QA—Register Transfer Language

Andrew Pochinsky

Version 1.4.0
$Id: qa0.nw 572 2008-05-03 19:34:07Z avp $

1 GRAMMAR

Here we define the grammar for QA. We start with the file level constructs:

⟨*qa0*⟩ → ⟨*decl*⟩*
⟨*decl*⟩ → ⟨*alias*⟩
| ⟨*constant*⟩
| ⟨*structure*⟩
| ⟨*array*⟩
| ⟨*verbose*⟩
| ⟨*include*⟩
| ⟨*macro definition*⟩
| ⟨*top level repeat*⟩
⟨*top level repeat*⟩ → ⟨*procedure*⟩
| `( repeat (` ⟨*iterator*⟩* `)` ⟨*top level repeat*⟩+ `)`

The ⟨*macro defintion*⟩ directive defines a hygenic macro.

⟨*macro definition*⟩ → `(define (` ⟨*name*⟩ ⟨*name*⟩* `)` ⟨*code*⟩+ `)`

The ⟨*include*⟩ directive inserts the contents of the file into the input.

⟨*include*⟩ → `(include` ⟨*file*⟩ `)`
⟨*file*⟩ → ⟨*string*⟩

The ⟨*verbose*⟩ directive allows one to include literals into various outputs:

⟨*verbose*⟩ → `(verbose` ⟨*verbose case*⟩+ `)`
⟨*verbose case*⟩ → `(` ⟨*target*⟩ ⟨*verbose value*⟩ `)`
⟨*target*⟩ → ⟨*symbol*⟩
⟨*verbose data*⟩ → ⟨*string*⟩

The ⟨*target*⟩ selects the back-end that will see the corresponding ⟨*vebose data*⟩.
There are two kinds of iterators. One is an enumeration, another is loop-like

⟨*iterator*⟩ → `(` ⟨*name*⟩ `(` ⟨*a-value*⟩+ `)` `)`
| `(` ⟨*name*⟩ ⟨*inclusive constant low*⟩ ⟨*exclusive constant high*⟩ `)`
⟨*inclusive constant low*⟩ → ⟨*constant expr*⟩
⟨*exclusive constant high*⟩ → ⟨*constant expr*⟩

Integer constants may be defined and given names. We do not specify the allowed set of operations in ⟨*c-expr*⟩ here, they will be introduced as needed.

⟨*constant*⟩ → `(constant` ⟨*name*⟩ ⟨*constant expression*⟩`)`
⟨*constant expression*⟩ → `(const` ⟨*c-expr*⟩ `)`
⟨*c-expr*⟩ → ⟨*number*⟩
| ⟨*name*⟩
| ⟨*string*⟩
| `(` ⟨*c-op*⟩ ⟨*c-expr*⟩* `)`
⟨*c-op*⟩ → ⟨*symbol*⟩
⟨*name*⟩ → ⟨*symbol*⟩

Structure definitions provide a way to compute offsets to various elements within a memory block. They also contain enough information to generate a corresponding C declaration.

⟨*structure*⟩ → `(structure` ⟨*external name*⟩ `(`⟨*field*⟩$^+$`))`
⟨*field*⟩ → `(` ⟨*external name*⟩ ⟨*type name*⟩ `)`
⟨*external name*⟩ → ⟨*name*⟩ ⟨*string*⟩
⟨*type name*⟩ → ⟨*name*⟩

Each ⟨*structure*⟩ definition provides the following set of ⟨*c-expr*⟩. First, there is `(size-of` ⟨*structure name*⟩`)` which computes the size of the structure in bytes. It properly handles all alignment requirements for parts of the structure and produces the same result as `sizeof (struct foo)` in C where `foo` is the corresponding C tag. Second, there is `(align-of` ⟨*structure name*⟩`)` computing the alignment of the structure in bytes. It is always a power of 2. In addition, for each component of the structure, there is `(offset-of` ⟨*structure name*⟩ ⟨*field name*⟩`)`, which computes an offset from the begining of the structure to the given field.
Array types are introduced with the following contruct:

⟨*array*⟩ → `(array` ⟨*external name*⟩ ⟨*base type name*⟩ ⟨*constant expression*⟩`)`
⟨*base type name*⟩ → ⟨*type name*⟩

There is also a set of predefined types. At this stage we do not specify them.
Constants, predefined types, arrays, and structures may be aliased. For simplicity, we do not keep separate name spaces for them. All definitions are checked for conflicts.

⟨*alias*⟩ → `(alias` ⟨*new name*⟩ ⟨*old name*⟩`)`
⟨*new name*⟩ → ⟨*name*⟩
⟨*old name*⟩ → ⟨*name*⟩

The last part of the top level structure is procedure. At this time we only define leaf procedures. Since we need to generate multiple variants of the procedures from the same sources, external names are generated from the attributes. The internal names are not used in this version.

⟨*procedure*⟩ → `(procedure` ⟨*name*⟩ `(` ⟨*attribute*⟩*`)` `(` ⟨*argument*⟩*`)` ⟨*code*⟩$^+$`)`
⟨*argument*⟩ → `(` ⟨*argument name*⟩ ⟨*type name*⟩ ⟨*C type*⟩ ⟨*C name*⟩ `)`
⟨*argument name*⟩ → ⟨*name*⟩
⟨*C type*⟩ → ⟨*string*⟩
⟨*C name*⟩ → ⟨*string*⟩
⟨*attribute*⟩ → ⟨*name*⟩
| `(` ⟨*name*⟩ ⟨*a-value*⟩$^+$`)`
⟨*a-value*⟩ → ⟨*string*⟩
| ⟨*number*⟩
| ⟨*symbol*⟩

The code is design to help with instruction generation and data flow analysis. At this stage we keep regular loops and provide conditional branching. There is no back branches or returns. We do not provide automatic variables since there are only leaf procedures.

⟨*code*⟩ → ⟨*operation*⟩
| ⟨*memory access*⟩
| ⟨*block*⟩
| ⟨*loop*⟩
| ⟨*conditional*⟩
| ⟨*macro call*⟩
| ⟨*inner repeat*⟩
⟨*inner repeat*⟩ → `(repeat (` ⟨*iterator*⟩*`)` ⟨*code*⟩$^+$`)`

To insert the body of ⟨*macro definition*⟩, one provides arguments to the ⟨*macro call*⟩ form. All registers used as outputs in the body of the definition will be renamed thus providing hygenic semantics of macro substitution. There is no syntactic distinction between inputs and outputs in the call parameters.

⟨*macro call*⟩ → `(macro` ⟨*name*⟩ ⟨*macro input*⟩*`)`
⟨*macro input*⟩ → `(macro` ⟨*name*⟩`)`
| ⟨*input*⟩

The `macro` form of the macro argument is used to pass macros by name.

Operations modify only their outputs and depends only on their inputs. The name space of the opcodes is open-ended, the attributes are specific for each opcode.

⟨*operation*⟩ → `(` `op` ⟨*opcode*⟩ `(` ⟨*attribute*⟩* `)` `(` ⟨*output*⟩$^+$ `)` `(` ⟨*input*⟩* `)` `)`
⟨*opcode*⟩ → ⟨*name*⟩
⟨*output*⟩ → ⟨*name*⟩
⟨*input*⟩ → ⟨*register*⟩
| ⟨*constant expression*⟩
⟨*register*⟩ → `(` `reg` ⟨*name*⟩ `)`

Memory operations read and write memory. The type of the data is provided explicitly and is checked for writes (reads define the output which is checked at use.)

⟨*memory access*⟩ → `(` `load` ⟨*type name*⟩ `(` ⟨*attribute*⟩* `)` ⟨*output*⟩ ⟨*address*⟩ `)`
| `(` `store` ⟨*type name*⟩ `(` ⟨*attribute*⟩* `)` ⟨*address*⟩ ⟨*input*⟩ `)`
⟨*address*⟩ → `(` ⟨*input*⟩$^+$ `)`

The next is the block. It packages codes together into a single code to simplify syntax.

⟨*block*⟩ → `(` `begin` ⟨*code*⟩$^+$ `)`

The conditional is also simple. We provide two forms. The constant predicates are handled in the compile time.

⟨*conditional*⟩ → `(` `if-else` ⟨*predicate*⟩ ⟨*non-zero branch*⟩ ⟨*zero branch*⟩ `)`
| `(` `if` ⟨*predicate*⟩ ⟨*non-zero branch*⟩ `)`
⟨*predicate*⟩ → ⟨*input*⟩
⟨*non-zero branch*⟩ → ⟨*code*⟩
⟨*zero branch*⟩ → ⟨*code*⟩

The loop construct always runs through a given number of iterations. The limits are computed once at the beginning of the loop. The loop variable is of time int and is visible only inside the loop. Unrolling and prefetching inside the loop is controlled by the attributes. The loop step is alway 1.

⟨*loop*⟩ → `(` `loop` `(` ⟨*attribute*⟩* `)` `(` ⟨*loop variable*⟩ ⟨*inclusive low*⟩ ⟨*exclusive high*⟩ `)` ⟨*code*⟩$^+$ `)`
⟨*loop variable*⟩ → ⟨*output*⟩
⟨*inclusive low*⟩ → ⟨*input*⟩
⟨*exclusive high*⟩ → ⟨*input*⟩

It is possible to use quare brackets `[]` instead if parentheses `()` in any place as long as the closing backet matches the open one. Also, `;` starts a commnet upto the end of line.

2 TARGETS

The following targets are supported:

`c-header` The header file for C.

`c99` The programming language C as defined in ISO/IEC 9989.

`c99-64` The programming language C as defined in ISO/IEC 9989 on a 64-bit machine.

`cee` The programming language C without complex arithmetics.

`cee-64` The programming language C without complex arithmetics on a 64-bit machine.

`xlc/bgl` IBM's XLC compiler for BG/L with intrinsics.

`bgl` BG/L assembler.

3 PROCEDURE ATTRIBUTES

The following procedure attributes are understood:

`( stem` ⟨*a-value*⟩$^+$`)`

All values of the `stem` are concatenated to produce the stem of the procedure name. The result is affixed and suffixed appropriately to produce the procedure name.

`count-flops`

If present, the floating point operations in the procedure will be counted and returned as an `unsigned int` result.

`( return` ⟨*name*⟩ ⟨*type*⟩ ⟨*C type*⟩`)`

Specifies that the procedure returns the final value of register ⟨*name*⟩ and has ⟨*C type*⟩ as the return type. Value of ⟨*type*⟩ is used to generate appropriate low level code and should match ⟨*C type*⟩.

`(return` ⟨*output*⟩ ⟨*type name*⟩ ⟨*C type*⟩`)`

The final value of ⟨*output*⟩ will be returned from the procedure. The procedure will have ⟨*C type*⟩ as its return type. The values of ⟨*output*⟩ should be of type ⟨*type name*⟩.
A procedure can not have both `count-flops` and `return` attributes.

4 CONSTANT OPERATIONS

`(size-of` T`)`

Computes size of type T.

`(offset-of` T f`)`

Computes offset of field f in type T.

`(+` a b^*`)`

Computes $a + b \ldots$.

`(-` a b^*`)`

Computes $a - b \ldots$.

`(*` a b^*`)`

Computes $a * b \ldots$.

`(=` a^+`)`

If all arguments are equal, evaluates to 1, otherwise to 0.

`(shift` a b`)`

Computes $\lfloor 2^b a \rfloor$.

`(and` a^*`)`

Evaluates to 1 is all arguments are non-zero, and to 0 otherwise.

`(or` a^*`)`

Evaluates to 1 is any argument is non-zero, and to 0 otherwise.

`(not` a`)`

Evaluates to 1 is $a = 0$ and to 0 otherwise.

5 OPERATIONS

In the following, non-terminal suffices denote implied types:

i	integer
f	single precision floating point
d	double precision floating point
c	double precision complex number
p	pointer
x, y, z, q	unspecifiend types used for reductions
F	QCD fermion
l	a lower half of QCD fermion
h	an upper half of QCD fermion
H	QCD projected fermion
S	QCD staggered fermion
G	QCD gauge field

5.1 QCD operations

`( load qcd-fermion ()` r_F `(` a_p b_i`*) )`

Load QCD fermion from $M[a + b + \cdots]$ in current precision into register r.

`( load qcd-fermion-lo ()` r_l `(` a_p b_i`*) )`

Load the lower half of QCD fermion from $M[a + b + \cdots]$ in current precision into register r.

`( load qcd-fermion-hi ()` r_h `(` a_p b_i`*) )`

Load the upper half of QCD fermion from $M[a + b + \cdots]$ in current precision into register r.

`( load qcd-projected-fermion ()` r_H `(` a_p b_i`*) )`

Load QCD projected fermion from $M[a + b + \cdots]$ in current precision into register r.

`( load qcd-staggered-fermion ()` r_S `(` a_p b_i`*) )`

Load QCD staggered fermion from $M[a + b + \cdots]$ in current precision into register r.

`( load qcd-su-n ()` r_G `(` a_p b_i`*) )`

Load QCD gauge matrix from $M[a + b + \cdots]$ in current precision into register r.

`( store qcd-fermion ) (` a_p b_i`*)` r_F `)`

Store QCD fermion r into $M[a + b + \cdots]$ in current precision.

`( store qcd-fermion-lo ) (` a_p b_i`*)` r_l `)`

Store the lower half of QCD fermion in register r into $M[a + b + \cdots]$ in current precision.

`( store qcd-fermion-hi ) (` a_p b_i`*)` r_h `)`

Store the upper half of QCD fermion in register r into$M[a + b + \cdots]$ in current precision.

`( store qcd-projected-fermion ) (` a_p b_i`*)` r_H `)`

Store QCD projected fermion r into $M[a + b + \cdots]$ in current precision.

`( store qcd-staggered-fermion ) (` a_p b_i`*)` r_S `)`

Store QCD staggered fermion r into $M[a + b + \cdots]$ in current precision.

`( store qcd-su-n ) (` a_p b_i`*)` r_G `)`

Store QCD gauge matrix r into $M[a + b + \cdots]$ in current precision.

`(op qcd-mulf ) (` r_F `) (` u_G a_F `) )`

Compute $r \leftarrow ua$ for a fermion.

`(op qcd-mulf-conj () ( $r_F$ ) ( $u_G$ $a_F$ ) )`

Compute $r \leftarrow u^\dagger a$ for a fermion.

`(op qcd-mulh () ( $r_H$ ) ( $u_G$ $a_H$ ) )`

Compute $r \leftarrow ua$ for a projected fermion.

`(op qcd-muls () ( $r_S$ ) ( $u_G$ $a_S$ ) )`

Compute $r \leftarrow ua$ for a staggered fermion.

`(op qcd-mulh-conj () ( $r_H$ ) ( $u_G$ $a_H$ ) )`

Compute $r \leftarrow u^\dagger a$ for a projected fermion.

`(op qcd-muls-conj () ( $r_S$ ) ( $u_G$ $a_S$ ) )`

Compute $r \leftarrow u^\dagger a$ for a staggered fermion.

`(op qcd-su-n-mul () ( $r_G$ ) ( $u_G$ $v_G$ ) )`

Compute $r \leftarrow uv$ for gauge fields.

`(op qcd-su-n-mul-conj () ( $r_G$ ) ( $u_G$ $v_G$ ) )`

Compute $r \leftarrow uv^\dagger$ for gauge fields.

`(op qcd-su-n-conj-mul () ( $r_G$ ) ( $u_G$ $v_G$ ) )`

Compute $r \leftarrow u^\dagger v$ for gauge fields.

`(op qcd-su-n-conj-mul-conj () ( $r_G$ ) ( $u_G$ $v_G$ ) )`

Compute $r \leftarrow u^\dagger v^\dagger$ for gauge fields.

`(op qcd-su-n-real-trace-conj-mul () ( $r_d$ ) ( $b_G$ $c_G$) )`

Computes $r \leftarrow \Re\mathrm{Tr}(b^\dagger c)$ for gauge fields.

`(op qcd-zerou () ( $r_G$ ) ( ) )`

Set $r \leftarrow 0$ for a gauge matrix.

`(op qcd-zerof () ( $r_F$ ) ( ) )`

Set $r \leftarrow 0$ for a fermion.

`(op qcd-zeroh () ( $r_H$ ) ( ) )`

Set $r \leftarrow 0$ for a projected fermion.

`(op qcd-zeros () ( $r_S$ ) ( ) )`

Set $r \leftarrow 0$ for a staggered fermion.

`(op qcd-scaleu () ( $r_G$ ) ( $v_d$ $x_G$ ) )`

Compute $r \leftarrow vx$ for a gauge matrix.

`(op qcd-scalef () ( $r_F$ ) ( $v_d$ $x_F$ ) )`

Compute $r \leftarrow vx$ for a fermion.

`(op qcd-scaleh () ( $r_H$ ) ( $v_d$ $x_H$ ) )`

Compute $r \leftarrow vx$ for a projected fermion.

`(op qcd-scales () ( $r_S$ ) ( $v_d$ $x_S$ ) )`

Compute $r \leftarrow vx$ for a staggered fermion.

`(op qcd-addu () ( $r_G$ ) ( $a_G$ $b_G$ ) )`

Compute $r \leftarrow a + b$ for a gauge matrix.

`(op qcd-addf () ( $r_F$ ) ( $a_F$ $b_F$ ) )`

Compute $r \leftarrow a + b$ for a fermion.

`(op qcd-addh () ( $r_H$ ) ( $a_H$ $b_H$ ) )`

Compute $r \leftarrow a + b$ for a projected fermion.

`(op qcd-adds () ( $r_S$ ) ( $a_S$ $b_S$ ) )`

Compute $r \leftarrow a + b$ for a staggered fermion.

`(op qcd-subu () ( $r_G$ ) ( $a_G$ $b_G$ ) )`

Compute $r \leftarrow a - b$ for a gauge matrix.

`(op qcd-subf () ( $r_F$ ) ( $a_F$ $b_F$ ) )`

Compute $r \leftarrow a - b$ for a fermion.

`(op qcd-subh () ( $r_H$ ) ( $a_H$ $b_H$ ) )`

Compute $r \leftarrow a - b$ for a projected fermion.

`(op qcd-subs () ( $r_S$ ) ( $a_S$ $b_S$ ) )`

Compute $r \leftarrow a - b$ for a staggered fermion.

`(op qcd-maddf () ( $r_F$ ) ( $a_F$ $s_d$ $b_F$ ) )`

Compute $r \leftarrow a + sb$ for a fermion.

`(op qcd-maddf-lo () ( $r_F$ ) ( $a_F$ $s_d$ $b_F$ ) )`

Compute $r \leftarrow a + sb$ for the lower half of a fermion.

`(op qcd-maddf-hi () ( $r_F$ ) ( $a_F$ $s_d$ $b_F$ ) )`

Compute $r \leftarrow a + sb$ for the upper halft of the fermion.

`(op qcd-maddh () ( $r_H$ ) ( $a_H$ $s_d$ $b_H$ ) )`

Compute $r \leftarrow a + sb$ for a projected fermion.

`(op qcd-madds () ( $r_S$ ) ( $a_S$ $s_d$ $b_S$ ) )`

Compute $r \leftarrow a + sb$ for a staggered fermion.

`(op qcd-msubf () ( $r_F$ ) ( $a_F$ $s_d$ $b_F$ ) )`

Compute $r \leftarrow a - sb$ for a fermion.

`(op qcd-msubf-lo () ( $r_F$ ) ( $a_F$ $s_d$ $b_F$ ) )`

Compute $r \leftarrow a - sb$ for the lower half of a fermion.

`(op qcd-msubf-hi () ( $r_F$ ) ( $a_F$ $s_d$ $b_F$ ) )`

Compute $r \leftarrow a - sb$ for the upper halft of the fermion.

`(op qcd-msubh () ( $r_H$ ) ( $a_H$ $s_d$ $b_H$ ) )`

Compute $r \leftarrow a - sb$ for a projected fermion.

`(op qcd-msubs () ( $r_S$ ) ( $a_S$ $s_d$ $b_S$ ) )`

Compute $r \leftarrow a - sb$ for a staggered fermion.

`(op qcd-madd-lohi () ( $r_F$ ) ( $a_F$ $x_d$ $b_l$ $y_d$ $c_h$ ) )`

Compute $r \leftarrow a + xb + yc$ for a fermion a, a lower half b and an upper half c.

`(op qcd-fnorm-init () ( $r_z$ ) ( ) )`

Prepare to compute fermion norm. The type of r is unspecified here.

`(op qcd-fnorm-add () ( $r_z$ ) ( $a_z$ $b_F$ ) )`

Add a fermion to the norm computation, $r \leftarrow a + b^\dagger b$.

`(op qcd-fnorm-lo-add () ( $r_z$ ) ( $a_z$ $b_F$ ) )`

Add low part of a fermion to the norm computation, $r \leftarrow a + lo(b)^\dagger lo(b)$.

`(op qcd-fnorm-hi-add () ( $r_z$ ) ( $a_z$ $b_F$ ) )`

Add high part of a fermion to the norm computation, $r \leftarrow a + hi(b)^\dagger hi(b)$.

`(op qcd-fnorm-fini () ( $r_d$ ) ( $a_z$ ) )`

Extract the final value of the norm from the norm computation.

`(op qcd-snorm-init () ( $r_z$ ) ( ) )`

Prepare to compute staggered fermion norm. The type of r is unspecified here.

`(op qcd-snorm-add () ( $r_z$ ) ( $a_z$ $b_S$ ) )`

Add a staggered fermion to the norm computation, $r \leftarrow a + b^\dagger b$.

`(op qcd-snorm-fini () ( $r_d$ ) ( $a_z$ ) )`

Extract the final value of the norm from the norm computation.

`(op qcd-fdot-init () ( $r_q$ ) ( ) )`

Prepare to compute fermion dot product. The type of r is unspecified here.

`(op qcd-fdot-add () ( $r_q$ ) ( $a_q$ $b_F$ $c_F$ ) )`

Add a fermion to the dot product computation, $r \leftarrow a + b^\dagger c$.

`(op qcd-fdot-fini () ( $r_c$ ) ( $a_q$ ) )`

Extract the final value from the dot product computation.

`(op qcd-sdot-init () ( $r_q$ ) ( ) )`

Prepare to compute staggered fermion dot product. The type of r is unspecified here.

`(op qcd-sdot-add () ( $r_q$ ) ( $a_q$ $b_F$ $c_F$ ) )`

Add a staggered fermion to the dot product computation, $r \leftarrow a + b^\dagger c$.

`(op qcd-sdot-fini () ( $r_c$ ) ( $a_q$ ) )`

Extract the final value from the dot product computation.

`(op qcd-fermion-offset () ( $r_i$ ) ( $c_i$ $\alpha_i$ ) )`

Compute an offset suitable for loading ψ^c_α with `load complex` for a fermion.

`(op qcd-projected-fermion-offset () ( $r_i$ ) ( $c_i$ $\alpha_i$ ) )`

Compute an offset suitable for loading ψ^c_α with `load complex` for a projected fermion.

`(op qcd-su-n-offset () ( $r_i$ ) ( $a_i$ $b_i$ ) )`

Compute an offset suitable for loading U^a_b with `load complex` for a gauge matrix.

`(op qcd-project ( [ project` d s `] ) ( $r_H$ ) ( $a_F$ ) )`

Compute the projection of a fermion: if s is `plus`, compute $r \leftarrow (1 + \gamma_d)a$, if s is `minus`, compute $r \leftarrow (1 - \gamma_d)a$.

`(op qcd-unproject ( [ unproject` d s `] ) ( $r_F$ ) ( $a_H$ ) )`

Recover a fermion from projection: if s is `plus`, assume the projection was $(1+\gamma_d)a$, if s is `minus`, assume the projection was $(1 - \gamma_d)a$.

`(op qcd-unproject-add ( [ unproject` d s `] ) ( $r_F$ ) ( $a_F$ $b_H$ ) )`

Recover a fermion from projection: if s is `plus`, assume the projection was $(1+\gamma_d)b$, if s is `minus`, assume the projection was $(1 - \gamma_d)b$; add the result to a.

5.2 Complex operations

`( load COMPLEX () ` r_c `(` a_p $b_i{}^*$`) )`

Load default precision complex from $M[a+b+\cdots]$ into registero r. register r.
Specific size operations:

`( load complex-float () ` r_c `(` a_p $b_i{}^*$`) )`

Load single precision complex from $M[a+b+\cdots]$ into registero r. register r.

`( load complex-double () ` r_c `(` a_p $b_i{}^*$`) )`

Load double precision complex from $M[a+b+\cdots]$ into registero r. register r.

`( store COMPLEX () (` a_p $b_i{}^*$`)` r_c `)`

Store default precision complex r into $M[a+b+\cdots]$.

`( store complex-float () (` a_p $b_i{}^*$`)` r_c `)`

Store single precision complex r into $M[a+b+\cdots]$.

`( store complex-double () (` a_p $b_i{}^*$`)` r_c `)`

Store double precision complex r into $M[a+b+\cdots]$.
Double precision complex computations:

`(op complex-add ( ) (` r_c `) (` a_c b_c `) )`

Compute $r \leftarrow a+b$.

`(op complex-sub ( ) (` r_c `) (` a_c b_c `) )`

Compute $r \leftarrow a-b$.

`(op complex-add-i ( ) (` r_c `) (` a_c b_c `) )`

Compute $r \leftarrow a+ib$.

`(op complex-sub-i ( ) (` r_c `) (` a_c b_c `) )`

Compute $r \leftarrow a-ib$.

`(op complex-times-plus-i ( ) (` r_c `) (` b_c `) )`

Compute $r \leftarrow ia$.

`(op complex-times-minus-i ( ) (` r_c `) (` a_c `) )`

Compute $r \leftarrow -ia$.

`(op complex-rmadd ( ) (` r_c `) (` a_c α_d b_c `) )`

Compute $r \leftarrow a+\alpha b$.

`(op complex-cmadd ( ) (` r_c `) (` a_c b_c c_c `) )`

Compute $r \leftarrow a+bc$.

`(op complex-rmul ( ) (` r_c `) (` α_d a_c `) )`

Compute $r \leftarrow \alpha a$.

`(op complex-cmul ( ) (` r_c `) (` a_c b_c `) )`

Compute $r \leftarrow ab$.

`(op complex-cmul-conj ( ) (` r_c `) (` a_c b_c `) )`

Compute $r \leftarrow a^*b$.

`(op complex-neg ( ) (` r_c `) (` a_c `) )`

Compute $r \leftarrow -a$.

`(op complex-zero ( ) ( $r_c$ ) ( ) )`

Set $r \leftarrow 0$.

`(op complex-move ( ) ( $r_c$ ) ( $a_c$ ) )`

Set $r \leftarrow a$.

`(op complex ( ) ( $r_c$ ) ( $a_d$ $b_d$ ) )`

Compute $r \leftarrow a + ib$.

`(op complex-real ( ) ( $r_d$ ) ( $a_c$ ) )`

Compute $r \leftarrow \Re a$.

`(op complex-imag ( ) ( $r_d$ ) ( $a_c$ ) )`

Compute $r \leftarrow \Im a$.

`(op complex-norm-init () ( $r_x$ ) () )`

Prepare to compute complex norm squared.

`(op complex-norm-add () ( $r_x$ ) ( $a_x$ $b_c$ ) )`

Add an element to the complex norm $r \leftarrow a + b^* b$.

`(op complex-norm-fini () ( $r_d$ ) ( $a_x$ ) )`

Extract the value for the norm $r \leftarrow a$.

`(op complex-dot-init () ( $r_y$ ) () )`

Prepare to compute complex dot product squared.

`(op complex-dot-add () ( $r_x$ ) ( $a_x$ $b_c$ $c_c$) )`

Add an element to the complex dot product $r \leftarrow a + b^* c$.

`(op complex-dot-fini () ( $r_c$ ) ( $a_x$ ) )`

Extract the value for the dot product $r \leftarrow a$.

`(op complex-real-cmul-conj-init () ( $r_q$ ) ( $a_c$ $b_c$) )`

Start computing $\sum \Re(x_i^* y_i)$: computes $r \leftarrow \Re(b^* c)$ but keeps r in an internal form.

`(op complex-real-cmul-conj-add () ( $r_q$ ) ( $a_q$ $b_c$ $c_c$) )`

Continues computing $\sum \Re(x_i^* y_i)$, keeps r in the internal form Computes $r \leftarrow a + \Re(b^* c)$.

`(op complex-real-cmul-conj-fini () ( $r_d$ ) ( $a_q$ ) )`

Finishes computing $r = \sum \Re(x_i^* y_i)$, extracts r from the internal form.

5.3 Real operations

`( load REAL () $r_d$ ( $a_p$ $b_i{}^*$) )`

Load default precision real from $M[a + b + \cdots]$ into register r.

`( load float () $r_d$ ( $a_p$ $b_i{}^*$) )`

Load single precision real from $M[a + b + \cdots]$ into register r.

`( load double () $r_d$ ( $a_p$ $b_i{}^*$) )`

Load double precision real from $M[a + b + \cdots]$ into register r.

`( store REAL () ( $a_p$ $b_i{}^*$) $r_G$ )`

Store default precision r into $M[a + b + \cdots]$ in current precision.

`( store float () ( $a_p$ $b_i{}^*$) $r_G$ )`

Store single precision r into $M[a+b+\cdots]$ in current precision.

`( store double () (` a_p b_i`*) ` r_G `)`

Store double precision r into $M[a+b+\cdots]$ in current precision.

`(op double-add ( ) (` r_d `) (` a_d b_d `) )`

Compute $r \leftarrow a+b$.

`(op double-sub ( ) (` r_d `) (` a_d b_d `) )`

Compute $r \leftarrow a-b$.

`(op double-mul ( ) (` r_d `) (` a_d b_d `) )`

Compute $r \leftarrow ab$.

`(op double-div ( ) (` r_d `) (` a_d b_d `) )`

Compute $r \leftarrow a/b$.

`(op double-madd ( ) (` r_d `) (` a_d b_d c_d `) )`

Compute $r \leftarrow a+bc$.

`(op double-msub ( ) (` r_d `) (` a_d b_d c_d `) )`

Compute $r \leftarrow a-bc$.

`(op double-move ( ) (` r_d `) (` a_d `) )`

Set $r \leftarrow a$.

`(op double-neg ( ) (` r_d `) (` a_d `) )`

Compute $r \leftarrow -a$.

`(op double-zero ( ) (` r_d `) ( ) )`

Set $r \leftarrow 0$.

5.4 Integer operations

`( load int () ` r_G `(` a_i b_i`*) )`

Load integer from $M[a+b+\cdots]$ in register r.

`( store int () (` a_p b_i`*) ` r_i `)`

Store integer r into $M[a+b+\cdots]$.

`(op int-add ( ) (` r_i `) (` a_i b_i `) )`

Compute $r \leftarrow a+b$.

`(op int-sub ( ) (` r_i `) (` a_i b_i `) )`

Compute $r \leftarrow a-b$.

`(op int-mul ( ) (` r_i `) (` a_i b_i `) )`

Compute $r \leftarrow ab$.

`(op int-div ( ) (` r_i `) (` a_i b_i `) )`

Compute $r \leftarrow a \,\mathbf{div}\, b$.

`(op int-mod ( ) (` r_i `) (` a_i b_i `) )`

Compute $r \leftarrow a \,\mathbf{mod}\, b$.

`(op int-and ( ) (` r_i `) (` a_i b_i `) )`

Compute $r \leftarrow a \,\mathbf{and}\, b$.

`(op int-or ( ) ( ` r_i ` ) ( ` a_i b_i ` ) )`

Compute $r \leftarrow a$ **or** b.

`(op int-xor ( ) ( ` r_i ` ) ( ` a_i b_i ` ) )`

Compute $r \leftarrow a$ **xor** b.

`(op int-move ( ) ( ` r_i ` ) ( ` a_i ` ) )`

Compute $r \leftarrow a$.

`(op int-neg ( ) ( ` r_i ` ) ( ` a_i ` ) )`

Compute $r \leftarrow -a$.

5.5 Pointer operations

`( load pointer () ` r_p ` ( ` a_p $b_i{}^*$`) )`

Load pointer from $M[a + b + \cdots]$ into register r.

`( store pointer () ( ` a_p $b_i{}^*$`) ` r_p ` )`

Store pointer r into $M[a + b + \cdots]$.

`(op pointer-add ( ) ( ` r_p ` ) ( ` a_p b_i`) )`

Compute $r \leftarrow a + b$.

`(op pointer-move ( ) ( ` r_p ` ) ( ` a_p ` ) )`

Compute $r \leftarrow a$.