The SURE Architecture

David May: December 11, 2016

Background

Computer programming is changing. Object-oriented languages, functional languages and others have accelerated software development. But these languages rely on automatic memory management which imposes high overheads when implemented on conventional computer architectures. It seems timely to investigate whether a change in architecture is now needed - similar to the shift from CISC to RISC in the 1980s in response to the shift from assembly programming to high-level languages.

Along with the change in programming languages, there has been a change in computer systems and their applications. Increasingly, computers are communicating with each other and with users, or they are interacting with a physical environment. This gives rise to a need for trust, achieved via an appropriate combination of simplicity, verification, testing and error containment. Again, it seems timely to investigate whether a change in architecture is needed.

An experimental architecture

The architecture described here employs a novel technique for allocating memory; a hardware *garbage collector* is embedded in the processor sharing memory cycles with the processor. The garbage collector runs continually and is normally fast enough to retrieve unused memory space as fast as new space is allocated.

It also checks that memory accesses are valid; dangling pointers and buffer overflows are detected by hardware. An address consists of two parts. One specifies a memory region; the other specifies a location within the region. For a memory access operation, the processor checks that one operand is an address and the other is a value to be used as an offset. It also checks that the offset specified in a memory access instruction lies within the region specified by the address operand.

The principles can be used in conjunction with any processor architecture; it would be simple to create an Intel, ARM, MIPS, SPARC or RISC-V processor including the memory management system described here. There would be a small cost in the hardware needed for the garbage collector; for a 32-bit architecture it needs about 400kbytes of high-speed memory for the garbage collector data-structures and about 3% more main memory.

It is also practical to implement a memory subsystem incorporating these principles, with one or more processors sharing access to it.

The principles are also compatible with most modern programming languages. They would provide more efficient implementations along with checks for memory errors. Buffer overflows and dangling pointers are detected by hardware.

The architecture has the potential to save energy. Along with a reducing the operations needed for memory allocation, the garbage collector creates a single region of unused memory which can be powered down.

Memory allocation

Memory is allocated in *tuples*. A GETM instruction allocates a new tuple containing n words numbered from 0 to $n-1$, and produces a *pointer* to word 0 of the tuple. A pointer differs from a data value. Every word in memory contains information to specify whether it holds a pointer value or a data value. The processor registers also contain information to specify whether they hold a pointer value or a data value; registers such as a program counter pc or a stack pointer sp always hold pointer values.

Each instruction checks to ensure that the register contents are appropriate for the operation it performs. For example, an instruction to load a word from memory must have a pointer operand. The memory access instructions that use a pointer to a tuple along with an offset check that the specified location falls within the tuple.

When a tuple is no longer needed, the memory space it occupies is recovered automatically by a *garbage collector* implemented by hardware in the processor. The garbage collector runs continually; each cycle of the garbage collector marks all of the tuples in use, then copies them all towards the bottom of the memory, recovering space occupied by the tuples no longer in use. Newly created tuples occupy space above the existing tuples.

To support the memory allocation and garbage collection, a tuple $directory$ is used. Each tuple has a corresponding entry in the directory which contains the address of the tuple in memory and the number of words in the tuple. It also has space to hold the address of another tuple which is used to form a list of unused directory entries (and is also used by the garbage collector). When a memory access is performed using a pointer, the most significant bits of the pointer are used as a *handle* to address a directory entry. This provides the memory address of word 0 of the tuple; the least significant bits of the pointer determine the offset of the word to be accessed. A check is made to ensure that the offset is less than the size of the tuple.

Each tuple in memory has a control word which contains the handle of the directory entry corresponding to the tuple. It can also be used to hold a small value to indicate, for example, the type of the tuple. This is known as a tag.

The garbage collector is implemented as a state machine. Each state transition performs at most one memory access. The state transitions are normally performed when the memory is not required for instruction fetch or instruction execution. If the memory or the directory is full, instruction execution stops until the garbage collector completes its current cycle.

Addresses and the Directory

Each word w in memory or in a register has a flag w_{ptr} and a word w_{word}. The word is a pointer value or a data value, indicated by w_{ptr} which is $true$ if w is a pointer, $false$ otherwise. If w_{word} is a pointer, then

w_{handle}	identifies a directory entry of a tuple
w_{offset}	identifies an address within a tuple

The handle is stored in the upper half of the word, the offset in the lower half.

Each directory entry d of a tuple has four components:

d_{addr}	the address of the tuple in memory
d_{size}	the size of the tuple (in words)
d_{mark}	the marking flag for garbage collection
d_{deep}	a flag to indicate whether the tuple contains pointers

For a 32-bit wordlength, there will be up to 65536 tuples each of size up to 65536 bytes and the directory will have 65536 entries each with 48 bits (only 14 are needed for the length).

Garbage collector variables

mem	represents the memory
$current$	points to list of directory entries currently being scanned
$next$	points to a list of directory entries to be scanned next
$free$	points to a list of free directory entries
$tuple$	is the tuple being processed
$size$	is the number of words in the tuple being processed
$index$	is the offset of a word within the tuple being processed
$livesize$	is the total size of the data that has been marked
$heappoint$	is the highest location in the heap
src	is the memory address from which a tuple are copied during sweeping
$dest$	is the memory address to which a tuple is copied during sweeping

Memory access variables

$buffer$	holds a word being read from or written to memory
$rwindex$	is the offset of a word being read from or written to memory
$pointer$	is a pointer used to access a tuple
$offset$	is a word offset used to access a word within a tuple

The Garbage Collector - Marking

The *marking* process uses two lists. The $current$ list contains tuples that have been marked but which need to be scanned to determine if they contain pointers to other tuples that must be marked. Tuples are taken from the current list and their contents are examined; when a pointer is found, the corresponding tuple is marked and added to the $next$ list. When the $current$ list is empty, normally the $next$ list wlll contain new tuples to be scanned; the $next$ list replaces the $current$ list and scanning continues. If the $next$ list is empty when the $current$ list becomes empty, the marking is complete and the garbage collector will move to the *sweeping* process. The last item on each list holds the value nil instead of a pointer to another item.

During the marking process, it is possible that new tuples will be created; these are added to the $next$ list. This also occurs whenever a tuple is assigned to a location in another tuple.

It is common for there to be tuples which contain data but no pointers. Each directory entry has an additional $deep$ marker. This is cleared when a tuple is created and is only set when a pointer is assigned to a location in the tuple. It is used to prevent unnecessary scanning during the marking process.

As the marking proceeds, a record $livesize$ is kept of the total amount of space that will be used by tuples that have been marked and that will be retained; this is used to optimise the sweeping process.

$$
\begin{aligned}
mark_{init}\text{: } \{\ & dir[pc_{handle}]_{mark} \leftarrow true \\
& \&\ dir[sp_{handle}]_{mark}, dir[sp_{handle}]_{deep} \leftarrow true, true \\
& \&\ dir[sp_{handle}]_{list}, next \leftarrow nil, sp_{handle} \\
& \&\ livesize \leftarrow livesize + dir[pc_{handle}]_{size} + dir[sp_{handle}]_{size} + 2 \\
& \&\ state \leftarrow mark_{next} \\
& \}
\end{aligned}
$$

$mark_{scan}$: if $index \le size$
then
if $mem[src+index]_{ptr} \wedge (mem[src+index]_{handle} \neq nil)$
then
$\{\ tuple, index \leftarrow mem[src+index]_{handle}, index+1$
$\&\ state \leftarrow mark_{add}$
$\}$
else $index \leftarrow index+1$
else
if $current = nil$
then $state \leftarrow mark_{next}$
else
$\{\ src, size, index \leftarrow dir[current]_{addr}, dir[current]_{size}, 1$
$\&\ current \leftarrow dir[current]_{list}$
$\}$

$mark_{add}$:$\{$ if $\neg dir[tuple]_{mark}$
then
$\{\ dir[tuple]_{mark}, livesize \leftarrow true, livesize + dir[tuple]_{size} + 1$
$\&$ if $dir[tuple]_{deep}$
then $dir[tuple]_{list}, next \leftarrow next, tuple$
else skip
$\}$
else skip
$\&\ state \leftarrow mark_{scan}$
$\}$

$mark_{next}$: if $next = nil$
then
$\{\ dest, src, tuple \leftarrow 0, 0, mem[0]_{handle}$
$\&\ state \leftarrow sweep_{scan}$
$\}$
else
$\{\ src, size, index \leftarrow dir[next]_{addr}, dir[next]_{size}, 1$
$\&\ current, next, state \leftarrow dir[next]_{list}, nil, mark_{scan}$
$\}$

The Garbage Collector - Sweeping

The *sweeping* process accesses all of the locations in memory up to the $heappoint$. It uses two address registers, src and $dest$ to copy each tuple from its current location to its final location. It uses the control word of each tuple to access the tuple's handle. This enables it to determine the size of the tuple and whether the tuple is marked. If the tuple is marked it is copied one word at a time from src to $dest$. This copying step is omitted if src and $dest$ are equal; this occurs if all tuples up to the current value of src have been retained.

The marking process can only operate correctly if newly created tuples are initialised so that their components are data (not pointer) values. This means that when marked tuples are moved, the locations they move from must be re-initialised. However, there is no need to do this if a tuple (or a tuple component) is moved from a location that will be occupied by other tuples moved during the sweeping process. This is controlled by comparing the source addresses against $livesize$. Similary, the locations occupied by unmarked tuples must be re-initialised unless they will be occupied by tuples moved during the sweeping process.

The sweeping process terminates when the src address matches the $heappoint$. Additional tuples may be created during the sweeping process; these will be marked as they are created so that they will be copied. When the sweeping process terminates, the $heappoint$ is set to the $dest$ address.

$sweep_{scan}$: val $size, nsrc = dir[tuple]_{size}, src + dir[tuple]_{size} + 1$ in
  if $dir[tuple]_{mark}$
  then
    if $src = dest$
    then
      if $nsrc = heappoint$
      then $livesize, state \leftarrow 0, sweep_{end}$
      else $src, dest, dir[tuple]_{mark}, tuple \leftarrow nsrc, nsrc, false, mem[nsrc]_{handle}$
    else $dir[tuple]_{mark}, index, state \leftarrow false, 0, sweep_{read}$
  else
  { $dir[tuple]_{list}, free \leftarrow free, tuple$
  & if $(src + size) < livesize$
    then $src, tuple \leftarrow nsrc, mem[nsrc]_{handle}$
    else
    if $src < livesize$
    then $index, state \leftarrow livesize - src, sweep_{zero}$
    else $index, state \leftarrow 0, sweep_{zero}$
  }

$sweep_{read}$: if $index <= size$
then
{ $copy_{word} \leftarrow mem[src + index]_{word}$
& $copy_{ptr} \leftarrow mem[src + index]_{ptr}$
& if $(src + index) < livesize$
then $state \leftarrow sweep_{write}$
else $state \leftarrow sweep_{clear}$
}
else
{ $dir[tuple]_{addr}, dest \leftarrow dest, dest + size + 1$
& if $(src + size + 1) = heappoint$
then $heappoint, livesize, state \leftarrow dest + size + 1, 0, sweep_{end}$
else
{ $src, tuple \leftarrow src + size + 1, mem[src + size + 1]_{handle}$
& $state \leftarrow sweep_{scan}$
}
}

$sweep_{clear}$: { $mem[src + index]_{word}, mem[src + index]_{ptr} \leftarrow 0, false$
& $state \leftarrow sweep_{write}$
}

$sweep_{write}$: { $mem[dest + index]_{word} \leftarrow copy_{word}$
& $mem[dest + index]_{ptr} \leftarrow copy_{ptr}$
& $index, state \leftarrow index + 1, sweep_{read}$
}

$sweep_{zero}$: if $index <= size$
then
{ $mem[src + index]_{word} \leftarrow 0$
& $mem[src + index]_{ptr} \leftarrow false$
& $index \leftarrow index + 1$
}
else
if $(src + size + 1) = heappoint$
then $heappoint, livesize, state \leftarrow dest, 0, sweep_{end}$
else
{ $src, tuple \leftarrow src + size + 1, mem[src + size + 1]_{handle}$
& $state \leftarrow sweep_{scan}$
}

Memory allocation and access

Memory for a tuple is allocated using one of the GETM instructions. These allocate a handle from the list of free handles and also allocate memory for the tuple at the $heappoint$. The number of words in the tuple, $space$ is specified by the operand of the GETM instruction. The control word of the newly created tuple is set and the tuple is marked.

$$
\begin{aligned}
getmem\text{: } \{\ & dir[free]_{addr}, dir[free]_{size} \leftarrow heappoint, space \\
& \&\ dir[free]_{mark}, dir[free]_{deep} \leftarrow true, false \\
& \&\ heappoint, livesize \leftarrow heappoint + space + 1, livesize + space + 1 \\
& \&\ mem[heappoint]_{handle}, mem[heappoint]_{ptr} \leftarrow free, false \\
& \&\ areg_{handle}, areg_{ptr}, free \leftarrow free, true, dir[free] \\
& \}
\end{aligned}
$$

The garbage collector operates at the same time that the processor is performing memory accesses. It is therefore possible that a tuple is being moved during the sweeping process when the processor attempts to read or write locations within it. This is detected by the hardware and the read or write access is made to the actual location of the data.

$$
\begin{aligned}
read\text{: } & \text{val } rwindex = pointer_{offset} + offset + 1 \text{ in} \\
& \text{if } (pointer_{handle} = tuple) \wedge (rwindex = index) \wedge \\
& \quad ((state = sweep_{clear}) \vee (state = sweep_{write})) \\
& \text{then } buffer_{word}, buffer_{ptr} \leftarrow copy_{word}, copy_{ptr} \\
& \text{else} \\
& \text{if } (pointer_{handle} = tuple) \wedge (rwindex < index) \wedge \\
& \quad ((state = sweep_{read}) \vee (state = sweep_{clear}) \vee (state = sweep_{write})) \\
& \text{then} \\
& \quad \text{val } address = dest + rwindex \text{ in} \\
& \quad buffer_{word}, buffer_{ptr} \leftarrow mem[address]_{word}, mem[address]_{ptr} \\
& \text{else} \\
& \quad \text{val } address = dir[pointer_{handle}]_{addr} + rwindex \text{ in} \\
& \quad buffer_{word}, buffer_{ptr} \leftarrow mem[address]_{word}, mem[address]_{ptr}
\end{aligned}
$$

When writing a pointer to memory during the marking process, the tuple addressed by the pointer must be marked and added to the $next$ list. Otherwise it would be possible for a tuple to remain unmarked because the pointer to it is moved by the executing program from a tuple that has not yet been marked into one that has already been marked. Also, if a pointer is written to a location in a tuple, the tuple must be marked by setting its $deep$ indicator to ensure that its contents are scanned during the marking process.

$write$: val $rwindex \ = \ pointer_{offset} + offset + 1$ in
  if $(pointer_{handle} = tuple) \wedge (rwindex = index) \wedge$
    $((state = sweep_{clear}) \vee (state = sweep_{write}))$
  then $copy_{word}, copy_{ptr} \ \leftarrow \ buffer_{word}, buffer_{ptr}$
  else
  if $(pointer_{handle} = tuple) \wedge (rwindex < index) \wedge$
    $((state = sweep_{read}) \vee (state = sweep_{clear}) \vee (state = sweep_{write}))$
  then
    val $address = dest + rwindex$ in
    $mem[address]_{word}, mem[address]_{ptr} \ \leftarrow \ buffer_{word}, buffer_{ptr}$
  else
    val $address = dir[pointer_{handle}]_{addr} + rwindex$ in
    $mem[address]_{word}, mem[address]_{ptr} \ \leftarrow \ buffer_{word}, buffer_{ptr}$

  if $buffer_{ptr}$
  { if $(\neg dir[buffer_{handle}]_{mark}) \wedge$
      $((state = mark_{scan}) \vee (state = mark_{add}) \vee (state = mark_{next}))$
    then
    { $dir[buffer_{handle}]_{mark}, \ \leftarrow \ true$
    & $livesize \ \leftarrow \ livesize + dir[buffer_{handle}]_{size} + 1$
    & $dir[buffer_{handle}]_{list}, next \ \leftarrow \ next, buffer_{handle}$
    }
    else skip
  & $dir[pointer_{handle}]_{deep} \ \leftarrow \ true$
  }

An experimental processor

The processor described here is a very simple design incorporating a memory allocation system as described above. It has been used in conjunction with a simple programming language and self-hosting compiler. The initial results of this suggest that in many practical situations, the garbage collector will be able to retrieve unused memory space as fast as new space is allocated.

The main features of the instruction set are:

- Short instructions are provided to allow efficient access to the stack and other data regions allocated by compilers; these also provide efficient branching and subroutine calling. The short instructions have been chosen so as to provide efficient program representation when used by modern compilers.
- Instructions are provided to allocate memory regions of size specified by an instruction operand.
- An address consists of two parts. One specifies a memory region; the other specifies a location within the region. The regions are byte addressed and the instructions are all single byte. However, most of the memory access instructions are designed to access words.
- The processor checks that the instruction operands are appropriate to the operation; for example, one operand of a memory access instruction must be an address and the other must be a value to be used as an offset. The processor checks that the resulting address lies within the specified region.
- The same instruction set can be used for processors with different wordlengths; the only requirement is that the wordlength is a number of bytes.
- The processor has a small number of registers. Some registers are used for specific purposes such as accessing the stack or building large constants.
- Instructions are easy to decode.

All instructions are 8-bit; each instruction contains 4 bits representing an operation and 4 bits of immediate data. A special instruction, OPR causes its operand to be interpreted as an inter-register operation. Instruction prefixes are used to extend the range of immediate operands and to provide more inter-register operations.

The prefixes, which are inserted automatically by compilers and assemblers, are:

- PFIX which concatenates its 4-bit immediate with the 4-bit immediate of the next 8-bit instruction.
- NFIX which complements its its 4-bit immediate and then concatenates the result with the 4-bit immediate of the next 8-bit instruction.

The state of the processor is represented by 5 registers. Two of these are operated as a stack and used to hold the sources and destination of inter-register operations.

register	**use**
pc	the program counter
sp	the stack pointer
$oreg$	the operand register
$areg$	the first register in the operand stack
$breg$	the second register in the operand stack

Instruction Issue and Execution

The processor core is intended to be implementable without a pipeline to maximise responsiveness; this potentially allows a very simple design.

The instructions are all 8-bit, so that on a 32-bit implementation four instructions are fetched every cycle. Typically less than 40% of instructions require a memory access, so it is practical to support the processor using a unified memory system.

Each processor has a short instruction buffer which is one word long. The rules for performing an instruction fetch are as follows:

- Any instruction which requires data-access performs it during the memory access stage.
- Branch instructions fetch their branch target instructions during the memory access stage.
- Any other instruction (such as ALU operations) performs an instruction fetch if it is the last instruction in the instruction buffer.
- If the instruction buffer is empty when an instruction should be issued, a special *no-op* is issued; this will load the instruction buffer.

Instruction set Notation and Definitions

In the following description

Bpw	is the number of bytes in a word
bpw	is the number of bits in a word
pc	represents the program counter
sp	represents the stack pointer
$oreg$	represents the operand register
$areg$	represents the first stack register
$breg$	represents the second stack register
$u4$	is a 4-bit unsigned source operand in the range $[0:15]$
$tuple[n]$	is word n of tuple $tuple$
$base \ @ \ offset$	is a pointer to a location $offset$ bytes from $base$

Data access

The data access instructions fall into several groups. One of these provides access via the stack pointer.

LDWSP	$areg, breg \leftarrow sp[oreg], areg$	load word from stack
STWSP	$sp[oreg], areg \leftarrow areg, breg$	store word to stack
LDAWSP	$areg, breg \leftarrow sp \ @ \ oreg_{\times Bpw}, areg$	load address of word in stack

Access to constants and program addresses is provided by instructions which either load values directly or enable them to be loaded from a location in the program:

LDC	$areg, breg \leftarrow oreg, areg$	load constant
LDAP	$areg, breg \leftarrow pc \ @ \ oreg, areg$	load address in program
PBASE	$areg, breg \leftarrow pc_{handle}, areg$	load pc handle

Access to data structures is provided by instructions which combine an address with an offset:

LDWI	$areg \leftarrow areg[oreg]$	load word
STWI	$areg[oreg] \leftarrow breg$	store word
LDAWI	$areg \leftarrow areg \ @ \ oreg_{\times Bpw}$	load address of word

Expression evaluation

SWAP	$areg, breg \leftarrow breg, areg$	swap top stack values
ADDC	$areg \leftarrow areg + oreg$	add constant
ADD	$areg \leftarrow breg + areg$	add
SUB	$areg \leftarrow breg - areg$	subtract
WSUB	$areg \leftarrow breg \mathbin{@} areg_{\times Bpw}$	form word address
EQC	$areg \leftarrow areg = oreg$	equal constant
EQ	$areg \leftarrow breg = areg$	equal
LSS	$areg \leftarrow breg <_{sgn} areg$	less than signed
AND	$areg \leftarrow breg \wedge areg$	and
OR	$areg \leftarrow breg \vee areg$	or
XOR	$areg \leftarrow breg \oplus areg$	exclusive or
NOT	$areg \leftarrow -1 \oplus areg$	not
SHL	$areg \leftarrow breg \ll areg$	logical shift left
SHR	$areg \leftarrow breg \gg areg$	logical shift right

Branching, jumping and calling

The branch instructions include conditional and unconditional relative branches. A branch using an offset in the stack is provided to support jump tables.

BR	$pc \leftarrow pc \mathbin{@} oreg$	branch relative unconditional
BRF	if $\neg areg$ then $pc \leftarrow pc \mathbin{@} oreg$	branch relative false
BRX	$pc, areg \leftarrow areg, breg$	branch absolute

The procedure calling instruction uses a program address in the stack to determine a subroutine entry point, leaving the return address on the stack. The RET instruction can also be used simply to branch to a program address on the stack.

CALL	$sp[0], pc, areg \leftarrow pc, areg, breg$	call subroutine
RET	$pc \leftarrow sp[0]$	return from subroutine

Typically, the stack is initialised at the start of program execution. It can be extended on subroutine entry and contracted on exit using the LDAWI instruction. It is also possible to allocate a new tuple on subroutine entry using a GETM instruction; The ENTER and EXIT instructions can then be used to move the stack pointer to and from the new tuple.

SETSP	$sp, areg \leftarrow areg, breg$	set stack pointer
ENTER	$sp, areg[1] \leftarrow areg, sp$	switch to new tuple
EXIT	$sp \leftarrow sp[1]$	switch from new tuple

Tuples

A new tuple is created using one of the GETM instructions. The tag is initialised when the tuple is created. It can be accessed using the TAG instruction. The number of words in the tuple can be obtained using the SIZE instruction. There is a special tuple nil of size 0 which differs from all other tuples.

GETMI	$areg \leftarrow$ array $areg : oreg$	create tuple immediate
GETM	$areg \leftarrow$ array $areg : breg$	create tuple
TAG	$areg \leftarrow$ tagof $areg$	read tag
SIZE	$areg \leftarrow$ sizeof $areg$	read size
NIL	$areg, breg \leftarrow nil, areg$	load nil tuple

Registers and Marking

The marking process starts from the registers; if a register holds the address of a tuple, the tuple is marked. For the simple instruction set described here, it is sufficient to mark the tuples addressed by sp and pc because any tuple created will be marked when it is created and will be stored in memory before the next garbage collection cycle starts.

For a processor with a register file, it is possible that the address of a newly created tuple will remain for some time in a register instead of being stored in memory. This means that the register contents would have to be available to the garbage collector at the start of each garbage collection cycle.

If the garbage collector is implemented in a memory subsystem, it would be possible for the memory subsystem to store the addresses of all created tuples between the time they are created and the time when they are stored in memory; these stored addresses could be used to mark the tuples at the start of each garbage collection cycle. Alternatively, the GETM instructions could be designed to store each newly created tuple address in a memory location.

Instruction summary

LDWSP	$areg, breg \leftarrow sp[oreg], areg$	load word from stack
STWSP	$sp[oreg], areg \leftarrow areg, breg$	store word to stack
LDAWSP	$areg, breg \leftarrow sp \mathbin{@} oreg_{\times Bpw}, areg$	load address of word in stack
LDC	$areg, breg \leftarrow oreg, areg$	load constant
LDAP	$areg, breg \leftarrow pc \mathbin{@} oreg, areg$	load address in program
LDWI	$areg \leftarrow areg[oreg]$	load word
STWI	$areg[oreg] \leftarrow breg$	store word
LDAWI	$areg \leftarrow areg \mathbin{@} oreg_{\times Bpw}$	load address of word
ADDC	$areg \leftarrow areg + oreg$	add constant
EQC	$areg \leftarrow areg = oreg$	equal constant
BR	$pc \leftarrow pc \mathbin{@} oreg$	branch relative unconditional
BRF	if $\neg areg$ then $pc \leftarrow pc \mathbin{@} oreg$	branch relative false
GETMI	$areg \leftarrow$ array $areg : oreg$	create tuple immediate
SWAP	$areg, breg \leftarrow breg, areg$	swap top stack values
ADD	$areg \leftarrow breg + areg$	add
SUB	$areg \leftarrow breg - areg$	subtract
WSUB	$areg \leftarrow breg \mathbin{@} areg_{\times Bpw}$	form word address
EQ	$areg \leftarrow breg = areg$	equal
LSS	$areg \leftarrow breg <_{sgn} areg$	less than signed
AND	$areg \leftarrow breg \wedge areg$	and
OR	$areg \leftarrow breg \vee areg$	or
XOR	$areg \leftarrow breg \oplus areg$	exclusive or
NOT	$areg \leftarrow -1 \oplus areg$	not
SHL	$areg \leftarrow breg \ll areg$	logical shift left
SHR	$areg \leftarrow breg \gg areg$	logical shift right
BRX	$pc, areg \leftarrow areg, breg$	branch absolute
CALL	$mem[sp], pc, areg \leftarrow pc, areg, breg$	call subroutine
RET	$pc \leftarrow mem[sp]$	return from subroutine
PBASE	$areg, breg \leftarrow pc_{handle}, areg$	load pc handle
SETSP	$sp, areg \leftarrow areg, breg$	set stack pointer
ENTER	$sp, areg[1] \leftarrow areg, sp$	switch to new tuple
EXIT	$sp \leftarrow sp[1]$	switch from new tuple
GETM	$areg \leftarrow$ array $areg : breg$	create tuple
TAG	$areg \leftarrow$ tagof $areg$	read tag
SIZE	$areg \leftarrow$ sizeof $areg$	read size
NIL	$areg, breg \leftarrow nil, areg$	load nil tuple