms	true

Benchmark For Binary Search In ZKWASM

circuit size	trace size	search buffer	proof time	verify time	Memory swap
18	585	26	44.800s	22 ms	false
19	616	63	88.200s	24 ms	false
20	647	124	173s	22 ms	false
21	678	246	342	24 ms	false
22	809	490	803	25 ms	true

- Fibonacci with different call stack depth
- Binary search with different buf size（*64k per page)
- Memory access table is too expensive in circuit size.
- Memory consumption is high at the moment.

Performance Benchmark of batching

- Batching circuit enforces the memory consistency and code execution consistency
- Batching circuit contains the verify circuit for multiple proofs
- Batching circuits size increases as the number of proofs of partitions increases
- Both partition proofs and batching proofs can be generated in parallel

Benchmark For Proof Batching In ZKWASM

Partition CS	Partition proof	Total pieces	Batching CS	Batching proof	Verify time	Memory swap
19	2	1	22	168s	22 ms	false
19	2	2	22	325s	24 ms	false
20	2	1	22	325s	22 ms	false
22	2	1	22	168s	24 ms	false
22	2	2	22	325s	25 ms	true

Thank you

Delphinus Lab

Use Cases (1)

UNIPASS

User can run a program to generate a proof to show that he(she) is the owner of certain email address and use that proof too reset the proxy contract

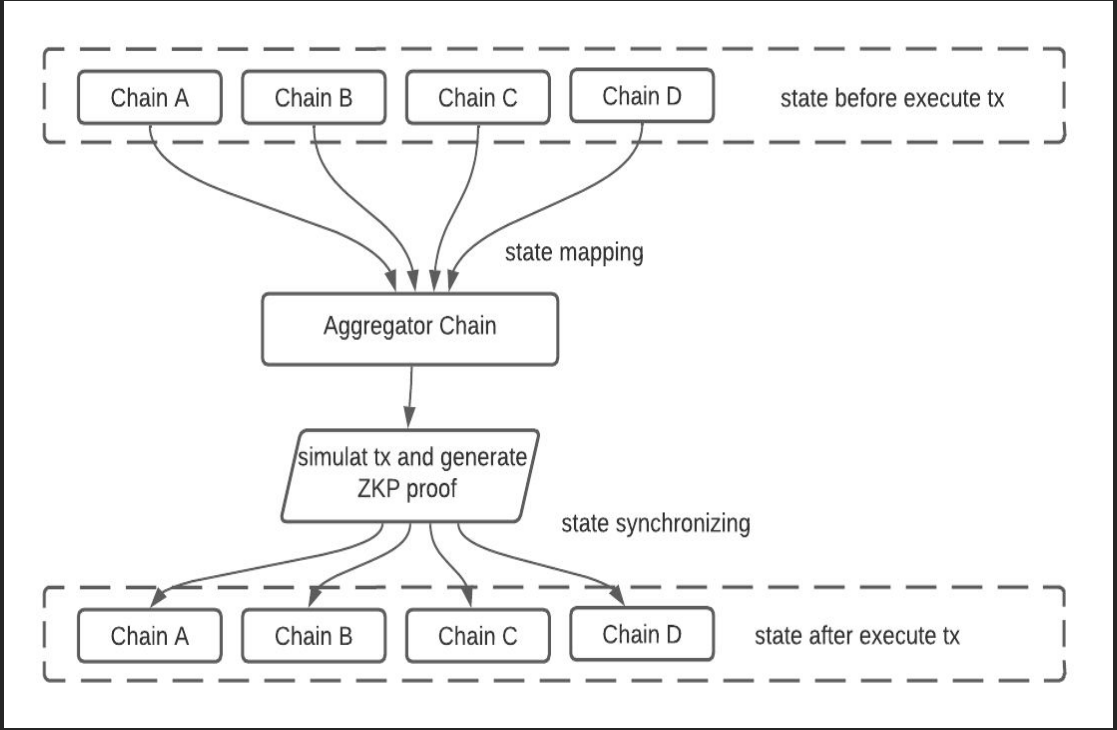
(5 minutes prove and batching time in total)

```
Private: {
        "emailHeader": "0x66726f6 … 623d" (around 4k),
        "fromPepper": “0x0316f3 … c846f4",
        "fromIndex": 0,
        "fromLeftIndex": 32,
        "fromRightIndex": 50,
        "subjectIndex": 92,
        "subjectRightIndex": 168,
        "dkimHeaderIndex": 390,
        "selectorIndex": 460,
        "selectorRightIndex": 469,
        "sdidIndex": 445,
        "sdidRightIndex": 456
    }
Public: {
        "headerHash": "0x0f1b318a8…3877bce",
        "fromHash": "0x15aa0b4….caa1c782",
        "Subject": "UP0xe6339….146e72",
        "selector": "selector1",
        "sdid": "outlook.com"
    }
```

Use Cases (2)

ZKCROSS

Simulating transactions in their sidechain in wasm bytecode and broadcasting the result together with its zkproof to native chains to do settlement.


Chain A
Chain B
Chain C
Chain D
state before execute tx
state mapping
Aggregator Chain
simulat tx and generate ZKP proof
state synchronizing
Chain A
Chain B
Chain C
Chain D
state after execute tx