

QA—Register Transfer Language

Andrew Pochinsky

\$Id: qa0.nw 505 2008-04-03 22:41:04Z avp \$

1 GRAMMAR

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

$\langle qa0 \rangle$	\rightarrow	$\langle decl \rangle^*$
$\langle decl \rangle$	\rightarrow	$\langle alias \rangle$
		$\langle constant \rangle$
		$\langle structure \rangle$
		$\langle array \rangle$
		$\langle verbose \rangle$
		$\langle include \rangle$
		$\langle macro\ definition \rangle$
		$\langle top\ level\ repeat \rangle$
$\langle top\ level\ repeat \rangle$	\rightarrow	$\langle procedure \rangle$
		$(\text{repeat } (\langle iterator \rangle^*) \langle top\ level\ repeat \rangle^+)$

The $\langle macro\ definition \rangle$ directive defines a hygenic macro.

$\langle macro\ definition \rangle$	\rightarrow	$(\text{define } (\langle name \rangle \langle name \rangle^*) \langle code \rangle^+)$
-------------------------------------	---------------	--

The $\langle include \rangle$ directive inserts the contents of the file into the input.

$\langle include \rangle$	\rightarrow	$(\text{include } \langle file \rangle)$
$\langle file \rangle$	\rightarrow	$\langle string \rangle$

The $\langle verbose \rangle$ directive allows one to include literals into various outputs:

$\langle verbose \rangle$	\rightarrow	$(\text{verbose } \langle verbose\ case \rangle^+)$
$\langle verbose\ case \rangle$	\rightarrow	$(\langle target \rangle \langle verbose\ value \rangle)$
$\langle target \rangle$	\rightarrow	$\langle symbol \rangle$
$\langle verbose\ data \rangle$	\rightarrow	$\langle string \rangle$

The $\langle target \rangle$ selects the back-end that will see the corresponding $\langle verbose\ data \rangle$.

There are two kinds of iterators. One is an enumeration, another is loop-like

$\langle iterator \rangle$	\rightarrow	$(\langle name \rangle (\langle a-value \rangle^+))$
		$(\langle name \rangle \langle inclusive\ constant\ low \rangle \langle exclusive\ constant\ high \rangle)$
$\langle inclusive\ constant\ low \rangle$	\rightarrow	$\langle constant\ expr \rangle$
$\langle exclusive\ constant\ high \rangle$	\rightarrow	$\langle constant\ expr \rangle$

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

$\langle constant \rangle$	\rightarrow	$(\text{constant } \langle name \rangle \langle constant\ expression \rangle)$
$\langle constant\ expression \rangle$	\rightarrow	$(\text{const } \langle c-expr \rangle)$
$\langle c-expr \rangle$	\rightarrow	$\langle number \rangle$
		$\langle name \rangle$
		$\langle string \rangle$
		$(\langle c-op \rangle \langle c-expr \rangle^*)$
$\langle c-op \rangle$	\rightarrow	$\langle symbol \rangle$
$\langle name \rangle$	\rightarrow	$\langle symbol \rangle$

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.

$\langle \text{structure} \rangle$	\rightarrow	$(\text{structure } \langle \text{external name} \rangle (\langle \text{field} \rangle^+))$
$\langle \text{field} \rangle$	\rightarrow	$(\langle \text{external name} \rangle \langle \text{type name} \rangle)$
$\langle \text{external name} \rangle$	\rightarrow	$\langle \text{name} \rangle \langle \text{string} \rangle$
$\langle \text{type name} \rangle$	\rightarrow	$\langle \text{name} \rangle$

Each $\langle \text{structure} \rangle$ definition provides the following set of $\langle \text{c-expr} \rangle$. First, there is $(\text{size-of } \langle \text{structure name} \rangle)$ 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 $(\text{align-of } \langle \text{structure name} \rangle)$ 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 $(\text{offset-of } \langle \text{structure name} \rangle \langle \text{field name} \rangle)$, which computes an offset from the beginning of the structure to the given field.

Array types are introduced with the following construct:

$\langle \text{array} \rangle$	\rightarrow	$(\text{array } \langle \text{external name} \rangle \langle \text{base type name} \rangle \langle \text{constant expression} \rangle)$
$\langle \text{base type name} \rangle$	\rightarrow	$\langle \text{type name} \rangle$

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.

$\langle \text{alias} \rangle$	\rightarrow	$(\text{alias } \langle \text{new name} \rangle \langle \text{old name} \rangle)$
$\langle \text{new name} \rangle$	\rightarrow	$\langle \text{name} \rangle$
$\langle \text{old name} \rangle$	\rightarrow	$\langle \text{name} \rangle$

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.

$\langle \text{procedure} \rangle$	\rightarrow	$(\text{procedure } \langle \text{name} \rangle (\langle \text{attribute} \rangle^*) (\langle \text{argument} \rangle^*) \langle \text{code} \rangle^+)$
$\langle \text{argument} \rangle$	\rightarrow	$(\langle \text{argument name} \rangle \langle \text{type name} \rangle \langle \text{C type} \rangle \langle \text{C name} \rangle)$
$\langle \text{argument name} \rangle$	\rightarrow	$\langle \text{name} \rangle$
$\langle \text{C type} \rangle$	\rightarrow	$\langle \text{string} \rangle$
$\langle \text{C name} \rangle$	\rightarrow	$\langle \text{string} \rangle$
$\langle \text{attribute} \rangle$	\rightarrow	$\langle \text{name} \rangle$
		$ (\langle \text{name} \rangle \langle \text{a-value} \rangle^+)$
$\langle \text{a-value} \rangle$	\rightarrow	$\langle \text{string} \rangle$
		$ \langle \text{number} \rangle$
		$ \langle \text{symbol} \rangle$

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.

$\langle \text{code} \rangle$	\rightarrow	$\langle \text{operation} \rangle$
		$ \langle \text{memory access} \rangle$
		$ \langle \text{block} \rangle$
		$ \langle \text{loop} \rangle$
		$ \langle \text{conditional} \rangle$
		$ \langle \text{macro call} \rangle$
		$ \langle \text{inner repeat} \rangle$
$\langle \text{inner repeat} \rangle$	\rightarrow	$(\text{repeat } (\langle \text{iterator} \rangle^*) \langle \text{code} \rangle^+)$

To insert the body of $\langle \text{macro definition} \rangle$, one provides arguments to the $\langle \text{macro call} \rangle$ 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.

$\langle \text{macro call} \rangle$	\rightarrow	$(\text{macro } \langle \text{name} \rangle \langle \text{macro input} \rangle^*)$
$\langle \text{macro input} \rangle$	\rightarrow	$(\text{macro } \langle \text{name} \rangle)$
		$ \langle \text{input} \rangle$

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 always 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 square brackets [] instead of parentheses () in any place as long as the closing bracket matches the open one. Also, ; starts a comment 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/bg1 IBM's XLC compiler for BG/L with intrinsics.

bg1 BG/L assembler.

3 PROCEDURE ATTRIBUTES

The following procedure attributes are understood:

(**stem** $\langle a\text{-value} \rangle^+$)

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** $\langle name \rangle$ $\langle type \rangle$ $\langle C\ type \rangle$)

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

(**return** $\langle output \rangle$ $\langle type\ name \rangle$ $\langle C\ type \rangle$)

The final value of $\langle output \rangle$ will be returned from the procedure. The procedure will have $\langle C\ type \rangle$ as its return type. The values of $\langle output \rangle$ should be of type $\langle type\ name \rangle$.

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 $a + b \dots$

(**-** $a\ b^*$)

Computes $a - b \dots$

(***** $a\ b^*$)

Computes $a * b \dots$

(**=** 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 if all arguments are non-zero, and to 0 otherwise.

(**or** a^*)

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

(**not** a)

Evaluates to 1 if $a = 0$ 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
G	QCD gauge field

5.1 QCD operations

(load qcd-fermion () r_F ($a_p b_i^*$))

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

(load qcd-fermion-lo () r_l ($a_p b_i^*$))

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

(load qcd-fermion-hi () r_h ($a_p b_i^*$))

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

(load qcd-projected-fermion () r_H ($a_p b_i^*$))

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

(load qcd-su-n () r_G ($a_p b_i^*$))

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

(store qcd-fermion () ($a_p b_i^*$) r_F)

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

(store qcd-fermion-lo () ($a_p b_i^*$) r_l)

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

(store qcd-fermion-hi () ($a_p b_i^*$) r_h)

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

(store qcd-projected-fermion () ($a_p b_i^*$) r_H)

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

(store qcd-su-n () ($a_p b_i^*$) r_G)

Store QCD gauge matrix r into $M[a + b + \dots]$ 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-mulh-conj () (rH) (uG aH))`
 Compute $r \leftarrow u^\dagger a$ for a projected fermion.

`(op qcd-su-n-mul () (rG) (uG vG))`
 Compute $r \leftarrow uv$ for gauge fields.

`(op qcd-su-n-mul-conj () (rG) (uG vG))`
 Compute $r \leftarrow uv^\dagger$ for gauge fields.

`(op qcd-su-n-conj-mul () (rG) (uG vG))`
 Compute $r \leftarrow u^\dagger v$ for gauge fields.

`(op qcd-su-n-conj-mul-conj () (rG) (uG vG))`
 Compute $r \leftarrow u^\dagger v^\dagger$ for gauge fields.

`(op qcd-su-n-real-trace-conj-mul () (rd) (bG cG))`
 Computes $r \leftarrow \Re \text{Tr}(b^\dagger c)$ for gauge fields.

`(op qcd-scaleu () (rG) (vd xG))`
 Compute $r \leftarrow vx$ for a gauge matrix.

`(op qcd-scalef () (rF) (vd xF))`
 Compute $r \leftarrow vx$ for a fermion.

`(op qcd-scaleh () (rH) (vd xH))`
 Compute $r \leftarrow vx$ for a projected fermion.

`(op qcd-addu () (rG) (aG bG))`
 Compute $r \leftarrow a + b$ for a gauge matrix.

`(op qcd-addf () (rF) (aF bF))`
 Compute $r \leftarrow a + b$ for a fermion.

`(op qcd-addh () (rH) (aH bH))`
 Compute $r \leftarrow a + b$ for a projected fermion.

`(op qcd-maddf () (rF) (aF sd bF))`
 Compute $r \leftarrow a + sb$ for a fermion.

`(op qcd-maddf-lo () (rF) (aF sd bF))`
 Compute $r \leftarrow a + sb$ for the lower half of a fermion.

`(op qcd-maddf-hi () (rF) (aF sd bF))`
 Compute $r \leftarrow a + sb$ for the upper half of the fermion.

`(op qcd-maddh () (rH) (aH sd bH))`
 Compute $r \leftarrow a + sb$ for a projected fermion.

`(op qcd-madd-lohi () (rF) (aF xd bl yd ch))`
 Compute $r \leftarrow a + xb + yc$ for a fermion a , a lower half b and an upper half c .

`(op qcd-fnorm-init () (rz) ())`
 Prepare to compute fermion norm. The type of r is unspecified here.

`(op qcd-fnorm-add () (rz) (az bF))`
 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-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-fermion-offset () (r_i) ($c_i \alpha_i$))

Compute an offset suitable for loading ψ_α^c with `load complex` for a fermion.

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

Compute an offset suitable for loading ψ_α^c 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_b^a 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 $M[a + b + \dots]$ into register r . register r .
Specific size operations:

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

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

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

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

(store COMPLEX () ($a_p b_i^*$) r_c)

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

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

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

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

Store double precision complex r into $M[a + b + \dots]$.

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) ($a_c \alpha_d b_c$))

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) ($\alpha_d a_c$))

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-move () (r_c) (a_c))

Compute $r \leftarrow a$.

(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 $M[a + b + \dots]$ into register r .

(load float () r_d (a_p b_i^*))

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

(load double () r_d (a_p b_i^*))

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

(store REAL () (a_p b_i^*) r_G)

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

(store float () (a_p b_i^*) r_G)

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

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

Store double precision r into $M[a + b + \dots]$ 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))

Compute $r \leftarrow a$.

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

Compute $r \leftarrow -a$.

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

Compute $r \leftarrow 0$.

5.4 Integer operations

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

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

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

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

(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 $r \leftarrow a \mathbf{or} b$.

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

Compute $r \leftarrow a \mathbf{xor} b$.

(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 $M[a + b + \dots]$ into register r .

(store pointer () (a_p b_i^*) r_p)

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

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

Compute $r \leftarrow a + b$.

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

Compute $r \leftarrow a$.