ADAM Architecture Specification

by

Andrew “bunnie” Huang

October 1, 2001

Revised February 19, 2002

DRAFT

Contents

List of Figures

List of Tables

Chapter 1

Aries Decentralized Abstract Machine Specification

The Aries Decentralized Abstract Machine (ADAM) is a massively parallel architecture that relies on the tight integration of processors and memory made possible by embedded memory fabrication processes. The ADAM is a large, homogeneous machine that is easy for programmers and optimizing compilers to target, but perhaps impractical to build. ADAM is intended to be emulated by a concrete machine with a similar architecture that may not be homogeneous and may change with time, and may vary in capability and capacity from machine to machine. Thus, ADAM is a bridge between the simplicity and elegance desired by software developers and the reality of budgets, yield, and change faced by the hardware engineers.

1.1 Overview

ADAM consists of an arbitrarily (anywhere between one and one million) sized fat-tree of simple processor-memory nodes. A single node of ADAM is similar to RISC architectures such as the Alpha and the ARM. Its features include:

- single-issue, in-order execution instruction semantics
- 128 entry 80-bit wide queue file (64-bit data types, 16-bit tags)
- separate code, environment and data address spaces
- named-state processor context management
- support for continuation-style threads
- inter-thread communication via queue remapping
- hardware assisted capability-based memory addressing
- hardware assisted data tagging
- method-signature tagged instruction memory

- native support for object-oriented programming languages
- spatially aware addressing model
- support for fast data migration
- management coprocessor for thread scheduling, performance tuning, and debugging
- no need for data caches; therefore, no aliasing-related coherence problems

Without recompilation, any program targeted for the ADAM platform can run on any scale of implementation, within the limit that the virtual memory of the implementation is not exhausted. It is also a goal of the ADAM platform that most large programs, without recompilation, will scale well in performance as the implementation scales to more nodes.

The ADAM node is intended to be implemented in an embedded memory process. Embedded memories have a latency and bandwidth performance on par with the caches of a typical processor; hence, no data caches are allowed in the ADAM specification. This alleviates a host of aliasing-related coherence issues. Embedded memories are also small, but it is the intention that even the smallest ADAM implementations should have thousands of nodes, so that the overall amount of fast memory is big enough to hold the desired programs in-core. In the case that the embedded memory is exhausted, secondary DRAM storage and tertiary hard drive storage is provided through a virtual memory mechanism.

1.2 Programming Model

1.2.1 Threads

The fundamental unit of computation in ADAM is a thread. Threads are very lightweight under ADAM, and they are opaque, monolithic memory structures. They could almost be called continuations except that they carry just a little more state than a program counter and an enviroment pointer.

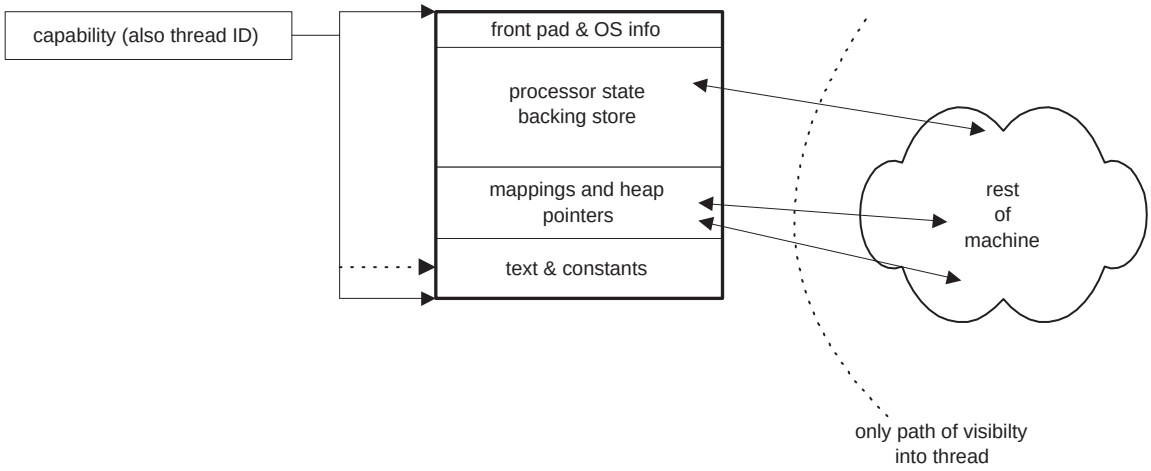

Figure 1-1: Structure of an ADAM thread

In place of registers in a typical machine, ADAM supplies queues of an unspecified depth. These queues can assume register semantics when necessary via a copy/clobber modifier. This modifier causes dequeues to be copies on reads, and enqueues to be clobbers on writes. The tail of any queue can be remapped onto the head of another queue in another thread context. Note that using a clobber operator on a remapped queue yields unpredictable results since the victim data depends upon how long it takes for the clobbering data to arrive at the destination.

Data from any given source is guaranteed to arrive in-order in the destination context's queue; however, when more than one sender is mapped to a single receiver, there is no guarantee as to the ordering of the received values between the two senders. A node can request that the source ID of incoming data be enqueued in a secondary queue in lock-step with the primary destination queue, so that ambiguity created by such a situation can be resolved by user code. The only allowed method of inter-thread communication is via these queue mappings. While a programmer can communicate data between threads by passing around heap-allocated data structures, it is not recommended because ADAM has a relaxed consistency memory model with no barrier or inter-processor memory synchronization instructions. The only guarantee is that all reads issued after a write to the same address will return with the most recently written data by the the issuing thread if and only if another thread has not modified that address in the meantime.

To a first approximation, the queues supplied by ADAM are of infinite depth. In a realistic implementation, the performance of the queues drops off as more data is shoveled into them. It is recommended that a compiler or hand-assembly coder try to minimize the use of *unmapped* queues as storage elements, since the performance of a hardware implementation may degrade rapidly if queues are used extensively as deeper-than-1 storage elements. For light-duty operation, as typically encountered when holding temporary results within a computation, the queues should perform as fast as a register in a standard RISC machine. However, as a communication element between streaming threads, some flow control mechanism is required, and the queues apply a back-pressure proportional to their fullness. The implementation and implications of this back-pressure in an ADAM implementation are discussed later.

The address space of the machine is divided into three parts: code, environment, and data. ADAM execution units cannot directly affect values in any address space. The code space is write-once, read many and data is striped across all nodes; interaction between code space and user space is possible only through the LDCODE opcode. Environment and data spaces are read-many, write-many and their address spaces are local to each node. Environment space is where thread contexts are stored; thus, all interaction with environment space is implicit. Data space accessed only through queue mappings in the execution unit. A memory management coprocessor is required to handle memory requests in memory space. Figure 1-2 illustrates the high level situation that leads to this division of address spaces.

There is no stack in ADAM; rather, arguments and return values are passed through the queue file via queue mappings. The backing store for the named-state queue file occupies the top part of the environment segment for the thread. All visibility into and out of the thread occurs via a set of queue mappings that are

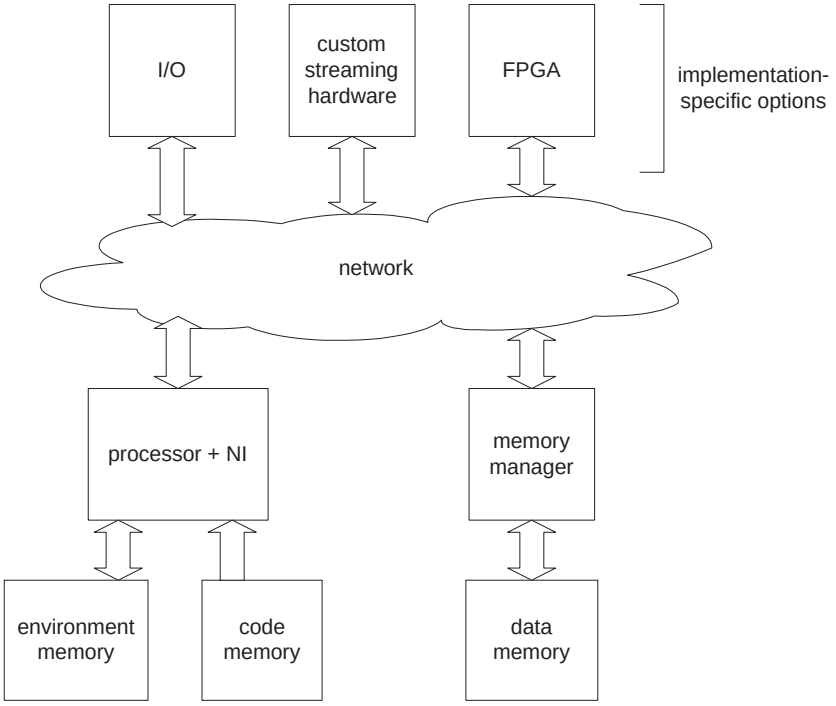

Figure 1-2: Pieces of an ADAM implementation. Node ID tags are uniform across the machine, so network-attached custom hardware is addressible like any processor or memory node.

bidirectional.

There is one reserved thread ID on each processor node known as the kernel capability. This capability is not accessible from any ADAM instructions, and is setup at initialization time by the management coprocessor. An exception causes machine execution to halt and control to be transferred to the kernel capability.

1.2.2 Data Types

All ADAM data types are 80 bits wide; they consist of a 64 bit data field and a 16 bit tag field. Four integer data types are supported: signed long (referred to as "word"), packed signed integer, packed signed short, and packed unicode characters. Only one floating point data type is supported, similar to the IEEE-754 double format. See figure 1-4 for detailed bit-level formatting of the data types.

Packed data is operated on in vector form; most arithmetic operations are supported on packed data. Any arithmetic operation involving a capability, however, is only valid with a long. Any integer type is supported for memory queue offsets, however. Please see section 1.2.4 for more information on the ADAM memory model.

All data types are fully tagged to identify their type, as well as any flags associated with their status. See figure 1-5 for details. Errors on arithmetic operations can be forced to be trapping and non-trapping. Trapping errors cause the thread to deadlock and an exception to be thrown; non-trapping errors allow execution to proceed normally (which may or may not imply deadlock) and the error condition to simply be noted in the

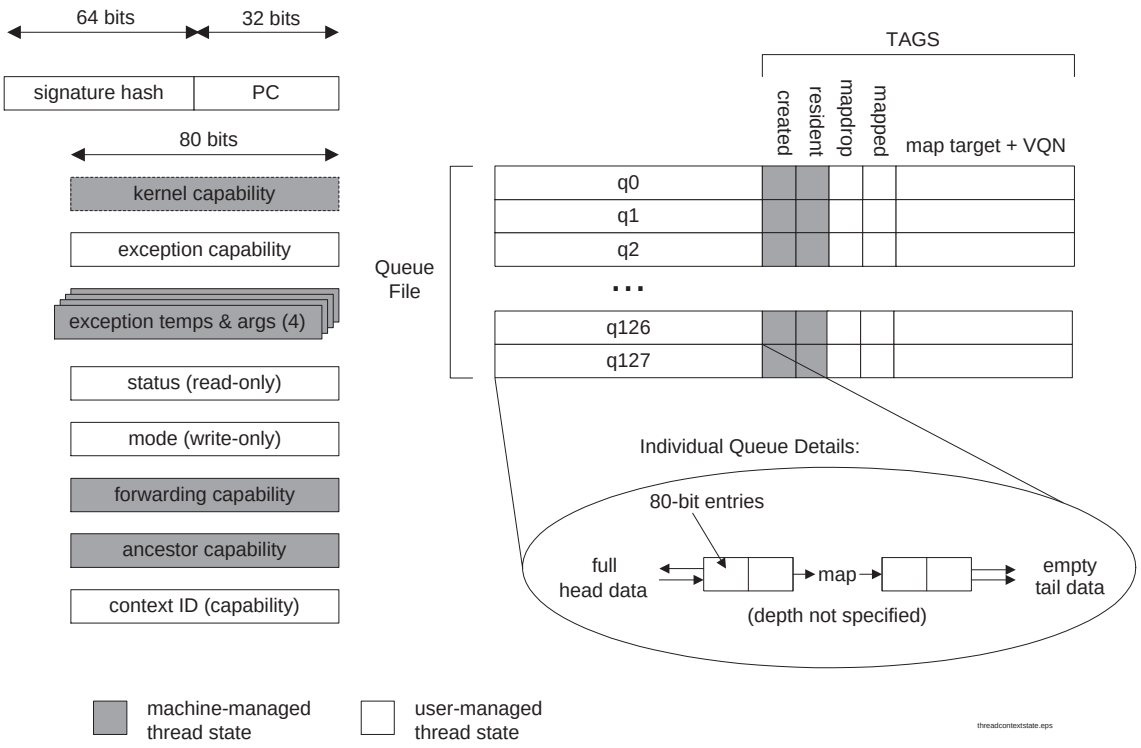

Figure 1-3: Programming model of ADAM

result's tag and flags field.

An immutable bit is also included in the tag field, to indicate static data that cannot be altered. Identifying data as static allows management routines to copy that piece of data freely, thus enabling cheap automatic mechanisms for distributing frequently referenced values. Writing to data that is declared as immutable has no effect on the data, may throw an exception, and always sets a bit in the status register to indicate that an illegal write occurred.

A primary bit is also included in each tag field that is used by the data migration manager to indicate if this is the primary copy of the data. This is particularly useful for the scenario of partial migration, where the primary capability containing some data has migrated but the data itself has yet to move. See chapter **??** for more details.

A subset of the IEEE 754-1985 floating point standard is required by ADAM architecture. The differences between the IEEE 754-1985 standard and the ADAM format are chosen to simplify implementation and enhance performance with a small reduction in precision. These differences are:

- no support for single-precision floats and its associated operations and conversions, with the exception of constant fields in opcodes
- NaN and $\pm\infty$ are specified in the tag field, so exponent = 2047 is now valid, and the exponent bias is now +1024
- no denorms (accuracy versus IEEE 754-1985 reclaimed by previous bullet point)

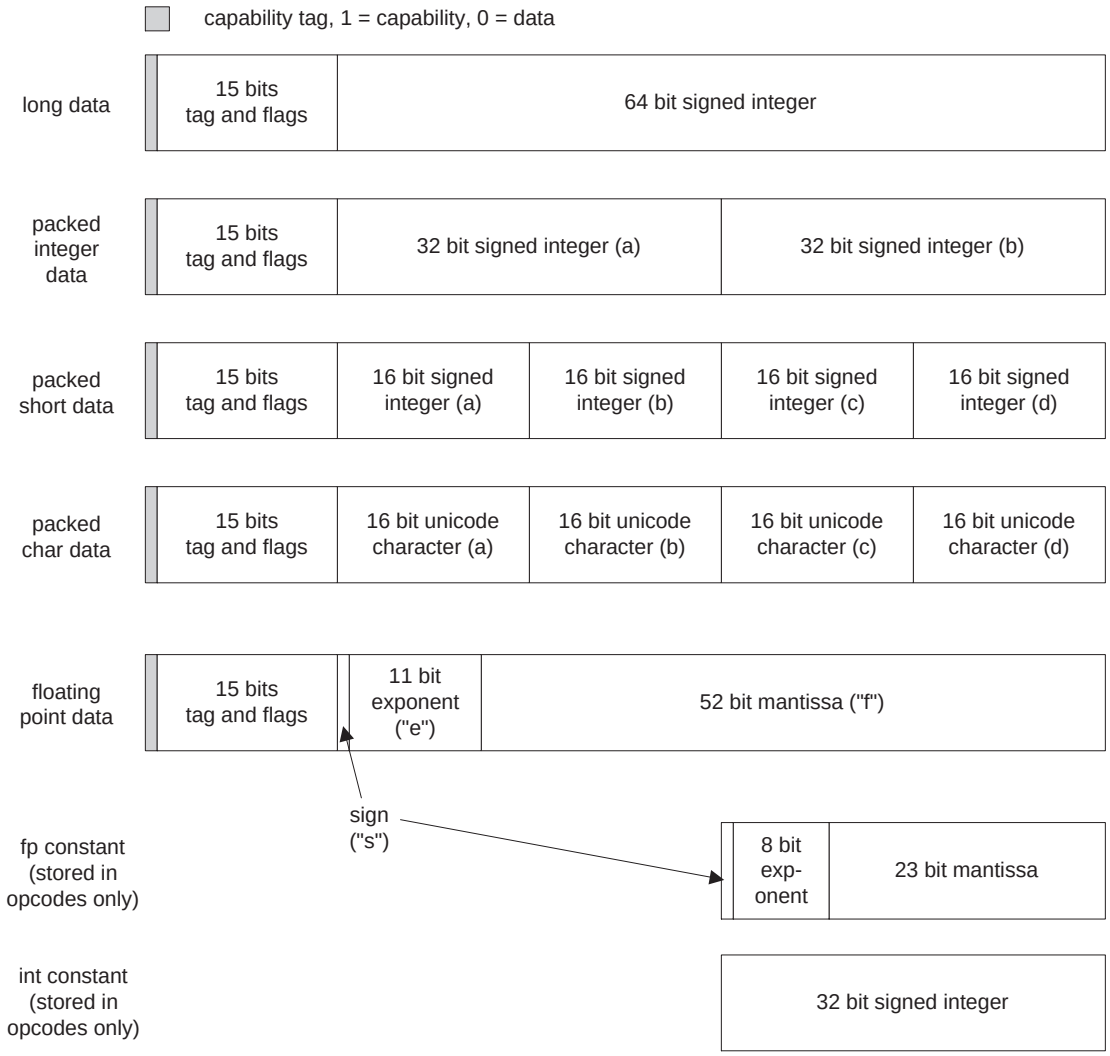

Figure 1-4: Data formats supported by ADAM

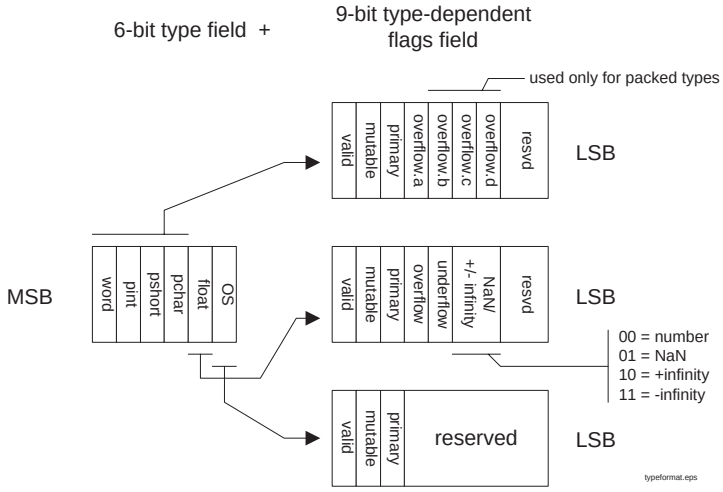

Figure 1-5: Tag and Flag field details

- one rounding mode: von Neumann style rounding

To summarize, the value of the floating point number is $v = (-1)^s 2^{e-1024}(1.f)$ unless $e = 0$ and $f = 0$, in which case the value is $v = (-1)^s 0$ (signed zero).

Aside from these, the ADAM floating point format defers to the IEEE 754-1985 standard [Ste85]. In particular, the handling of NaNs, Infinity, and Signed Zero in the context of Exceptions, Traps, Comparisons and Conversions are identical.

The ADAM instruction format allows for 32-bit constants to be stored in a standard opcode. Floating point instructions can thus store a single-precision format float in the constant field, but this is immediately converted to a double-precision number upon use. The single precision floats likewise do away with the denorm representation; hence, NaNs and $\pm\infty$ are not representable in the single-precision floating point constant field. The value of a single precision floating point number is $v = (-1)^s 2^{e-128}(1.f)$ unless $e = 0$ and $f = 0$, in which case the value is $v = (-1)^s 0$ (signed zero).

von Neumann style rounding is implemented by adding an LSB of precision to floats as they enter the arithmetic pipeline and carrying this LSB of precision throughout the pipe. This extra LSB is set to a binary "1" as numbers enter the pipe, and rounding is done by simple truncation at the end of the pipe. This results in an expected value of the extra LSB to be $\frac{1}{2}$ at the end of the day.

An implementation may choose use to full IEEE 754-1985 style rounding to gain the extra precision, but there is no provision in the stock architecture specification to choose which rounding mode to use; the default and only rounding mode should thus be "round to nearest" per IEEE 754-1985.

1.2.3 Instruction Formats

ADAM has a sequestered code space. The code space, unlike the data and environment spaces, is global and shared among all nodes; this is feasible because the code space is mostly read-only. The management coprocessor takes care of handling any page faults or the loading and unloading of code in code space. ADAM can dynamically request new object classes to be loaded into code space with the `LDCODE` instruction.

The code space is mostly read-only because some instructions contain hint fields to the instruction prefetcher. The actual values contained in the hint fields are implementation-dependent and any ADAM implementation must execute code correctly regardless of the hint field's contents; however, a compiler is free to warm up the hint fields with bit patterns that may improve start-up performance for a specific implementation. Because there is no correctness impact upon code execution if the hint fields are wrong, instruction caches can replace lines that have not been written back due to a lack of instruction bandwidth. Likewise, write values do not have to propagate throughout the system even though the code is globally shared among all nodes. However, in the case that values do make their way back to their original file on disk, the next time code is loaded, it may run faster.

Instructions are 64 bits long and have four basic formats: standard, branch, jump, and hint (see 1-6). Every instruction has an 8-bit opcode field. Every queue specifier in every instruction is modified by a copy/replace

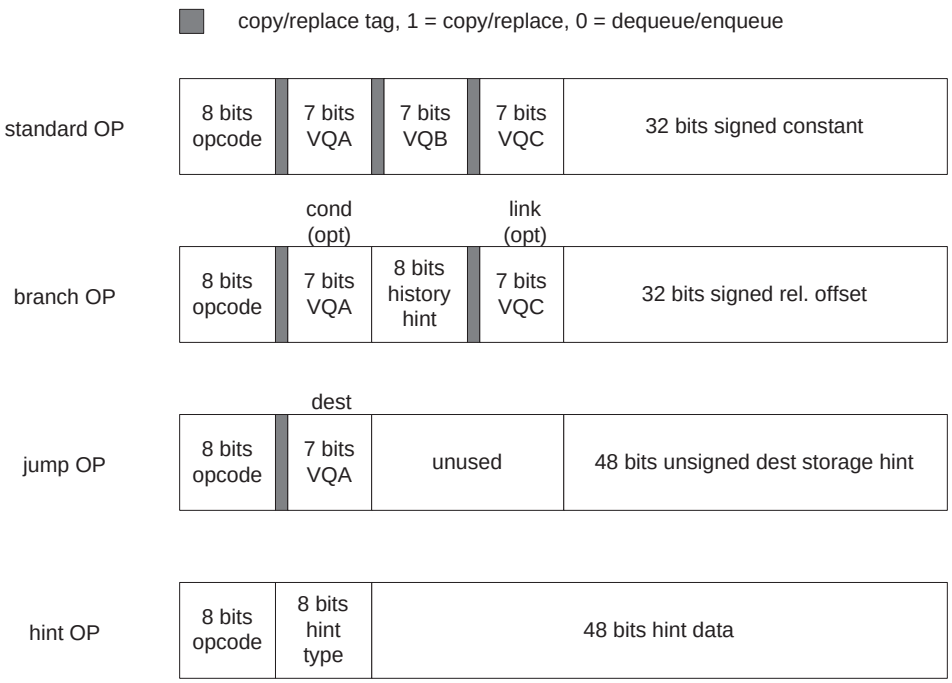

Figure 1-6: Format of ADAM opcodes

bit. Setting the copy/replace tag enables the compiler to treat the queue with semantics similar to that of a register. A copy operation extracts a value from a queue without changing any of the values in the queue; a replace operation tests to see if a queue is empty, and if it is, waits until a value is written to it, and then replaces the value. The replace operation is invalid on a remapped queue and attempting to perform such an operation triggers an exception.

The standard instruction has three virtual queue specifiers, each 7 bits long. The first two (VQA and VQB) specify read queues; the final (VQC) specifies the write queue. The standard instruction also contains a 32-bit signed constant field, thus allowing the standard instruction to specify up to three data sources and one data destination, although most instructions do not take full advantage of this capability.

Certain instructions, known as special-format instructions, may interpret the VQA, VQB, or VQC fields as constants instead of as a queue to reference to extract or store data to the queue file. These instructions typically deal with the creation, maintenance and destruction of queue maps. Since the compiler and/or assembly language programmer typically knows at all times the exact queue number that a mapping is applied to, it does not make sense for most queue map maintenance instructions to accept arbitrary dynamically generated queue values. Hence, the VQA, VQB, and VQC fields can be used to immediately refer to a queue number for these instructions.

Branch instructions have a condition field, a link field, a branch history hint field, and a 32-bit signed branch offset. Either the condition or the link field may be omitted from an instruction, but not both. An 8-bit history hint field is also provided so that a branch history can be stored with the branch instruction. Note that

the format of the hint field is implementation-specific, and that any ADAM implementation must function correctly regardless of the hint field contents.

Jump instructions have a destination field and a 32 bit unsigned jump destination hint. Only the lower 32 bits of the value in the queue specified by the jump destination field is loaded into the program counter. The jump destination hint field is provided so that an implementation can memoize the most recent jump address. Note that the format of the hint field is implementation-specific, and that any ADAM implementation must function correctly regardless of the hint field contents.

Hint instructions are no-ops that provide hints to the runtime system. The hint may or may not be platform dependent; this information is encoded within the hint type field. Examples of hints are data placement directives, prefetch directives, and thread yield directives. Hints that are not recognized by the run-time are ignored.

Please consult the appendix for detailed listing of the instructions supported by ADAM and their descriptions.

1.2.4 Memory Model

The ADAM uses a virtually addressed capability-based data memory model with memory striped across the machine using an explicit node ID as part of the address. The node ID field and address field can steal bits from each other depending upon the implementation; for the sake of concreteness, figure 1-7 uses some "typical" field sizes. Node location coding within the address has been seen in the parallel language Sather. The actual translation of the virtual addresses and paging mechanisms are transparent to the specification and implementation-specific. A summary of the capability format can be seen in figure 1-7. The ADAM has no load or store instructions; instead, data memory is an opaque object accessed only through queue mappings.

The capability format used by ADAM allows for exact base and bounds determination from an arbitrary capability with the use of front-padding to eliminate a small amount of rounding overhead. The total padding penalty incurred by the capability format is bounded to be less than 11.2%. Please refer to [BGKH00] for a detailed discussion of the capability format; only a summary of the capability format and properties are given here.

The method for extracting the base and bounds from a front-padded capability is as follows, written in pseudo-code:

Warning: the psuedocode for capability arithmetic is known to be wrong. It is, however, implemented correctly in the simulator. Updates pending...

```
B = block size field value
L = length field value
F = finger field value
A = address field value

if( B == 63 ) {
```

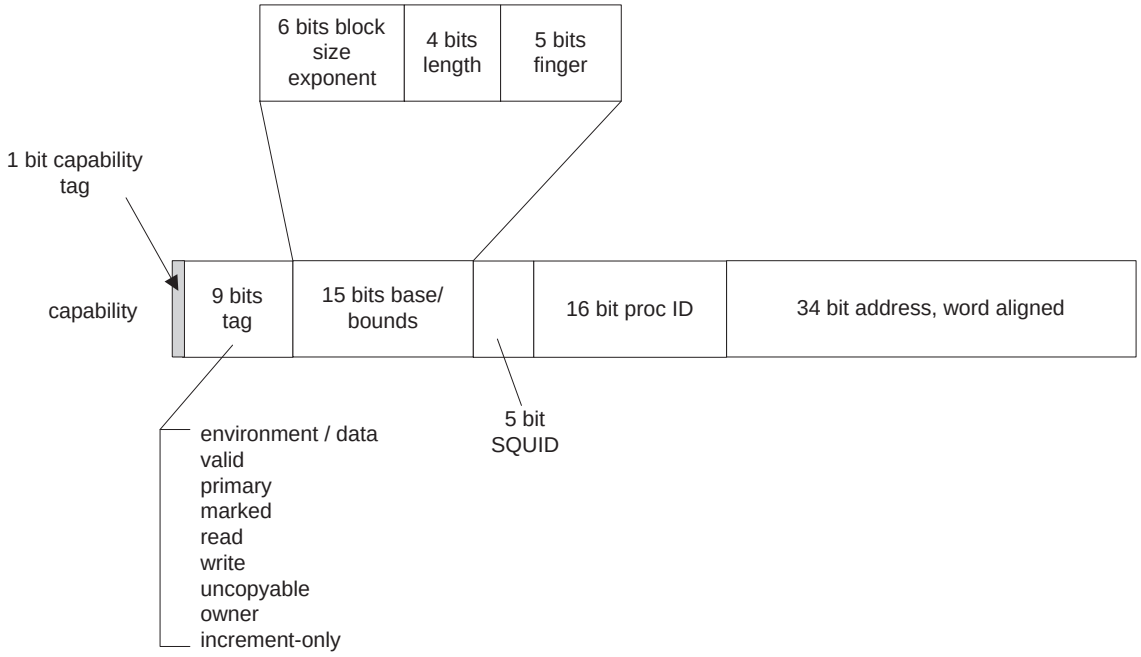

Figure 1-7: ADAM capability format

```
    // L, B are immutable
    // A and F are updated by capability arithmetic ops,
    // with check made to ensure that F < L
    capability.length = L + 1;
    if( F ≥ L ) {
      throw capability bounds exception
    }
    capability.beginning = A − F;
    capability.end = A − F + L + 1;
    desired data = *A;
  }
  else {
    // & is the bitwise AND operator
    capability.beginning = (A & (2^B − 1)) − (F ≪ B)
    capability.length = (L + 17) ∗ 2^B
    capability.end = capability.beginning + capability.length;
    desired data = *A;
  }
```

The only valid operations on a capability are addition and subtraction. The new address that results from an arithmetic operation is simple to calculate:

```
X = signed integer offset to be added
A_2 = new address
A_1 = old address
```

A_2 = A_1 + X

The method for recalculating the finger field of a capability that has had an arithmetic operation on it is as follows, written in pseudo-code with verilog bitfield syntax:

F_2 = original finger field
F_1 = new finger field
X = signed integer offset to be added

B = value of the block length field
$F_2 = F_1 + X[63 : B] + (X[B-1 : 0] + A_1[B-1 : 0]) \gg B$

The value of the new finger field should be less than the value of the length field but greater than zero; if not, an error should be flagged. An efficient hardware implementation of the above calculation is given in [BGKH00]. Note that capabilities cannot be dynamically resized. This implies that the length and block size fields should never change after an arithmetic operation. In order to grow a capability, a new one must be created and the contents of the old one copied into the new one.

The ADAM capability format contains an explicit processor node ID embedded within the address field of the capability. The size of the node ID field allows for up to one million processors to be present in the system, but the actual allocation of capabilities on these nodes is left up to the operating system. Because capabilities are opaque to the programmer and the allocation process is implementation-specific, all ADAM applications can run on implementations with anywhere between one and one million nodes, with no requirement on the distribution of node IDs. Valid node IDs can even change dynamically, so long as the OS is careful to ensure that a node is empty before deactivating its ID. This can be useful in situations where environmental monitors detect an impending failure, or where users wish to hot-swap nodes to perform upgrades or service. Note that the amount of available memory for applications to run does vary with the number of nodes in the system, but the address space is fairly large so users should rarely encounter this situation.

The capability format also includes a number of bits for memory management and security purposes. These bits are:

- environment/data: indicates if the capability is for environment space or for data space. Normally this bit should not be modified after capability creation.
- increment-only: indicates that only arithmetic operations that result in offsets that result in an address greater than the current address are valid
- valid: indicates if a capability is valid. An attempt to dereference an invalid capability results in a protection fault.
- marked: used for garbage collection
- read: indicates that data can be read from the capability.
- write: indicates that data can be written to the capability.
- uncopyable: indicates that only dequeue operations are allowed on the capability; an attempt to copy the capability will result in an exception being raised.

- owner: when the owner bit is set, the read, write, and uncopyable bits can be overridden.
- primary: indicates that this capability is the primary working copy. For capabilities in data space, it marks the endpoint of a migration list. For capabilities in environment space, it also marks a thread with this bit set as the only runnable copy.
- SQUID: **S**hort **Q**uasi-**U**nique **ID**. A short tag field that contains a randomly generated ID number assigned at the time of capability allocation; when a capability is migrated, this field is directly copied. Use of this field reduces the cost of capability inequality comparisons.

There are no load or store instructions in the ADAM specification. The memory is an opaque object accessed only through queue mappings. The `MML` and `MMS` opcodes are used to initiate load and store queue pairs, respectively. `MML` takes an outgoing address queue and a return data queue as arguments; `MMS` takes an outgoing address queue and an outgoing data queue as arguments. The ordering of data in any single given load or store queue mapping is guaranteed to be preserved; address and data occur in lock-step. However, the ordering between mappings is not guaranteed; hence, sharing a single queue for multiple memory queue maps is not recommended as it results in nondeterministic behavior.

Locks and semaphores can be implemented using a combination of the `MMS`, `MML`, and `EXCH` opcodes. The `EXCH` opcode declares a previously estabished store and load mapping as an *e*xchange pair. This exchange pair declaration causes any store issued into the store queue to automatically return the displaced value in the load queue. This exchange is guaranteed by hardware in the memory subsystem to be atomic. The timing of the exchange is not deterministic; the actual exchange on the memory location happens whenever the store request arrives at the destination memory location.

When initializing a memory queue, the first piece of data written into an address queue must be a capability or a memory access exception is raised. Subsequent accesses may pass more capabilities or any integer data type. When an integer data type is put into a memory queue, it is assumed to be an offset off of the most recent capability passed into the address queue. Putting a packed integer into an address queue causes data to be returned for each of the packed sub-values, starting with the least significant value end ending with the most significant value.

One feature that the memory queue access form enables is the user can extend the ADAM specification by add intelligence to the memory system. Capabilities and offsets are thrown into a memory queue, and the memory system is free to do what it likes before returning some data. Thus, the memory system can be augmented to be more than just a table of stored values.

As a note on implementation, the actual amount of fast memory available to each node is fairly small. The presumption is that the fast memory is managed like a cache with very large lines, and that the working set of any processor is kept small by migrating excess data to neighboring nodes. It is also presumed that hitting off-chip memory and ultimately the hard drive provides some kind of performance rolloff. While the processor is waiting for data to be swapped or moved around, the garbage collector and migration manager are run more frequently in an attempt to reduce the size of the working set and balance the load among

neighboring nodes.

1.2.5 Über-Capability

ADAM provides no supervisor mode or explicit kernel permissions in the style of Java. Initialization of ADAM is performed by a third-party operating system or boot monitor; the initialization process creates an über-capability that is the size of accessible memory and gives it to an initial thread. This initial thread is effectively the kernel thread, as it is responsible for managing memory and starting all child processes. Since ADAM is a virtual machine, multitasking on a single large machine is accomplished by dividing the machine into smaller groups of physical nodes and starting an ADAM per task, and each ADAM runs only one task. The equivalent of "nicing" a process on a machine running multiple ADAMs is dynamically varying the number of nodes allocated to an ADAM.

On power-up, each physical node starts code execution at location 0 in code space, and an über-capability is initially placed in q0. The über-capability is set to be the size of the entire virtual memory available for that node, and the owner bit is set. The über-capability effectively is the process ID for the node's OS, as it is responsible for allocating portions of the über-capability to user level tasks and devices, hence the ALLOCATE family of instructions interact with special code in the über-process. User threads cannot stomp on other threads because capabilities cannot be grown and base and bounds behavior is always enforced on dereferences.

1.2.6 Exception Handling

Exceptions on ADAM are inherently imprecise. ADAM is a distributed machine that runs many parallel threads. Hence, just as in the principle of relativity, the definition of simultaneity is blurred. ADAM's take on exceptions is two-pronged: first, the result of every exception-causing event is tagged; second, as much local state relevant to the exception is preserved at the instant an exception is detected.

When a thread is swapped in, an exception capability is included as part of its state. This exception capability is initialized on thread creation to point to a default exception handling object (usually an OS-defined object), and can be over-ridden by the user at any time. In the case that the user pointer is invalid, the processor falls back to the default handler. The first word of any exception handler object must be a value for the PC that corresponds to the object's server code.

An exception handler is expected to first inspect the processor status register to determine the source of the exception. The exception handler can determine which context caused the exception by examining the Exceptioned Context ID register, which contains a capability to the thread that caused the exception. A set of status bits within the status register are reserved for hardware exceptions. Having no hardware exception bits set indicates that a software exception was thrown, in which case a user-defined protocol is typically employed for communicating the exception type. A properly written OS handler should handle all exception

cases, with the ultimate exception being one that halts the program and makes debug information available. An improperly written OS handler can lead to unpredictable machine operation.

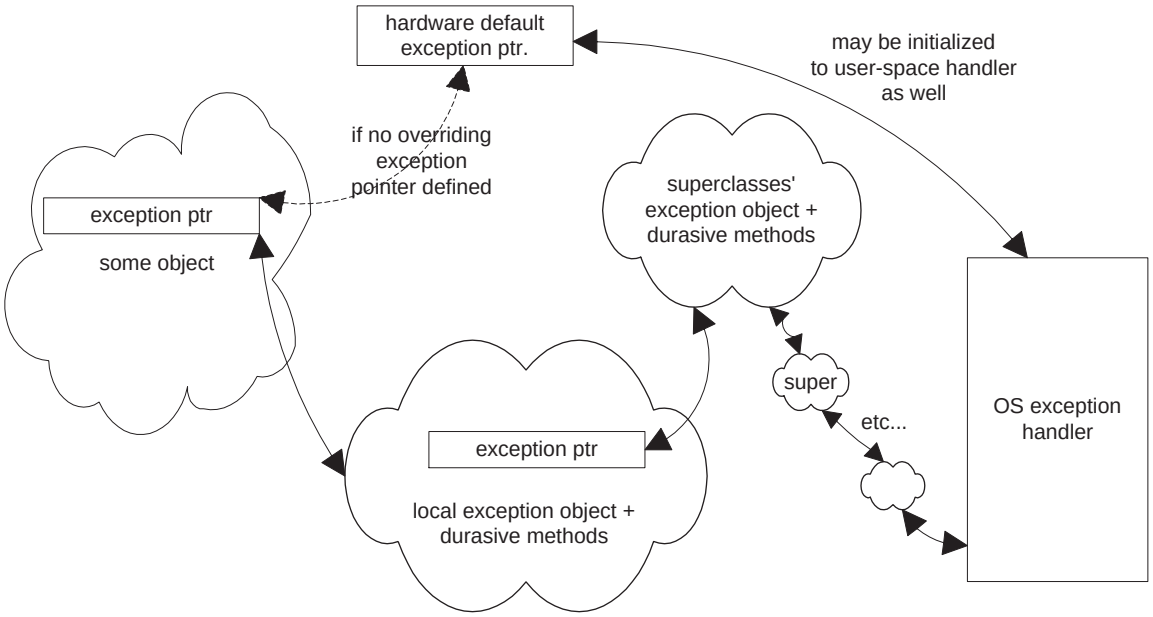

Figure 1-8: Exception handling overview

Exception handler objects are intended to be instantiated only once within a machine, and thus throwing an exception could be relatively slow because the handler is typically a remote object. Thus, exceptions are intended to be rare events, and users should avoid using the exception mechanism for anything other than exception handling. It is possible, however, that very popular run-time exception handlers could be migrated like any other object, since handlers are first-class objects.

Illegal opcode exceptions are handled in a different manner, similar to the Alpha architecture's PALcode. An illegal opcode dispatches into a look-up table in memory that has a hard-wired address, and control flow is transferred to an implementation-specific microcode processor that has access to all local state. The microcode processor could be as simple as a dedicated context ID on the ADAM plus instruction set extensions. The code that the microcode processor executes is stored in a reserved location in kernel memory. This allows for instructions implemented in future versions of the architecture to be emulated via software patches set up by the OS. During emulation mode, the processor behaves as if it had stalled, and errors during emulation mode lead to undefined behavior. It is recommended that the default behavior for an illegal opcode be an emulated `THROW` instruction.

1.2.7 Status and Mode Register

Things in the status and mode register include:

- exception masks (copy/clobber, overflow, div by 0, NaN, invalid type, capability access violation, immutable violation)
- exception status bits – has any of the above masking conditions occured?
- other mode bits (rounding modes, etc) ... will be defined later
- net traffic on thread-mapped queues since last checkpoint

Exceptions can either be trapping or non-trapping. Users can handle errors by just checking the tag bits on return data types and turn off all exception trapping if desired.

Things in the implementation register include (perhaps these are just directly controlled by the management processor):

- configuration (cache on/off)
- version
- temperature
- current consumption

1.2.8 Thread and Class File Format

Need to address stuff like:

- how to grow the heap and still keep the thread mobile (mini-page table in every capability!) – only use on threads that call `ALLOCATE` (not `ALLOCATEC`)
- status word at top of every thread to determine what type of thread it is
- storage of queue mappings
- storage of constants and text

1.3 Issues

also need to write up some ideas about transactional behavior and ways to implement them using the ADAM architecture (persistant FIFO's and pointer state).

memory subsystem and capabilities. two axes of decisions to make:

* move versus copy policy for capabilities - good for GC and management of data in machine - bad because every data access becomes a synchronize in a capability with embedded offset scenario, works okay with separate capability and offset streams * capability + offset or capability with embedded offset - embedded offset allows for easier pass-by-reference of data within a capability - capability + offset allows you to have a move policy for capabilities, and it allows you to do more with your capabilities (moves tags out of band)

what to do in the case of a dead child thread in the case of a suicide. usually parent kills child thread and child is GC'd; if child commits suicide and parent does not know this, does an access to the child

mean the parent dies? or does it just throw an exception? perhaps it should throw an exception that can programmatically lead to parent death.

Appendix A

Opcodes

A.1 General Notes

RTL descriptions of opcode operations are given in blocking form; i.e., the following lines of code

```
PC ← PC + 1
qc ← PC
PC ← PC + offset
```

stores the value of the initial `PC + 1` into `qc`, and the value of the initial `PC + 1 + offset` into `PC`.

Also, note that if no `PC` operation is specified, a default operation of `PC ← PC + 1` is implied, and that an exception can be thrown as a result of the `PC` increment if the `PC` enters into a protected or invalid code region.

A.2 Summary

Integer Arithmetic Instructions:

```
ADD qa, qb, qc
SUB qa, qb, qc
MUL qa, qb, qc
DIV qa, qb, qc
ADDC qa, n, qc
SUBC qa, n, qc
MULC qa, n, qc
DIVC qa, n, qc
```

Logical Operator Instructions:

```
AND qa, qb, qc
OR qa, qb, qc
XOR qa, qb, qc
ANDC qa, n, qc
ORC qa, n, qc
XORC qa, n, qc
SHL qa, qb, qc
```

```
SHR qa, qb, qc
SRA qa, qb, qc
SHLC qa, n, qc
SHRC qa, n, qc
SRAC qa, n, qc
```

Integer Comparison Instructions:

```
SEQ qa, qb, qc
SNE qa, qb, qc
SLT qa, qb, qc
SGT qa, qb, qc
SLE qa, qb, qc
SGE qa, qb, qc
SEQC qa, n, qc
SNEC qa, n, qc
SLTC qa, n, qc
SGTC qa, n, qc
SLEC qa, n, qc
SGEC qa, n, qc
```

Floating point to Integer Conversions:

```
TOINT qa, qc
TOREAL qa, qc
```

Floating Point Arithmetic Instructions:

```
FADD qa, qb, qc
FSUB qa, qb, qc
FMUL qa, qb, qc
FDIV qa, qb, qc
FADDC qa, n, qc
FSUBC qa, n, qc
FMULC qa, n, qc
FDIVC qa, n, qc
```

Floating Point Comparison Instructions:

```
FSEQ qa, qb, qc
FSNE qa, qb, qc
FSLT qa, qb, qc
FSGT qa, qb, qc
FSLE qa, qb, qc
FSGE qa, qb, qc
FSEQC qa, n, qc
FSNEC qa, n, qc
FSLTC qa, n, qc
FSGTC qa, n, qc
FSLEC qa, n, qc
FSGEC qa, n, qc
```

Branch and Jump Instructions:

```
BR label
BRL label, qc
BRZ qa, label
BRNZ qa, label
BRNE qa, label
BREL qa
JMP qa
```

Internal Data Manipulation Instructions:

```
MOVE qa, qc
MOVECF n, qc
```

```
MOVECL n, qc
MOVECI n, qc
MOVECS n, qc
MOVECC n, qc
PACKN qa, qb, qc, n
PACKH qa, qb, qc
PACKL qa, qb, qc
PACKI qa, qb, qc
UNPACK qa, qb, qc
UNPACKC qa, n, qc
EXTAG qa, qc
SETTAG qa, qb, qc
```

Queue Management Instructions:

```
FLUSHQ qc
SPAWN qa, qc
SPAWNC label, qc
SPAWNL qa, qb, qc
MAPQ qa, qb, qc
MAPQC qa, qb, qc
MAPSQ qa, qb
MAPDROP n
UNMAPQ n
CONSUME qa
SEMPTY qa, qc
EEQ qc
```

Thread and Context Management Instructions:

```
PROCID qc
LDCODE qa, qc
OSIZE n
```

Memory Instructions:

```
PTRSIZE qa, qc
ALLOCATE qa, qc
ALLOCATEC n, qc
MML qa, qb
MMS qa, qb
EXCH qa, qb
PARCEL qa, qb, qc
```

Mode and Exception Handling Instructions:

```
GETSTAT qc
SETSTAT qa GETEX qc
SETEX qa
THROW
```

Transaction Support:

```
INITCHK qa, qc (allocate checkpoint image -> qc)
CHKBAR (flush memory queues for impending checkpoint)
CHKPNT qa (copy machine state to memory, then copy primary
image to checkpoint image in qa)
ROLLBCK qa (reset machine state to checkpoint image in qa)
```

Miscellaneous Instructions:

```
RANDOM qc
HINT t,hint
```

Machine Specific Instructions used by OS trap handlers:

```
MOVEMI msr, qc
MOVEMO qa, msr
```

```
SCHEDULE qa, qb
SYNC qa, qc
```

A.3 To do:

Exceptions that can be triggered: access violation Need to define a method for handling exceptions; current proposal is to do arithmetic exceptions “exactly” by setting an exception bit in the resulting tag. Access violations could be handled similarly, where a piece of data tagged invalid or violation can be returned on a read queue. It may be helpful to return the violating capability as well, since many operations deep could have occurred.

ADD

ADD qa, qb, qc

Description:

ADD (addition) takes the sum of qa and qb and returns the result in qc. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. Also, qa may be a capability and qb may be a word, in which case the result will be a capability. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised. If qa is a capability and the add operation with word in qb is not permitted or results in an invalid capability, an operation exception is raised and the result in qc is an invalid capability.

If qa is non-copyable capability, then a successful ADD operation dequeues qa even if the copy/clobber modifier for qais set to copy and an exception is thrown.

The ADD operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    qc ← qa + qb
elif( type(qa,qb) == packed int )
    qc.a ← qa.a + qb.a
    qc.b ← qa.b + qb.b
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a + qb.a
    qc.b ← qa.b + qb.b
    qc.c ← qa.c + qb.c
    qc.d ← qa.d + qb.d
elif( (type(qa) == capability) && (type(qb) == word) )
    temp ← qa + SEXT(qb & ADDRMASK)
    if( temp is valid )
       qc ← temp
       if( qa == non-copyable )
          forceDequeue( qa ) // flag error if copy bit is set on qa
    else
       throw operation exception
       qc ← invalid
else
    throw type exception
```

Exceptions:

Type exception, operation exception, and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

ADDC

ADDC qa, n, qc

Description:

ADDC (addition with constant) takes the sum of qa and n and returns the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is added to each sub-integer. Also, qa may be a capability, in which case the result will be a capability. If qa is a capability and the add operation with word in qb is not permitted or results in an invalid capability, an operation exception is raised and the result in qc is an invalid capability.

If qa is non-copyable capability, then a successful ADDC operation dequeues qa even if the copy/clobber modifier for qais set to copy and an exception is thrown.

The ADDC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa + SEXT(n)
elif( type(qa) == packed int )
    qc.a ← qa.a + n
    qc.b ← qa.b + n
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a + n
    qc.b ← qa.b + n
    qc.c ← qa.c + n
    qc.d ← qa.d + n
elif( type(qa) == capability )
    temp ← qa + SEXT(n & ADDRMASK)
    if( temp is valid )
      qc ← temp
      if( qa == non-copyable )
        forceDequeue( qa ) // flag error if copy bit is set on qa
    else
      throw operation exception
      qc ← invalid
else
    throw type exception
```

Exceptions:

Type exception, operation exception, and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

SUB qa, qb, qc

Description:

SUB (subtraction) takes the difference of qa and qb and returns the result in qc. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. Also, qa may be a capability and qb may be a word, in which case the result will be a capability. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised. If qa is a capability and the add operation with word in qb is not permitted or results in an invalid capability, an operation exception is raised and the result in qc is an invalid capability.

If qa is non-copyable capability, then a successful SUB operation dequeues qa even if the copy/clobber modifier for qa is set to copy, and an exception is thrown.

The SUB operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    qc ← qa - qb
elif( type(qa,qb) == packed int )
    qc.a ← qa.a - qb.a
    qc.b ← qa.b - qb.b
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a - qb.a
    qc.b ← qa.b - qb.b
    qc.c ← qa.c - qb.c
    qc.d ← qa.d - qb.d
elif( (type(qa) == capability) && (type(qb) == word) )
    temp ← qa - SEXT(qb & ADDRMASK)
    if( temp is valid )
       qc ← temp
       if( qa == non-copyable )
         forceDequeue( qa )
    else
       throw operation exception
       qc ← invalid
else
    throw type exception
```

Exceptions:

Type exception, operation exception, and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

SUBC qa, n, qc

Description:

SUBC (subtraction with constant) takes the difference of qa and n and returns the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is subtracted from each sub-integer. Also, qa may be a capability, in which case the result will be a capability. If qa is a capability and the add operation with word in qb is not permitted or results in an invalid capability, an operation exception is raised and the result in qc is an invalid capability.

If qa is non-copyable capability, then a successful SUBC operation dequeues qa even if the copy/clobber modifier for qais set to copy and an exception is thrown.

The SUBC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa - SEXT(n)
elif( type(qa) == packed int )
    qc.a ← qa.a - n
    qc.b ← qa.b - n
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a - n
    qc.b ← qa.b - n
    qc.c ← qa.c - n
    qc.d ← qa.d - n
elif( type(qa) == capability )
    temp ← qa - SEXT(n & ADDRMASK)
    if( temp is valid )
      qc ← temp
      if( qa == non-copyable )
        forceDequeue( qa ) // flag error if copy bit is set on qa
    else
      throw operation exception
      qc ← invalid
else
    throw type exception
```

Exceptions:

Type exception, operation exception, and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

MUL qa, qb, qc

Description:

`MUL` (multiplication) takes the product of `qa` and `qb` and returns the lowest bits of the result in `qc`. `qa` and `qb` must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in `qc` will have the same type as its predecessors. If `qa` or `qb` have incompatible types, `qc` will be tagged as invalid and a type exception raised.

The `MUL` operation is only executed if both `qa` and `qb` operands are available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    qc ← (qa * qb) & 0xFFFFFFFFFFFFFFFF
elif( type(qa,qb) == packed int )
    qc.a ← (qa.a * qb.a) & 0xFFFFFFFF
    qc.b ← (qa.b * qb.b) & 0xFFFFFFFF
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← (qa.a * qb.a) & 0xFFFF
    qc.b ← (qa.b * qb.b) & 0xFFFF
    qc.c ← (qa.c * qb.c) & 0xFFFF
    qc.d ← (qa.d * qb.d) & 0xFFFF
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

MULC qa, n, qc

Description:

`MULC` (multiplication with constant) takes the product of `qa` and `n` and returns the lowest bits of the result in `qc`. `qa` can be of an integer type (word, packed int, packed short, or packed char), in which case the result in `qc` will have the same type as its predecessors. In the case of packed types, the same constant is multiplied to each sub-integer.

The `MULC` operation is only executed if `qa` is available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← (qa * n) & 0xFFFFFFFFFFFFFFFF
elif( type(qa) == packed int )
    qc.a ← (qa.a * n) & 0xFFFFFFFF
    qc.b ← (qa.b * n) & 0xFFFFFFFF
elif( type(qa) == (packed char or packed short) )
    qc.a ← (qa.a * n) & 0xFFFF
    qc.b ← (qa.b * n) & 0xFFFF
    qc.c ← (qa.c * n) & 0xFFFF
    qc.d ← (qa.d * n) & 0xFFFF
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

DIV qa, qb, qc

Description:

DIV (integer divide) takes the division of qa and qb and returns the result in qc. Non-integer results are truncated. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised. If the divisor qb is zero, a divide by zero exception is thrown and qc is marked as invalid, with the specific packed component of qc that is erroneous marked as overflowed.

The DIV operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    if(qb == 0)
      throw divide-by-zero exception
      type(qc) ← invalid, overflow.a
    else
      qc ← qa / qb
elif( type(qa,qb) == packed int )
    if(qb.a == 0)
      throw divide-by-zero exception
      type(qc.a) ← invalid, overflow.a
    else
      qc.a ← qa.a / qb.a
    if(qb.b == 0)
      throw divide-by-zero exception
      type(qc.b) ← invalid, overflow.b
    else
      qc.b ← qa.b / qb.b
elif( type(qa,qb) == (packed char or packed short) )
    if(qb.a == 0)
      throw divide-by-zero exception
      type(qc.a) ← invalid, overflow.a
    else
      qc.a ← qa.a / qb.a
    if(qb.b == 0)
      throw divide-by-zero exception
      type(qc.b) ← invalid, overflow.b
    else
      qc.b ← qa.b / qb.b
    if(qb.c == 0)
      throw divide-by-zero exception
      type(qc.c) ← invalid, overflow.c
    else
      qc.c ← qa.c / qb.c
    if(qb.d == 0)
```

```
      throw divide-by-zero exception
      type(qc.d) ← invalid, overflow.d
    else
      qc.d ← qa.d / qb.d
else
    throw type exception
```

Exceptions:

Type exception, and divide-by-zero exception.

Qualifiers:

None.

Notes:

None.

DIVC qa, n, qc

Description:

DIVC (division with constant) takes the division of qa and n and returns the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is multiplied to each sub-integer. If the divisor n is zero, a divide by zero exception is thrown and qc is marked as invalid, with the specific packed component of qc that is erroneous marked as overflowed.

The DIVC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    if(n == 0)
      throw divide-by-zero exception
      type(qc) ← invalid, overflow.a
    else
      qc ← qa / SEXT(n)
elif( type(qa) == packed int )
    if(n == 0)
      throw divide-by-zero exception
      type(qc.a) ← invalid, overflow.a
      type(qc.b) ← invalid, overflow.b
    else
      qc.a ← qa.a / n
      qc.b ← qa.b / n
elif( type(qa) == (packed char or packed short) )
    if(n == 0)
      throw divide-by-zero exception
      type(qc.a) ← invalid, overflow.a
      type(qc.b) ← invalid, overflow.b
      type(qc.c) ← invalid, overflow.c
      type(qc.d) ← invalid, overflow.d
    else
      qc.a ← qa.a / n
      qc.b ← qa.b / n
      qc.c ← qa.c / n
      qc.d ← qa.d / n
else
    throw type exception
```

Exceptions:

Type exception, divide-by-zero exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

AND,OR,XOR qa, qb, qc

Description:

AND, OR, and XOR perform logical operations on qa and qb and returns the result in qc. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The AND,OR,XOR operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of bitwise AND, OR, XOR
if( type(qa,qb) == word )
    qc ← qa OP qb
elif( type(qa,qb) == packed int )
    qc.a ← qa.a OP qb.a
    qc.b ← qa.b OP qb.b
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a OP qb.a
    qc.b ← qa.b OP qb.b
    qc.c ← qa.c OP qb.c
    qc.d ← qa.d OP qb.d
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

ANDC,ORC,XORC

ANDC,ORC,XORC qa, n, qc

Description:

ANDC, ORC, and XORC perform a logical operator on qa and a sign-extended n and returns the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is operated on each sub-integer.

The ANDC,ORC,XORC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of bitwise AND, OR, XOR
if( type(qa) == word )
    qc ← qa OP SEXT(n)
elif( type(qa) == packed int )
    qc.a ← qa.a OP n
    qc.b ← qa.b OP n
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a OP n
    qc.b ← qa.b OP n
    qc.c ← qa.c OP n
    qc.d ← qa.d OP n
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SHL	qa, qb, qc

Description:

SHL (shift-left) performs a logical left-shift on the contents of qa by the number of digits specified in qb, and returns the result in qc. Bits shifted off the left are thrown away, and zeroes are shifted in from the right. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The SHL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    qc ← qa ≪ (qb & 0x3F)
elif( type(qa,qb) == packed int )
    qc.a ← qa.a ≪ (qb.a & 0x1F)
    qc.b ← qa.b ≪ (qb.b & 0x1F)
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a ≪ (qb.a & 0xF)
    qc.b ← qa.b ≪ (qb.b & 0xF)
    qc.c ← qa.c ≪ (qb.c & 0xF)
    qc.d ← qa.d ≪ (qb.d & 0xF)
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SHLC qa, n, qc

Description:

SHLC (shift left by constant) performs a logical left-shift on the contents of qa by the number of digits specified in n, and returns the result in qc. Bits shifted off the left are thrown away, and zeroes are shifted in from the right. qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. In the case that qa is a packed type, each subword will be shifted left by the same amount. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The SHLC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa ≪ (n & 0x3F)
elif( type(qa) == packed int )
    qc.a ← qa.a ≪ (n & 0x1F)
    qc.b ← qa.b ≪ (n & 0x1F)
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a ≪ (n & 0xF)
    qc.b ← qa.b ≪ (n & 0xF)
    qc.c ← qa.c ≪ (n & 0xF)
    qc.d ← qa.d ≪ (n & 0xF)
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SHR qa, qb, qc

Description:

SHR (logical shift right)performs a logical right-shift on the contents of qa by the number of digits specified in qb, and returns the result in qc. Bits shifted off the right are thrown away, and zeroes are shifted in from the left. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The SHR operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    qc ← qa ≫ (qb & 0x3F)
elif( type(qa,qb) == packed int )
    qc.a ← qa.a ≫ (qb.a & 0x1F)
    qc.b ← qa.b ≫ (qb.b & 0x1F)
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a ≫ (qb.a & 0xF)
    qc.b ← qa.b ≫ (qb.b & 0xF)
    qc.c ← qa.c ≫ (qb.c & 0xF)
    qc.d ← qa.d ≫ (qb.d & 0xF)
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SHRC	qa, n, qc

Description:

SHRC (logical shift right by constant) performs a logical right-shift on the contents of qa by the number of digits specified in n, and returns the result in qc. Bits shifted off the right are thrown away, and zeroes are shifted in from the left. qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. In the case that qa is a packed type, each subword will be shifted left by the same amount. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The SHRC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa ≫ (n & 0x3F)
elif( type(qa) == packed int )
    qc.a ← qa.a ≫ (n & 0x1F)
    qc.b ← qa.b ≫ (n & 0x1F)
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a ≫ (n & 0xF)
    qc.b ← qa.b ≫ (n & 0xF)
    qc.c ← qa.c ≫ (n & 0xF)
    qc.d ← qa.d ≫ (n & 0xF)
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SRA	qa, qb, qc

Description:

SRA (arithmetic shift right) performs an arithmetic (sign-preserving) right-shift on the contents of qa by the number of digits specified in qb, and returns the result in qc. Bits shifted off the right are thrown away, and the value of the sign bit is shifted in from the left (zero if the number being shifted is positive, one if the number being shifted is negative). qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The SRA operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    qc ← qa SRA (qb & 0x3F)
elif( type(qa,qb) == packed int )
    qc.a ← qa.a SRA (qb.a & 0x1F)
    qc.b ← qa.b SRA (qb.b & 0x1F)
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a SRA (qb.a & 0xF)
    qc.b ← qa.b SRA (qb.b & 0xF)
    qc.c ← qa.c SRA (qb.c & 0xF)
    qc.d ← qa.d SRA (qb.d & 0xF)
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SRAC qa, n, qc

Description:

SRAC (arithmetic shift right by constant) performs an arithmetic right-shift on the contents of qa by the number of digits specified in n, and returns the result in qc. Bits shifted off the right are thrown away, and the value of the sign bit is shifted in from the left (zero if the number being shifted is positive, one if the number being shifted is negative). qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. In the case that qa is a packed type, each subword will be shifted left by the same amount. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The SRAC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa SRA (n & 0x3F)
elif( type(qa) == packed int )
    qc.a ← qa.a SRA (n & 0x1F)
    qc.b ← qa.b SRA (n & 0x1F)
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a SRA (n & 0xF)
    qc.b ← qa.b SRA (n & 0xF)
    qc.c ← qa.c SRA (n & 0xF)
    qc.d ← qa.d SRA (n & 0xF)
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SEQ,SLT,SLE

SEQ,SLT,SLE qa, qb, qc

Description:

SEQ, SLT, and SLE perform magnitude comparisons on its arguments and produce a binary result. SEQ test if qa and qb are equal; SLT tests if qa is less than qb; and SLE tests if qa is less than or equal to qb˙qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The Sxx operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of arithmetic =, <, ≤:
if( type(qa,qb) == word )
    qc ← qa OP qb ?  1 :  0
elif( type(qa,qb) == packed int )
    qc.a ← qa.a OP qb.a ?  1 :  0
    qc.b ← qa.b OP qb.b ?  1 :  0
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a OP qb.a ?  1 :  0
    qc.b ← qa.b OP qb.b ?  1 :  0
    qc.c ← qa.c OP qb.c ?  1 :  0
    qc.d ← qa.d OP qb.d ?  1 :  0
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SEQC,SLTC,SLEC

SEQC,SLTC,SLEC qa, qb, qc

Description:

SEQC, SLTC, and SLEC perform magnitude comparisons on its arguments and produce a binary result. SEQC test if qa and n are equal; SLTC tests if qa is less than n; and SLEC tests if qa is less than or equal to n. qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The SxxC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of arithmetic =, <, ≤:
if( type(qa,qb) == word )
    qc ← qa OP n ?  1 :  0
elif( type(qa,qb) == packed int )
    qc.a ← qa.a OP n ?  1 :  0
    qc.b ← qa.b OP n ?  1 :  0
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a OP n ?  1 :  0
    qc.b ← qa.b OP n ?  1 :  0
    qc.c ← qa.c OP n ?  1 :  0
    qc.d ← qa.d OP n ?  1 :  0
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

TOINT	qa, qc

Description:

`TOINT` (floating point to integer convert) converts the floating-point value in `qa` to an integer stored in `qc`. Conversion is done using the truncation or "round to zero" method, so that the number 9.6 is converted to 9, and the number -2.8 is converted to -2. Overflow in either sign extreme results in `qc` having the maximum sized integer of the appropriate sign and the overflow bit being set in `qc`'s type field. `qa` must be of the floating point type, and the result in `qc` is of type word. If `qa` has an incompatible type, `qc` will be tagged as invalid and a type exception raised.

The `TOINT` operation is only executed if `qa` is available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← (word) qa
    type(qc) ← word
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field. Attempting to convert $+\infty$ will result in the largest positive representable integer in `qc` and set the overflow bit of `qc`. Likewise, converting $-\infty$ will result in the most negative representable integer in `qc` and set the overflow bit of `qc`.

Attempting to convert NaN's will result in `qc` having an invalid type.

TOREAL

TOREAL	qa, qc

Description:

TOREAL (integer to floating point convert) converts the integer value in qa to the nearest representable floating-point value stored in qc. qa must be of the word type, and the result in qc is of the floating point type. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The TOREAL operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← (floating-point) qa
    type(qc) ← floating-point
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

FADD

FADD qa, qb, qc

Description:

FADD (floating-point addition) takes the sum of qa and qb and returns the result in qc. qa and qb must be of the floating-point type, and the result qc is of the floating point type.

The FADD operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == floating-point )
    qc ← qa + qb
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If any operand is a NaN, the result will be NaN.

FADDC

FADDC qa, n, qc

Description:

FADDC (floating-point addition with constant) takes the sum of qa and n and returns the result in qc. qa must be of the floating-point type, and the result qc is of the floating point type.

The FADDC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← qa + n
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If qa is a NaN, the result will be NaN.

FSUB qa, qb, qc

Description:

FSUB (floating-point subtraction) takes the difference of qa and qb and returns the result in qc. qa and qb must be of the floating-point type, and the result qc is of the floating point type.

The FSUB operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == floating-point )
    qc ← qa - qb
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If any operand is a NaN, the result will be NaN.

FSUBC qa, n, qc

Description:

FSUBC (floating-point addition with constant) takes the difference of qa and n and returns the result in qc. qa must be of the floating-point type, and the result qc is of the floating point type.

The FSUBC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← qa - n
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If qa is a NaN, the result will be NaN.

FMUL qa, qb, qc

Description:

FMUL (floating-point multiply) takes the product of qa and qb and returns the result in qc. qa and qb must be of the floating-point type, and the result qc is of the floating point type.

The FMUL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == floating-point )
    qc ← qa * qb
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If any operand is a NaN, the result will be NaN.

FMULC qa, n, qc

Description:

FMULC (floating-point multiply with constant) takes the product of qa and n and returns the result in qc. qa must be of the floating-point type, and the result qc is of the floating point type.

The FMULC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← qa + n
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If qa is a NaN, the result will be NaN.

FDIV qa, qb, qc

Description:

`FDIV` (floating-point division) divides `qa` by `qb` and returns the quotient in `qc`. `qa` and `qb` must be of the floating-point type, and the result `qc` is of the floating point type. If `qb` is zero, a divide-by-zero exception is thrown and the result `qc` is tagged as invalid.

The `FDIV` operation is only executed if both `qa` and `qb` operands are available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == floating-point )
    qc ← qa / qb
else
    throw type exception
```

Exceptions:

Type exception, overflow exception, and divide-by-zero exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

If any operand is a NaN, the result will be NaN.

FDIVC qa, n, qc

Description:

FDIVC (floating-point divide by constant) divides qa by n and returns the result in qc. qa must be of the floating-point type, and the result qc is of the floating point type. If n is zero, a divide-by-zero exception is thrown and the result qc is tagged as invalid.

The FDIVC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← qa / n
else
    throw type exception
```

Exceptions:

Type exception, overflow exception, and divide-by-zero exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If qa is a NaN, the result will be NaN.

FSEQ,FSLT,FSLE qa, qb, qc

Description:

FSEQ, FSLT, and FSLE perform magnitude comparisons on its arguments and produce a binary integer result. FSEQ test if qa and qb are equal; FSLT tests if qa is less than qb; and FSLE tests if qa is less than or equal to qb˙qa and qb must be of the floating-point type. The result qc is of type word. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The FSxx operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of arithmetic =, <, ≤:
if( type(qa,qb) == floating-point )
    qc ← qa OP qb ?  1 :  0
    type(qc) ← word
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

If any of the operands are NaNs, the result is tagged as invalid.

FSEQC,FSLTC,FSLEC

FSEQC,FSLTC,FSLEC qa, qb, qc

Description:

FSEQC, FSLTC, and FSLEC perform magnitude comparisons on its arguments and produce a binary result. FSEQC test if qa and n are equal; FSLTC tests if qa is less than n; and FSLEC tests if qa is less than or equal to n. qa must be of the floating-point type, and the result in qc is of type word. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The FSxxC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of arithmetic =, <, ≤:
if( type(qa,qb) == floating-point )
    qc ← qa OP n ?  1 :  0
    type(qc) ← word
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

If qa is a NaN, the result is tagged as invalid.

BR offset

Description:

BR (unconditional branch) adds the number specified in the offset field to the incremented program counter. Execution immediately begins at the new PC value; there are no branch delay slots.

Operation:

```
PC ← PC + 1
PC ← PC + offset
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown.

Qualifiers:

None.

Notes:

None.

BRL offset, qc

Description:

BRL (unconditional branch with link) adds the number specified in the offset field to the incremented program counter. Execution immediately begins at the new PC value; there are no branch delay slots. The incremented program counter offset relative to the start of code (be it method, object, or absolute-referenced) is stored in qcas a word data type; execution stalls if qc is full and applying backpressure.

The BRL operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
PC ← PC + 1
qc ← PC
PC ← PC + offset
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown.

Qualifiers:

None.

Notes:

None.

BRZ qa, offset, hint

Description:

BRZ (branch if zero) adds the number specified in the offset field to the incremented program counter if the value in qa is zero; otherwise, the program counter is just incremented to the next instruction. qa must be of the word type. Execution immediately begins at the new PC value; there are no branch delay slots.

The BRZ operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    if( qa == 0 )
      PC ← PC + 1 + offset
    else
      PC ← PC + 1
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown. A type exception is thrown if the type of qa is not word.

Qualifiers:

None.

Notes:

The hint field is an implementation-specific 8-bit number that serves as a branch prediction hint. The semantics of hint are such that an incorrect branch hint still leads to correct but slower execution. The actual value of hint is allowed to have cache-incoherent mutation during run-time as the dynamic hardware branch-predictor sees fit.

BRNZ qa, offset, hint

Description:

BRNZ (branch if not zero) adds the number specified in the `offset` field to the incremented program counter if the value in `qa` is not zero; otherwise, the program counter is just incremented to the next instruction. `qa` must be of the word type. Execution immediately begins at the new PC value; there are no branch delay slots.

The BRNZ operation is only executed if `qa` is available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    if( qa != 0 )
      PC ← PC + 1 + offset
    else
      PC ← PC + 1
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown. A type exception is thrown if the type of `qa` is not word.

Qualifiers:

None.

Notes:

The `hint` field is an implementation-specific 8-bit number that serves as a branch prediction hint. The semantics of `hint` are such that an incorrect branch hint still leads to correct but slower execution. The actual value of `hint` is allowed to have cache-incoherent mutation during run-time as the dynamic hardware branch-predictor sees fit.

BRNE qa, offset

Description:

BRNE (branch if not empty) adds the number specified in the offset field to the incremented program counter if qa is not empty; otherwise, the program counter is just incremented to the next instruction. The data in qa is not affected by this instruction. Execution immediately begins at the new PC value; there are no branch delay slots.

Operation:

```
if( qa != empty )
    PC ← PC + 1 + offset
else
    PC ← PC + 1
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown.

Qualifiers:

The qualifier is ignored by this instruction; qa is never dequeued.

Notes:

None.

BREL qa

Description:

BREL (unconditional relative branch) adds the number in qa to the incremented program counter. qa must be of the word type. Execution immediately begins at the new PC value; there are no branch delay slots.

The BREL operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    PC ← PC + 1 + qa
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown. A type exception is thrown if the type of qa is not word.

Qualifiers:

None.

Notes:

None.

JMP	qa, hint

Description:

`JMP` (unconditional jump) sets the value in `PC` to the value in `qa`. Execution immediately begins at the new `PC` value; there are no branch delay slots. `qa` must be of type word.

The `JMP` operation is only executed if `qa` is available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
   PC ← qa
```

Exceptions:

If the destination of the `PC` is in a protected or invalid page, an exception is thrown.

Qualifiers:

None.

Notes:

The `hint` field is an implementation-specific 48-bit number that serves as a jump prediction destination hint. The semantics of `hint` are such that an incorrect jump hint still leads to correct but slower execution. The actual value of `hint` is allowed to have cache-incoherent mutation during run-time as the dynamic hardware jump-predictor sees fit.

MOVE

MOVE qa, qc

Description:

MOVE (move) takes the value in qa and puts it into qc. The exact state of the queues after the MOVE instruction depends on the @ (copy/clobber) modifiers applied to the queue specifiers.

The MOVE operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

qc ← qa

Exceptions:

An operation exception is thrown if a copy operator is applied to data in qa that is tagged non-copyable. The result in qc is tagged as invalid, and the original value remains untouched in qa.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

MOVECF n, qc

Description:

MOVECF (move floating point constant) takes the 32-bit floating-point constant specified in n, converts it to the nearest ADAM 64-bit floating point number, and puts the properly typed result into qc. The exact state of qc after the MOVECF instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The MOVECF operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc ← (floating-point) n
type(qc) ← floating-point
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Because of the conversion from a 32-bit opcode-stored representation to a 64-bit standard ADAM floating point representation, the result in qc may exhibit some small roundoff error when compared to the desired constant.

Exact semantics vary according to the use of the @ modifier.

MOVECL n, qc

Description:

`MOVECL` (move long integer constant) takes the 32-bit constant specified in `n`, sign-extends it to an ADAM native 64-bit word, and puts the properly typed result into `qc`. The exact state of `qc` after the `MOVECL` instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The `MOVECL` operation is only executed if there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
qc ← SEXT(n)
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

MOVECI n, qc

Description:

`MOVECI` (move packed integer constant) takes the 32-bit constant specified in `n`, places it in the lower bits of a packed integer, sets the upper bits of the packed integer to zero, and puts the properly typed result into `qc`. The exact state of `qc` after the `MOVECI` instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The `MOVECI` operation is only executed if there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
qc.a ← 0
qc.b ← n
type(qc) ← packed integer
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

MOVECS n, qc

Description:

MOVECS (move packed short constant) takes the dual 16-bit packed short constant specified in n, places it in the lower bits of a packed short, sets the upper bits of the packed short to zero, and puts the properly typed result into qc. The exact state of qc after the MOVECS instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The MOVECS operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc.a ← 0
qc.b ← 0
qc.c ← n[31:16]
qc.d ← n[15:0]
type(qc) ← packed short
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

MOVECC n, qc

Description:

MOVECC (move packed unicode character constant) takes the dual 16-bit packed unicode character constant specified in n, places it in the lower bits of a packed char, sets the upper bits of the packed char to zero, and puts the properly typed result into qc. The exact state of qc after the MOVECC instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The MOVECC operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc.a ← 0
qc.b ← 0
qc.c ← n[31:16]
qc.d ← n[15:0]
type(qc) ← packed character
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

PACKN qa, qb, qc, n

Description:

PACKN (Pack Anything) takes the data in qa and inserts it at a position specified by n into the data from qb, and places the result into qc. qa must be of type word, and qb must be of a packed integer type. The result in qc has the same type as qb.

The PACKN operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
   if( type(qb) == packed int )
      if( n == 0 )
         qc.a ← qa & 0xFFFFFFFF
         qc.b ← qb.b
      else
         qc.a ← qb.a
         qc.b ← qa & 0xFFFFFFFF
   elif( type(qb) == packed short or packed char )
      if( n == 0 )
         qc.a ← qa & 0xFFFF
         qc.b ← qb.b
         qc.c ← qb.c
         qc.d ← qb.d
      elif( n == 1 )
         qc.a ← qb.a
         qc.b ← qa & 0xFFFF
         qc.c ← qb.c
         qc.d ← qb.d
      elif( n == 2 )
         qc.a ← qb.a
         qc.b ← qb.b
         qc.c ← qa & 0xFFFF
         qc.d ← qb.d
      else
         qc.a ← qb.a
         qc.b ← qb.b
         qc.c ← qb.c
         qc.d ← qa & 0xFFFF
   else
      throw type exception
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

PACKH	qa, qb, qc

Description:

PACKH (Pack High Half of Packed Short or Char) takes packed integer data in qa, masks the data and inserts it into the high half of qb, and places the result into qc. qb must be of type packed short or packed char. The result in qc has the same type as qb.

The PACKH operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == packed int && type(qb) == packed short or packed char)
   qc.a ← qa.a & 0xFFFF
   qc.b ← qa.b & 0xFFFF
   qc.c ← qb.c
   qc.d ← qb.d
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

PACKL	qa, qb, qc

Description:

PACKL (Pack Low Half of Packed Short or Char) takes packed integer data in qa, masks the data and inserts it into the low half of qb, and places the result into qc. qb must be of type packed short or packed char. The result in qc has the same type as qb.

The PACKL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == packed int && type(qb) == packed short or packed char)
   qc.a ← qb.a
   qc.b ← qb.b
   qc.c ← qa.a & 0xFFFF
   qc.d ← qa.b & 0xFFFF
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

PACKI	qa, qb, qc

Description:

PACKI (Pack to Packed Integer) takes word data in qa and qb, masks the data and packs it into a packed integer stored in qc.

The PACKI operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
   qc.a ← qa & 0xFFFFFFFF
   qc.b ← qb & 0xFFFFFFFF
   type(qc) ← packed integer
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

UNPACK

UNPACK	qa, qb, qc

Description:

UNPACK (Unpack) takes a packed integer type qa and extracts and sign-extends the data at location qb into qc. The result qc is of type word.

The UNPACK operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qb) == word)
   if(type(qa) == packed int)
      if(qb == 0)
         qc ← SEXT(qa.a)
      else
         qc ← SEXT(qa.b)
   elif(type(qa) == packed short or packed char)
      if(qb == 0)
         qc ← SEXT(qa.a)
      elif(qb == 1)
         qc ← SEXT(qa.b)
      elif(qb == 2)
         qc ← SEXT(qa.c)
      else
         qc ← SEXT(qa.d)
   else
      throw type exception
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

UNPACKC qa, n, qc

Description:

UNPACKC (Unpack with constant) takes a packed integer type qa and extracts and sign-extends the data at location n into qc. The result qc is of type word.

The UNPACKC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == packed int)
   if(n == 0)
      qc ← SEXT(qa.a)
   else
      qc ← SEXT(qa.b)
elif(type(qa) == packed short or packed char)
   if(n == 0)
      qc ← SEXT(qa.a)
   elif(n == 1)
      qc ← SEXT(qa.b)
   elif(n == 2)
      qc ← SEXT(qa.c)
   else
      qc ← SEXT(qa.d)
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

FLUSHQ qc

Description:

FLUSHQ (Flush Queue) is a special-format instruction, where qc is interpreted as an immediate constant. FLUSHQ discards all values currently in the queue specified by the immediate constant qc. The function of FLUSHQ upon a queue which has mappings to other contexts, be it head or tail mappings, is UNPREDICTABLE. If qc is already empty, nothing happens and execution continues.

Operation:

```
qc ← empty
```

Exceptions:

Throws a mapping exception if qc has any mappings.

Qualifiers:

None.

Notes:

None.

PROCID qc

Description:

`PROCID` (Get Process ID) places the value of the current context ID into `qc`. `qc` is a capability with the owner bit set. In addition, the read and write bits are set. If the context ID is to be passed to another thread, care must be taken to set the permissions properly.

The `PROCID` operation is only executed if there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
qc ← context ID
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

PTRSIZE	qa, qc

Description:

`PTRSIZE` (Get Pointer Size) computes the size of the region of data pointed to by the capability in `qa` and places the size, in words, in `qc`. The `PTRSIZE` operation is valid on any capability, regardless of its permissions. The result in `qc` is of the word type.

The `PTRSIZE` operation is only executed if `qa` is available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == capability )
   qc ← sizeof(qa) in words
   type(qc) ← word
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

CONSUME qa

Description:

`CONSUME` (Consume Data) reads exactly one piece of data out of `qa` and discards it. If `qa` is initially empty, `CONSUME` blocks.

Operation:

```
while( qa is empty )
   stall
if( no @ operator on qa )
   dequeue head of qa
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

SEMPTY	qa, qc

Description:

SEMPTY (Set if Empty) is a special format instruction, where qa is interpreted as an immediate constant. SEMPTY tests to see if the queue specified by the immediate constant qa is empty, and if it is, it places an integer 1 into qc. Otherwise, a 0 is written into qc. The type of the result qc is word.

Operation:

```
if((qa & 0x7F) is empty)
   qc ← 1
else
   qc ← 0
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

EEQ qa

Description:

EEQ (forcE Empty Queue) is a special format instruction, where qa is interpreted as an immediate constant. EEQ tests to see if the queue specified by the immediate constant qa is empty, and if it is, it increments the PC; if not, the PC remains constant and a yielding stall is reported to the scheduler.

Operation:

```
if((qa & 0x7F) is empty)
   pc ← pc + 1
else
   pc ← pc
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction complicates the implementation of the processor core. An alternative would be to use SEMPTY and a BRZ instruction to create a programmatic loop to check for the emptiness of a queue. However, for the purposes of backward compatibility with an older ISA, it is included in the documentation.

RANDOM qc

Description:

RANDOM (Generate Random Number) places a cryptographically secure random integer of type word into qc. RANDOM may be implemented as an external hardware device to the processor. Because 64 bits of entropy must be collected for each RANDOM instruction, it is possible to request random numbers faster than the processor or device is capable of generating them. In this case, the operation blocks until a random number becomes available. In order to smooth out demand patterns, the number generating device may queue up several pre-generated numbers.

The RANDOM operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

qc ← random number between -2^{63} and $2^{63}-1$
type(qc) ← word

Exceptions:

None.

Qualifiers:

None.

Notes:

The exact implementation of the RANDOM function should be disclosed in a public fashion before it can be trusted. More information on cryptographically secure random numbers can be found in Annex D.6 “Random number generation” of the IEEE 1363-2000 standard and in RFC1750, “Randomness Recommendations for Security”. A user desiring to verify the randomness properties of the RANDOM instruction may wish to refer to Ueli M. Maurer’s “A Universal Statistical Test for Random Bit Generators”, *Institute of Theoretical Computer Science, ETH Zürich*, 1992, Journal of Cryptology, Vol. 5, No. 2.

GETSTAT

GETSTAT qc

Description:

GETSTAT (Get Status Register) copies the contents of the status register into qc. There are some portions of the status register that are implementation-specific. qc is of type word.

The GETSTAT operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc ← status register
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Please refer to the implementation notes and the architecture specification for the meaning of the status register bits.

SETSTAT qa

Description:

SETSTAT (Set Status Register) copies the contents of qa into the modifiable portions of the status register. There are some portions of the status register that are implementation-specific. qa must be of type word.

The SETSTAT operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == word)
   status register ← qa
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

Please refer to the implementation notes and the architecture specification for the meaning of the status register bits. Some of the bits of the status register are read-only and are unaffected by SETSTAT.

GETEX qc

Description:

GETEX (Get Exception Context ID) places the current exception handler's context ID into qc. The permissions on the exception handler ID are set to opaque and owner.

The GETEX operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc ← Exception Register
type(qc) ← capability
permissions(qc) ← opaque, owner
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

SETEX

SETEX qa

Description:

SETEX (Set Exception Context ID) sets the current context's exception handler ID to be the capability in `qa`. The operation blocks if `qa` is applying backpressure.

Operation:

```
if(type(qa) == capability)
   Exception Register ← qa
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

THROW

THROW

Description:

THROW (Throw Soft Exception) causes the current context to be set to the exception handler context and for the PC to jump to the exception handler's server code. In addition, the current context ID is saved into the Exceptioned Context ID register. The user may layer additional conventions on top of the basic THROW semantics; for example, the user may require that q127 contain a soft exception ID.

Operation:

```
PC ← PC + 1
Exceptioned Context ID ← context ID
context ID ← exception handler ID
PC ← exception handler server code start
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Note that there is no requirement for a saved PC because the PC of the exceptioned context is not overwritten by the exception handler PC: the context ID is set to the exception handler before the PC is modified.

This is a multi-cycle, variable execution duration instruction.

EXTAG	qa, qc

Description:

EXTAG (Extract Tag) extracts the tag bits out of qa and places them into qc. The tag bits are placed in the MSB's of qc and zero-padded to the right. The tag region of a piece of data includes the top 16 bits, whereas the tag region for a capability includes the top 45 bits. The type of the result in qc is word.

The EXTAG operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == capability)
   qc ← {qa[79:55],39'b0}
else   qc ← {qa[79:64],48'b0}
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

SETTAG	qa, qb, qc

Description:

SETTAG (Set Tag) sets the tag of the data in qb to the value of the LSB's of qa, and places the result into qc. This is a very powerful operator, as it can force a literal binary transmutation of data types and change several important attributes about a piece of data. If the value of the bits in qa corresponds to a capability, the type of qb must also be a capability, and the owner bit for qb must be set. qa must be of type word.

The SETTAG operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == word)
   if(type(qb) == capability)
      if(!owner(qb))
         throw operation exception
      else
         tags(qb) ← qa[63:39]
   else
      if(qa[63] == 1)
         throw operation exception
      else
      tags(qb) ← qa[63:48]
else
   throw type exception
```

Exceptions:

Operation exception, type exception.

Qualifiers:

None.

Notes:

None.

ALLOCATE qa, qc

Description:

ALLOCATE (Allocate Capability) creates a capability qc of the size nearest to the number of words specified in qa. The address of the capability and the increment-only bit are set to restrict the accessible portion of the capability to exactly the size specified in qa. qa must be of type word. If the allocation fails, qc is returned as an invalid capability, and an out of memory exception is thrown.

The ALLOCATE operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == word)
   if(qa words available)
      qc ← capability of size qa bytes
   else
      qc ← invalid capability
      throw out of memory exception
else
   throw type exception
```

Exceptions:

Out of memory exception, type exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

ALLOCATEC n, qc

Description:

ALLOCATEC (Allocate Capability, Size in Constant Field) creates a capability qc of the size nearest to the number of words specified in n. The address of the capability and the increment-only bit are set to restrict the accessible portion of the capability to exactly the size specified in n. If the allocation fails, qc is returned as an invalid capability, and an out of memory exception is thrown.

The ALLOCATEC operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(n words available)
   qc ← capability of size n bytes
else
   qc ← invalid capability
   throw out of memory exception
```

Exceptions:

Out of memory exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

MML qa, qb

Description:

MML (Map Memory Load) maps the queue number specified in qa to a load address queue, and maps the return data of the load into the queue number specified in qb. qa and qb must be of type word.

The memory subsystem expects that the first address entered into a memory address queue be the access capability, and that subsequent entries to the load address queue be offsets on the initial capability. Enqueueing the initialization capability does not cause the memory subsystem to return a load value. If a capability is sent to the memory subsystem following the initialization capability, the new capability subsumes the old one; again, no load value is returned in response to this load capability being sent.

This operation stalls until both qa and qb contain a value.

Operation:

```
if(type(qa,qb) == word)
   MAP (qa & 0x7F) to memory load address queue
   MAP memory load return data queue to (qb & 0x7F)
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

MMS qa, qb

Description:

`MMS` (Map Memory Store) maps the queue number specified in `qa` to a store address queue, and maps the queue number specified in `qb` to a store data queue. `qa` and `qb` must be of type word.

Data and addresses may be enqueued at differing times and rates, but the invariant is that the store blocks until both queues have at least one element in them, and that data and address pairs are strictly correlated by their relative order in the queues.

The memory subsystem expects that the first address entered into a memory address queue be the access capability; this first access is **not** matched with a data element in the store data queue. Subsequent addresses are then interpreted as offsets to the initial access capability and are paired with data values in the store data queue.

This operation stalls until both `qa` and `qb` contain a value.

Operation:

```
if(type(qa,qb) == word)
   MAP (qa & 0x7F) to memory store address queue
   MAP (qb & 0x7F) to memory store data queue
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

EXCH qa, qb

Description:

EXCH (Declare Exchange Pair) marks the queues numbers specified in qa and qb as a memory exchange pair. qa must be previously established by an MMS instruction as the data queue for a store, and qb must be previously established by an MML instruction as the data queue for a load. Upon completion of this instruction, any store to qa will cause the contents of the data displaced by the store to appear in qb. This operation is guaranteed by the memory system to be atomic.

Once qa and qb have been declared as an exchange pair, their behavior is as follows. When an instruction attempts to put a result into qa, it will block until a lock is obtained on the capability referenced by the store mapping. It is thus **critical** that the address queue for the store pair be touched before attempting to move data into the data queue, or else execution will deadlock. Once the lock has been obtained, the data is transmitted to the storage location and swapped remotely. At this point, the lock is re-opened, even as the return value is transmitted back to this node's qb.

The EXCH mapping remains in effect until either the store or load mapping is destroyed.

This operation stalls until both qa and qb contain a value.

Operation:

```
if(type(qa,qb) == word)
   if(qa mapped to data part of store queue AND qb mapped to data part of
load queue)
      associate qa, qb as an exchange pair in the memory subsystem
   else
      throw exchange exception
else
   throw type exception
```

Exceptions:

Type exception and exchange exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

SPAWN qa, qc

Description:

SPAWN (Spawn) starts a new thread by allocating space for the thread, creating an entry in the thread scheduler for the thread with PC set to the value in qa, and returning the thread ID (which is also a capability to thread's data) in qc. The permissions of the thread ID capability are set to opaque and not owner. The size of the space to be allocated for the thread is encoded in an OSIZE opcode that should be the first instruction of the new thread.

qa must be of type word.

The SPAWN operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == word)
   qc ← capability of size indicated in OSIZE opcode at address in qa
   if(qc == invalid)
      throw out of memory exception
   else
      create thread scheduler entry (new thread ID, PC = qa)
else
   throw type exception
```

Exceptions:

Type exception, Out of memory exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

SPAWNL	qa, qb, qc

Description:

SPAWNL (Load Code and Spawn) starts a new thread by allocating space for the thread, loading its code specified in qb into code space, and creating an entry in the thread scheduler for the thread with PC set to the value in qa, and returning the thread ID (which is also a capability to thread's data) in qc. The permissions of the thread ID capability are set to opaque and not owner. The size of the space to be allocated for the thread is encoded in an OSIZE opcode that should be the first instruction of the new thread.

qa must be of type word, and qb must be a capability to a character array that describes a universal locator for the code resource.

The SPAWNL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == word && type(qb) == capability)
   load code specified by qb into code space   qc ← capability of size in-
dicated in OSIZE opcode at address in qa
   if(qc == invalid)
      throw out of memory exception
   else
      create thread scheduler entry (new thread ID, PC = qa)
else
   throw type exception
```

Exceptions:

Type exception, Out of memory exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

SPAWNC n, qc

Description:

SPAWNC (Spawn with PC-constant offset) starts a new thread by allocating space for the thread, creating an entry in the thread scheduler for the thread with PC set to the value of PC + 1 + n, and returning the thread ID (which is also a capability to thread's data) in qc. The permissions of the thread ID capability are set to opaque and not owner. The size of the space to be allocated for the thread is encoded in an OSIZE opcode that should be the first instruction of the new thread.

The SPAWNC operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc ← capability of size in OSIZE opcode at (n + PC + 1)
if(qc == invalid)
   throw out of memory exception
else
   create thread scheduler entry (new thread ID, PC = n + PC + 1)
```

Exceptions:

Out of memory exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

MAPQ	qa, qb, qc

Description:

MAPQ (Map Queue) is a special-format instruction. qa is actually interpreted as an immediate constant: it specifies the queue number in the current context that is to be mapped. MAPQ does not actually read or modify the contents of qa in any way. The copy/clobber modifier has no effect on the value of qa in this case. qb specifies the queue number to read for the queue number of the destination mapping, and qc specifies the queue number to read for the destination context ID.

The MAPQ operation is only executed if both qb and qc operands are available.

Operation:

```
if(type(qb) == word && type(qc) == capability)
   map queue ‘‘qa’’.tail in current context to
   queue ((qb & 0x7F) ≫ 7).head in context qc
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

The odd format of this instruction is an artifact of backward compatibility with an earlier version of the instruction set. This instruction may be represented inside the hardware implementation in a more typical fashion and require the assembler to do a simple format translation. This instruction may take multiple cycles to complete.

MAPQC	qa, qb, qc

Description:

MAPQC (Map Queue with Destination as Constant) is a special-format instruction. qa and qb are actually interpreted as immediate constants: they specify the queue number in the current context and the destination queue number, respectively, that is to be mapped. MAPQC does not actually read or modify the contents of qa or qb in any way. The copy/clobber modifier has no effect on the value of qa and qb in this case. qc specifies the queue number to read for the destination context ID.

The MAPQC operation is only executed if the qc operand is available.

Operation:

```
if(type(qc) == capability)
   map queue ‘‘qa’’.tail in current context to
   queue ‘‘qb’’.head in context qc
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

The odd format of this instruction is an artifact of backward compatibility with an earlier version of the instruction set. This instruction may be represented inside the hardware implementation in a more typical fashion and require the assembler to do a simple format translation. This instruction may take multiple cycles to complete.

MAPSQ qa, qb

Description:

`MAPSQ` (Map Queue Source) is a special-format instruction. `qa` and `qb` are actually intepreted as immediate constants. `MAPSQ` creates a mapping such that every element enqueued *by the network interface* into the queue specified in the immediate constant `qa` also enqueues the context ID of the data's source into the queue specified by the immediate constant `qb`. The arrival of data from the network interface in the queue specified by `qa` is guaranteed to be simultaneous with the arrival of the context ID in the queue specified by `qb`. The resulting type of the IDs in `qb` are capability, with the opaque bit set and the owner bit cleared.

Operation:

```
map incoming data source ID of queue (qa & 0x7F) to (qb & 0x7F)
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

Note that data arriving in `qa` via local operations do not cause `qb` to have the source enqueued; thus, it is not recommended to share `qa` as both a target for local and remote operations.

MAPDROP qa

Description:

MAPDROP (Set Mapping to Drop Mode) is a special-format instruction where qa is interpreted as an immediate constant. MAPDROP sets the mode of the mapping of the queue number specified by the immediate constant qa to “drop” mode. In this mode, backpressure on the queue causes data to be dropped instead of stalling the context. This is particularly useful when implementing pure streaming operators on real-time datatypes such as video or audio.

Operation:

```
set mode of queue (qa & 0x7F) to drop mode
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

UNMAPQ qa

Description:

`UNMAPQ` (Unmap A Queue) is a special format instruction, in that `qa` is interpreted as an immediate constant. `UNMAPQ` resets the mapping of the queue specified by the immediate constant `qa` to the default (current context ID). Care should be taken to guarantee that the specified queue is empty before issuing this instruction, otherwise left-over data that may be in the queue when this instruction retires will never be delivered to its destination.

Operation:

```
set the mapping of queue (qa & 0x7F) to the current context ID
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

PARCEL qa, qb, qc

Description:

PARCEL (Parcel out a Capability) takes a capability in qa and attempts to create a sub-capability with the address and tags described in qb. The result is placed in qc. qa must be a capability, qb is a word type, and the result qc is a capability. The format of the sub-capability address and tag specifier is 15 bits of tags followed by a 1 bit increment-only field, followed by a 35 bit address field. The unused bits to the left are ignored.

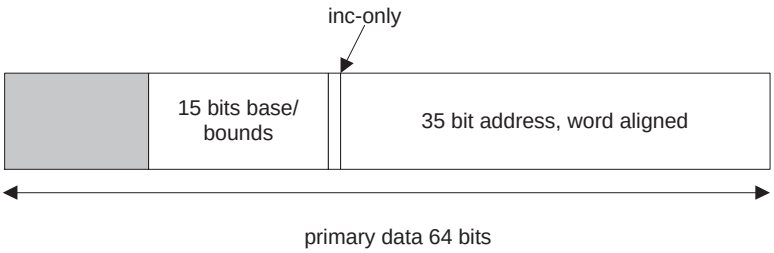

Figure A-1: qb format for the PARCEL instruction

If the capability described by qb is outside the bounds of the given capability in qa, an operation exception is thrown and the result in qc is invalid.

The PARCEL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == capability && type(qb) == word)
   qc ← sub-capability of qa described by qb
else
   throw type exception
```

Exceptions:

Type exception and operation exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

LDCODE	qa, qc

Description:

LDCODE (Dynamically Load Code) takes a capability in qa which contains a character array that names a code object and its path, attempts to load it into code memory, and returns the absolute PC address of the code as a word in qc. A failure to complete this operation causes a code load exception to be thrown and qc to be invalid.

(Need to determine if the return should be a PC value, or if it should be a context ID to an object server that was started...)

The LDCODE operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == capability)
   if(qa.permissions == read, not opaque, valid)
      load code described by character array in qa
      qc ← PC of code entry point
      if(tags(qc) == invalid)
         throw code load exception
   else
      throw operation exception
else
   throw type exception
```

Exceptions:

Type exception, operation exception, and code load exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

OSIZE n

Description:

OSIZE (Object Size Directive) is a compiler directive that uses the “hint” opcode format to inform ADAM how large a region needs to be allocated for a particular thread object. The size of the region to allocate in words is indicated in n. This opcode may be located anywhere, but it only has meaning when it is in the entry point instruction sequence for an object’s initializer code. When executed, this instruction does nothing to the machine state except increment the PC.

Operation:

```
PC = PC + 1
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

HINT t,hint

Description:

`HINT` (Compiler Hint) is a hint from the compiler or programmer to the ADAM runtime system. A `HINT` instruction has no effect on the ADAM machine state except for incrementing the `PC`; however, it may have a profound impact upon the OS and/or management coprocessor.

The type of hint is encoded in the `t` field, and the actual value of the hint is encoded in the `hint` field. The valid hint types are TBD, but they fall into two broad categories: machine specific and machine independent. Machine specific hints include data placement directives. Machine independent hints include thread swap hints, prefetch directives, and migration hints. A hint with an unrecognized hint type is ignored.

An incorrect hint never leads to incorrect program results; an incorrect just leads to poor performance.

Operation:

```
PC = PC + 1
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

NOP

NOP

Description:

`NOP` (No Operation) A `NOP` instruction has no effect on the ADAM machine state except for incrementing the `PC`.

Operation:

```
PC = PC + 1
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

Bibliography

[BGKH00] Jeremy H. Brown, J.P. Grossman, Tom Knight, and Andrew Huang. A capability representation with embedded address and near-exact object bounds. In *Submitted to ASPLOS 2000*, 2000.

[Ste85] David Stevenson. Ieee standard for binary floating-point arithmetic. ANSI/IEEE standard 754-1985, August 1985.