opcode	imm.	stack before	stack after	mv	$\perp$	comment
`dup`	u64:n k	$a_1..a_n$	$a_1..a_n$ $a_1..a_k$			duplicate values on the stack, $k \le n$
`pop`	u64:n	$a_1..a_n$				pop n entries off the stack
`rot`	u64:n k	$a_1..a_n$	$a_{k+1}..a_n$ $a_1..a_k$			rotate n entries on stack left by $k \le n$
`tpack`		$a_1..a_n$ i64:n	$(a_1..a_n)$			pack n elements from stack into a new tuple
`texpl`		$(a_1..a_n)$	$a_1..a_n$ i64:n			unpack tuple onto stack
`tsplt`		$(a_1..a_n)$ i64:k	$(a_1..a_k)$ $(a_{k+1}..a_n)$			split tuple at index $k \in \{0, \dots, n\}$
`tcat`		$(a_1..a_n)$ $(b_1..b_m)$	$(a_1..a_n, b_1..b_m)$			concatenate tuples
`tlen`		$(a_1..a_n)$	$(a_1..a_n)$ i64:n			extract length of tuple
`apush`	adr:f		f			push immediate address to stack
`dcall`		$a_1..a_n$ adr:f	$b_1..b_m$			call dyn. $f(a_1, \dots, a_n) = (b_1, \dots, b_m)$, n, m unspec.
`scall`	adr:f	$a_1..a_n$	$b_1..b_m$			call imm. $f(a_1, \dots, a_n) = (b_1, \dots, b_m)$, n, m unspec.
`ret`						return from function call or exit
`jmp`	adr:l					unconditional jump to l
`jnz`	adr:l	i64:v	v			conditional jump to l if $v \neq 0$
`or`		i64:v i64:w	i64:$x \in \{0,1\}$			boolean disjunction $x = 0 \iff v = 0 = w$
`and`		i64:v i64:w	i64:$x \in \{0,1\}$			boolean conjunction $x = 1 \iff v \neq 0 \neq w$
`not`		i64:v	i64:$x \in \{0,1\}$			boolean negation $x = 1 \iff v = 0$
`ipush`	i64:v		i64:v			push imm. 64-bit two's complement v to stack
`ineg`		i64:v	i64:$(-v)$			64-bit two's complement negation
`iadd`		i64:v i64:w	i64:$(v + w)$			64-bit binary addition with overflow
`imul`		i64:v i64:w	i64:$(v \cdot w)$			64-bit two's complement multiplication w/ overflow
`idiv`		i64:v i64:w	i64:(v/w)		y	64-bit two's complement division with truncation
`isgn`		i64:v	i64:$(\operatorname{sgn} v)$			sign of 64-bit two's complement int
`zconv`		i64:v	Z:v			convert 64-bit two's complement int to integer
`zneg`		Z:v	Z:$(-v)$			integer negation
`zadd`		Z:v Z:w	Z:$(v + w)$			integer addition
`zmul`		Z:v Z:w	Z:$(v \cdot w)$			integer multiplication
`zdiv`		Z:v Z:w	Z:(v/w)		y	integer division with truncation
`zsgn`		Z:v	i64:$(\operatorname{sgn} v)$			sign of integer
`zsh`		Z:v i64:n	Z:$(v \cdot 2^n)$			integer multiplication by 2^n with truncation
`rconv`		Z:v	R:v			convert an integer to real
`rneg`		R:v	R:$(-v)$			real negation
`radd`		R:v R:w	R:$(v + w)$			real addition
`rinv`		R:v	R:$(1/v)$		y	real inversion
`rmul`		R:v R:w	R:$(v \cdot w)$			real multiplication
`rsh`		R:v i64:w	R:$(v \cdot 2^w)$			multiplication by 2^w
`rin`			$(a_1..a_n)$			get real input, n unspec.
~~`rlim`~~	~~adr:f~~	~~$a_1..a_n$~~	~~$b_0..b_m$~~	~~?~~	~~y~~	~~$\lim_{p \to \infty} (f(-p, a_1..a_n))_p = (b_0..b_m)$, n, m unspec.~~
`rch`		R:$r_1..r_n$ i64:n	i64:k	y	y	mv-choice, $\{0\}$ if $\forall i : r_i < 0$, $\{i : r_i > 0\}$ otherwise
`rapx`		R:x i64:p	Z:m i64:e	y	y	approx. to abs. prec.: $\lvert x - m \cdot 2^{e - \lceil \log_2(\lvert m \rvert + 1) \rceil} \rvert < 2^p$ if $\log_2(\lvert x \rvert + 1) < 2^{63}$
~~`rilog`~~		~~R:x i64:p~~	~~i64:k~~	~~y~~		~~approx. integer logarithm $\{k : 2^k \le \lvert x \rvert + 2^p < 2^{k+2}\}$~~
`entc`			Z:$\tilde{p}$			enter continuous section with (volatile) prec. $\tilde{p}$
`lvc`	u64:n	Z:$\tilde{p}$ $a'_1..a'_n$	$a_1..a_n$		y	leave continuous section (last $\implies \tau(a_i) \neq \mathrm{R}$)

Figure 1: Instruction set of the low-level language.

Let $\tau \coloneqq \{\mathrm{i64}, \mathrm{adr}, \mathrm{Z}, \mathrm{K}, \mathrm{R}\}$ and for $t \in \tau$ let

$$\operatorname{dom} t \coloneqq \begin{cases} \{-2^{63}, \ldots, 2^{63}-1\} \subseteq \mathbb{Z} & \text{if } t = \mathrm{i64} \\ \mathbb{Z}_{2^{64}} & \text{if } t = \mathrm{adr} \\ \mathbb{Z} & \text{if } t = \mathrm{Z} \\ \mathbb{K} & \text{if } t = \mathrm{K} \\ \mathbb{R} & \text{if } t = \mathrm{R} \end{cases}$$

and let $\operatorname{top} t$ be the discrete topology on $\operatorname{dom} t$ for $t \in \{\mathrm{i64}, \mathrm{adr}, \mathrm{Z}\}$, the topology $\{\varnothing, \{0\}, \{1\}, \{0,1\}, \{0,1,\bot\}\}$ of the lifted booleans $\operatorname{dom} t$ for $t = \mathrm{K}$ and the standard topology on the real line $\operatorname{dom} t$ for $t = \mathrm{R}$. Note that for all $t \in \tau$, $(\operatorname{dom} t, \operatorname{top} t)$ are complete and for $t \neq \mathrm{K}$ these are also metric spaces. In the following $d : (\operatorname{dom} t)^2 \to \mathbb{R}$ denotes the respective metric if it exists. For finite sequences $s \in \tau^*$ let $\operatorname{dom} s \coloneqq \bigtimes_{i=1}^{|s|} \operatorname{dom} s_i$ be the product space of $(\operatorname{dom} s_i)_i$, $\operatorname{top} s$ be its product topology and if all $\operatorname{dom} s_i$ are metric spaces, let also $d : (\operatorname{dom} s)^2 \to \mathbb{R}$ denote the metric induced by $\|\cdot\|_\infty$ on $\operatorname{dom} s$.

Let $s \in (\tau \setminus \mathrm{K})^*$ and $\tilde{p} \in \mathbb{Z}$ and $x, x' \in \operatorname{dom} s$. Then x' is a *$\tilde{p}$-approximation* of x if $d(x, x') \leq 2^{\tilde{p}}$.

Let $\mathcal{T} = \bigcup_{t \in \tau} \operatorname{dom} t$ and let p be a *program*, that is, a finite word over the set of instructions from fig. 1 of length $n \leq 2^{64}$. We call (c, v, s, r) a *configuration of p* where $c \in \mathbb{Z}_{2^{64}}$ with $c < n$ is the *program counter*, $v \in \mathcal{T}^*$ is the *value stack*, $s \in (\mathbb{Z}_{2^{64}})^*$ is the *continuous section stack* and $r \in (\mathbb{Z}_{2^{64}})^*$ is the *return stack*. For any *program* p (that is,) of length n and any $1 \leq i \leq n$, $(i, \epsilon, \epsilon, \epsilon)$ is an *initial configuration of p*, where ϵ denotes the empty word.

We will now give the context in which fig. 1 defines the transition relation $\vdash$ on configurations of p for a program p. Let (c, v, s, r) be a configuration of p and let $I = p_c$. Then $(c, v, s, r) \vdash (c', v', s', r')$ iff

-

If code inside a continuous section enclosed by the pair of instructions ($\texttt{entc}, \texttt{lvc}\ n$) computes a $\tilde{p}$-approximation $(a'_1, \ldots, a'_n)$ of $(a_1, \ldots, a_n)$, then the continous section computes $(a_1, \ldots, a_n)$.

How to implement `main()` depends on what we want to express by our program. Should it compute $f : \mathbb{R} \to \mathbb{R}$, $g : \mathbb{Z} \times \mathbb{R} \to \mathbb{Q}$ or $h : \mathbb{Z} \times \mathbb{Q} \to \mathbb{Q}$?

```
#include <iRRAM/lib.h>
/* define input() and
 * output() via kirk */

using namespace iRRAM;
int main() {
  iRRAM_init();
  exec([]{

    REAL x = input();
    REAL y = sqrt(x);
    output(y);

  }); }
```

(a) $f : \mathbb{R} \to \mathbb{R}$

```
#include <iRRAM/lib.h>
/* define input()
   using kirk */

using namespace iRRAM;
int main() {
  iRRAM_init();
  exec([]{
    int n; cin >> n;
    REAL x = input();
    REAL y = sqrt(x);
    cout << setRwidth(n)
         << y;
  }); }
```

(b) $g : \mathbb{Z} \times \mathbb{R} \to \mathbb{Q}$

```
#include <iRRAM/lib.h>



using namespace iRRAM;
int main() {
  iRRAM_init();
  exec([]{
    int  n; cin >> n;
    REAL x; cin >> x;
    REAL y = sqrt(x);
    cout << setRwidth(n)
         << y;
  }); }
```

(c) $h : \mathbb{Z} \times \mathbb{Q} \to \mathbb{Q}$

Figure 2: Implementations of square root with different composeability.

The only line common to the continous parts of the algorithms is `REAL y = sqrt(x);`, which is exactly the composeable part of these algorithms.

Actually, the kirk-versions are cheated, it might look like fig. 3.

```
#include <kirk/kirk-irram.hh>

extern "C" void sqrt(kirk_real_t **in, int n_in, kirk_real_t **out) {
  iRRAM_init();
  using namespace iRRAM;
  assert(n_in == 1);
  auto machine = kirk::irram::eval(in, n_in, out, 1,
    [](const REAL *in, REAL *out){
      const REAL &x = in[0];
      REAL &y = out[0];
      y = iRRAM::sqrt(x);
    });
  /* can't make use of the machine, yet, forget it */
}

/* what should main() do? */
```

Figure 3: Library-like implementation of $f : \mathbb{R} \to \mathbb{R}$.

What should `main()` do? The point being, in the discrete setting of `main()`, “executing” a function on continuous data does not make sense using a model like oracle machines. Only as a transformation of a stream of approximations. Therefore, there are two options.

1. Implement algorithms on continuous data not in terms of `main()` but as library functions that operate on (e.g. kirk-provided) function pointers as in fig. 3.

2. Transform a stream of approximations from `stdin` to `stdout`, an example is provided in fig. 4.

```
#include <kirk/kirk-c-types.h>
int main() {
  kirk_real_t *x[] = { kirk_real_from_file(stdin) };
  kirk_real_t *y[1];
  sqrt(x, 1, y);
  kirk_real_to_file(y[0], stdout); /* returns only when stream errors */
}
```

Figure 4: Stream-like implementation of $f : \mathbb{R} \to \mathbb{R}$.

It does not seem as if a program like fig. 4 in general would be of much use.

With respect to composeability, it is my impression that a design like $g : \mathbb{Z} \times \mathbb{R} \to \mathbb{Q}$ or $h : \mathbb{Z} \times \mathbb{Q} \to \mathbb{Q}$ is not the right choice for the language. Therefore, programs in this language are meant to be library-like, i.e. for the stack-based variant of the low-level language this would mean an initial configuration where there already are real numbers (type R) on the stack and when the program returns, it leaves zero or more objects of type R on the stack.

Programs that expect this kind of input/output have to be executed in a continuous section or a limit respectively and they are library-like functions, that is, not `main()`.