Q Instruction Set Architecture

John A. Watlington

FEB 2000 DRAFT

1 Introduction

This is a specification for Q, an instruction set architecture (ISA) for a parallel processing system supporting stream-based hybrid dataflow, such as MagicEight [6][1].This architecture is motivated primarily by a need for flexible and scalable parallel media processing. Other motivations include supporting specialized processors, capable of executing a restricted set of algorithms much more efficiently than a general purpose processor, and a decoupled memory access mechanism.

Q is an attempt to define an architecture that leverages the properties of dataflow machines to support parallelism, while retaining the advantages of sequential machines when executing singly. The programming model proposed minimizes the synchronization costs by synchronizing blocks of operations upon blocks of data, and allowing the size of the data blocks to be decided at runtime to match the available machine granularity.

In order to aid the task of describing the Q ISA, a virtual machine implementing the architecture will be presented, along with an architectural overview. The program elements are each described in depth, followed by an example program using Q program elements and its evaluation in Section 6. Another approach to describing the instruction set, without the use of a virtual machine, may be found in Section 7. A programming language for describing algorithms using elements of the Q instruction set is described in an appendix.

1.1 A Virtual Machine

The virtual machine used for this description contains multiple components: a control machine and one or more data machines. The control machine is solely concerned with performing the scheduling of computations on the data

[1]This work is being done at the MIT Media Laboratory. The author may be contacted at `wad@media.mit.edu`. A current version (HTML and Postscript) of this document, along with other MagicEight information, may be found at: `http://magiceight.www.media.mit.edu/projects/magiceight/`

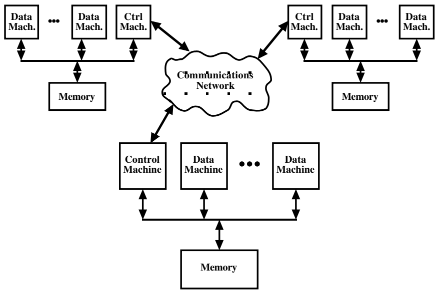

Figure 1: The Virtual Machine

machines and managing the system resources. The control machine synchronizes and provides arguments for lower level functions. The data machines perform all the "useful" computation done by the system. A shared memory is provided, in addition to any local state included in each data machine. One of the goals of Q is to support a decoupled memory access mechanism, where the data processor's access of data memory is decoupled from the data processing.

In order to control this architecture, two levels of algorithm specification are used. The upper level is specified as a dataflow graph (DFG) to maximize the available functional parallelism, and is interpreted solely by the control machine. The lower level is executed by a data machine, with control specified in a manner native to the implementation (serial, fixed, dataflow, associative, etc...) It is assumed that functions ranging up to several hundred primitive arithmetic or logical operations will be implemented in the lower level. More complex operations should be implemented as dataflow graphs (at the upper level) to allow parallelization of their constituent operations.

Q specifies the instruction set architecture used for the upper level of algorithm specification, including a definition of the interface between the two levels. The instruction set architectures of the data machines (the lower level) is varied and implementation specific.

One simple implementation of the virtual machine realizes the control machine and a data machine as alternating threads executing on a single processor. A parallel version of the virtual machine may have multiple control machines performing the evaluation of the dataflow graphs, each controlling a small set of data machines. Communication between the individual clusters (each with a single control machine) is performed explicitly by the control machines.

1.2 Dataflow Graph Components

The dataflow graphs used to represent the upper level of execution are composed of *streams* (the arcs) and *tasks* (the nodes). A task is the application of a *function* to a set of input and output streams. A compositing *graph* operator is provided for representing a function using a dataflow graph. Except for streams, dataflow graph elements are considered immutable once loaded, in order to simplify their duplication and distribution.

1.2.1 Functions

A *function* is an operation that takes some number of stream elements as input parameters, some number of bound dimensions, and produces one or more output stream elements as output parameters. Functions are not allowed to input or output any data except through formally bound parameters – no "side-effects" are allowed. Likewise, a function is not allowed to reference along a dimension not formally bound to it. See Section 3 for a longer description.

A function refers to a specific program element which may have either or both an upper and lower level of algorithm representation. The upper level representation is a *graph*, and a lower level representation will referred to as a *sequential function*.

1.2.2 Streams

A *stream* is a sequence of *stream elements*, which are each a tuple of value and location. Each stream element may only have a single value assigned to it. An element may also have no value assigned, in which case the element is considered undefined — no operation using the value of the element may be performed. All elements of a stream have the same scalar type (see section 1.3). The location is an N-tuple of integers, specifying the element's position in an N-dimensional space. A stream is said to *span* each of the dimensions in its N-dimensional space.

A stream may be written using a particular sequence of stream elements, and read (possibly multiple times) using a different sequence of elements. Its elements may be produced and consumed "out-of-order". The number of stream elements in the stream may be unbounded. In particular, the set of defined locations along one or more dimensions may be unbounded. In a realizable implementation, only a [small] finite extent along most dimensions will be defined at any one time. Streams are discussed further in Section 2.

A *stream fragment* is the set of stream elements with locations inside a particular multidimensional volume. For simplicity, the Q ISA defines this volume as an orthotope — a multidimensional generalization of a rectangular parallelepiped. Thus a stream fragment is defined as elements of a stream having a location within the multidimensional volume defined by an integer starting position, step, and extent along each dimension spanned by the stream.

The set of stream elements of a single stream accessed by a single invocation of a function is the stream *access pattern* of the function on that stream. This access pattern is specified in the N-dimensional space defined by the function's bound dimensions, and identifies the data dependencies of a function invocation. Stream access patterns always contain at least one location, and may have a larger extent along any bound dimension. They are defined in a manner similar to stream fragments, and are discussed further in section 3.1.

1.2.3 Tasks

A *task* is the application of a particular function to a particular list of bound parameters and dimensions. It is the dataflow graph element which produces or consumes streams.

1.2.4 Graphs

A *graph* is a compositing operator, allowing complex operations to be specified as a dataflow graph composed of streams and tasks. It may be specified as an implementation of a function (other possible implementations include, for example, object code for a particular processor or configuration data for a reconfigurable processor).

Unlike sequential function implementations, the internal data objects (streams) in a graph are embedded in the algorithm representation. This requires special considerations when "calling" a function represented by a graph, which are described in Section 4.

1.3 Data Types

In order to allow the transparent integration of processors with differing representations for basic data types, Q uses strong data typing. Polymorphism, the ability to represent data of varying type, is not currently supported. Neither is the user able to define composite data structures. The following data types are supported:

- `unsigned integer` – Unsigned integer representation, in sizes of 8, 16, 32, and 64 bits.
- `signed integer` – Two's complement signed integer representation, in sizes of 8, 16, 32, and 64 bits.
- `float` – (IEEE) Floating Point, in sizes of 32 and 64 bits.
- `boolean` – a Boolean value, true or false

Individual data machine architectures are free to determine how each data type is actually represented. A common example of a representation difference is the storage byte-order, or endianess, used for multi-byte data types. Translation between the different representations of a single type is automatically performed as needed, simplifying the programming model and enabling a heterogeneous mix of data machines in a system.

1.4 Naming

Q uses a system-wide, machine-independent namespace. Each dataflow graph element in an algorithm specification — streams, tasks, functions, and graphs — is contained in a *named object* in this space. The *tag*, or name, is a unique 64-bit identifier.

Regardless of the mechanism used to implement the basic tag to named object pointer mapping, there are advantages to using a multilevel mapping scheme. In Q a two level mapping is provided by an *environment* mechanism. An environment is a named object which contains information about the location of other named objects. A tag cannot be resolved unless the tag identifying its environment is also specified. Thus the cost of a Q name is typically 128-bits.

The lower level of algorithm specification uses the local processing node's namespace, which is implementation specific. All references to function parameter stream data are done through pointers in the local namespace, resolved from tags (and context) by the runtime system before calling the function.

1.5 Evaluation Model

It is expected that a Q dataflow graph will be evaluated in a dynamic demand-driven, or lazy, method. While incurring additional computational latency, such a method attempts to minimize the total amount of computation performed. We refer to demand-driven evaluation as *eduction.* The "iteration number" used to identify an instance of a task is the multidimensional location of the task's output in its output streams, or its *context.* The number of dimensions used to define this context is algorithm dependent.

Throttling of computation to avoid overwhelming the control machine will be performed in three ways: a demand for computation will be partitioned at runtime only as finely as is needed to provide sufficient parallelism for the executing machine; demands for infinite stream extents are explicitly throttled, usually to match associated real time input/output rates; and if in the process of eduction the control machine resources are exhausted, the demand being educed is simply deferred until some are freed.

```
int8 One = 1;
int8 Line.x,y = {{ x=0 }{x}{ 1 }};
int8 Plate.x,y,t,f = {{ x=0+512, y=0+512, f=0 }{x,y,f}{ 1 }};
```

Figure 2: Example Stream Initializations

2 Streams

Q supports multidimensional streams (defined in sec. 1.2.2) as the basic mechanism for data storage and communication. These streams may be viewed as a mapping of a multidimensional array of stream elements into a one dimensional sequence, such as a linear memory address or a transmission over a serial communication channel. Streams are used both for algorithm specification and at the interface between the upper and lower levels of algorithm specification.

In the upper (or DFG) level of algorithm specification, streams represent data being output from a single function, and input to one or more other functions. A mutable program variable (*e.g.* those used in a conventional programming model) may be modeled as a stream by adding a dimension which represents the various assignments over time.

Synchronization between a stream producer and consumers is effectively performed on each individual stream element. For efficiency's sake, however, synchronization in the Q architecture is almost always performed on larger sets of stream elements.

The number of stream elements synchronized as a unit should be determined by the control machine at runtime. The partitioning should generate only sufficient parallelism to occupy all the data machines available. Excessively fine partitioning generates undesirable synchronization overhead. In the worst case, it can overload the control machines resources. On the other hand, an excessively coarse partitioning won't take advantage of the available computational resources. This partitioning may be done in a semi-static manner to minimize overhead yet allow execution to adapt to changing resources.

2.1 Stream Initialization

A stream in a DFG must have certain characteristics specified by the algorithm: the data type of the stream element, and the dimensions explicitly spanned by the stream.

A fragment of a stream may also be specified as having a pre-defined value — one known when the program was written. This is used to define the constants and stream initial conditions used by an algorithm. Some examples are shown in Fig. 2, using the Q language described in Appendix

A[2].

The first example declaration is for a stream, `One`, with no defined dimensions — a scalar, or a single stream element with a defined value of one (and a non-existent location). It will have a value of one for all stream elements in a stream fragment demanded of it. The second declaration, `Line`, is for a stream with two dimensions in which all stream elements with a position of zero along the x dimension are defined and have a value of one. The third one, `Plate`, specifies a four dimensional stream, with a two dimensional boundary condition specified in a three dimensional space. Predefined stream fragments will be "replicated indefinitely" as needed along all stream dimensions not specified in their definition (such as y in `Line`, or t in `Plate`) by a runtime mechanism.

3 Functions

A function represents an operation upon a set of stream elements (the bound input parameters), applied over a set of bound dimensions, which produces one or more output stream elements as bound output parameters. The function is the interface between the dataflow graph level of algorithm specification and the lower level (which is data machine dependent.)

While generally equivalent, a Q function isn't necessarily identical to the "function" of a typical high level language. It may instead represent one portion of a high level function, a "code fragment" or sequence of operations which together access only a small set of parameters, using temporary local storage to intercommunicate intermediate values. Previous work on hybrid dataflow architectures has partitioned these based on any access to non-register memory. Q assumes that data machines will also have relatively exclusive (deterministic) access to larger memories, allowing input and output stream elements to be passed by reference. This relaxation of the code partitioning requirements allows coarser grained (larger) partitions.

Functions are restricted to accessing bound parameters (streams) and whatever local storage (registers, stack) a particular data machine may provide (*i.e.* no "side effects" are allowed). Any local storage is assumed undefined upon function entry, unless used by the data machine ISA for function arguments.

Another restriction is that sequential function implementations must terminate in bounded time. Such a function is viewed as a fundamental unit of execution, which is never "context-switched" as all required input data and output data space is by definition available "local" to the processing node — accessible with a deterministic latency, known at compile time of the function implementation.

[2] There are more elegant ways of declaring these stream initializations in the Q language. Fig. 2 illustrates the canonical method actually defined by the instruction set architecture.

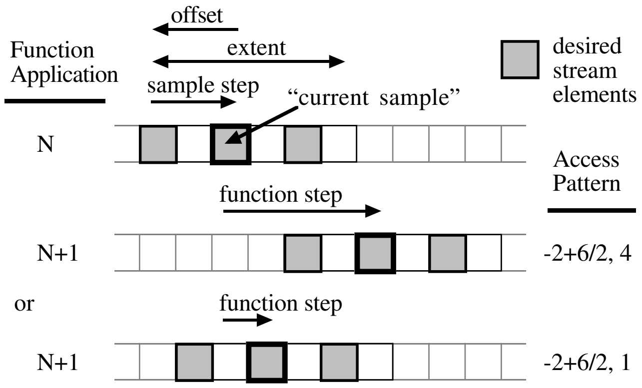

Figure 3: A Stream Access Pattern

Sequential function evaluation in Q is strict, in that all function arguments must be present for function evaluation. If non-strictness is desired in an algorithm (and it usually is), a conditional primitive, `mcond` is provided at the DFG level. A function with a graph implementation will be evaluated in a non-strict manner.

A Q function may be defined as applying over zero or more dimensions, which are specified when the function is applied. These bound dimensions are analogous (and in addition) to the bound variables of a traditional function. All stream fragments generated for a sequential function implementation will have the information about particular dimensions ordered appropriately: bound dimensions are first, and in the same order as their declaration. It is important that a function specify all dimensions accessed; otherwise the function has no way of correctly accessing (indexing) the parameters describing the data along that dimension.

Q functions typically reside in shared libraries. These libraries may have function implementations (compiled object code or configuration data) for multiple machine architectures, including specialized ones. A description of the function, containing the information necessary for proper scheduling — the function's stream access patterns and available machine implementations along with their performance — is contained in the function object. The actual function implementations, which are located separately, may be dataflow graphs, or sequential functions: compiled object code or configuration data for particular machine architectures.

3.1 Stream Access Patterns

A function reads one or more stream elements from each input stream and writes one or more stream element to each output stream. In order that functions may be executed in a non-blocking manner, the Q ISA requires that the set of stream elements, or stream access pattern, accessed in one function invocation be identified for each stream input or output parameter.

An access pattern spans a specified number of a function's bound dimensions, and is specified with locations relative to a function specific origin, typically located within the set. Nulladic[3] input or output parameters are located at this origin, which represents the "current" location when a function is invoked.

In order to minimize the complexity and size of the access pattern representation, it is defined as being the "bounding box" of the locations of all set elements for a given parameter. More specifically, it is the orthotope[4] that includes all accessed locations. An access pattern defines the following parameters per specified dimension:

- *offset* indicates where the start of the access pattern along a dimension is located relative to the "current" location.
- *extent* indicates the size of the access pattern, in distance along the dimension — not actual number of samples, which is indicated by *extent* divided by *step*. A negative *extent* indicates that the region starts at *offset* (including it) and extends in the earlier direction.
- *sample step* indicates the increment between adjacent samples along this dimension. If zero, the same value will be accessed *extent* times.
- *function step* indicates the increment between adjacent applications of the function, along this dimension. If zero, then the offset is absolute (not relative) and the indicated region is the only one of interest.
- *dimension* indexes which of the function's bound dimensions this dimension in the access pattern represents.

The order in which dimensions are defined in an access pattern indicates the desired ordering of the sequence of stream elements accessed by the function. This information is required when a processor utilizes a decoupled memory access mechanism.

It is a common situation for a stream access pattern to be *variable*, or dependent on the actual data being input. In this case, the access pattern specifies the bounding box of the union of possible sets of stream elements. Upon function termination, the stream fragments passed as parameters will be modified to communicate to the runtime system the number of elements actually consumed along each dimension. Functions with variable access

[3] The "Greek" terms (monadic, dyadic, etc...) will be used when referring to the number of dimensions, and the "Latin" terms (unary, binary, etc...) will be used to refer to the number of input parameters, as suggested by Ashcroft, *et al* [2]

[4] An orthotope is a multidimensional generalization of the rectangular parallelepiped. It is a point, line, rectangle, and rectangular parallelepiped in zero, one, two, and three dimensions, respectively.

patterns must be explicitly parallelized in an application specific manner, as there is no general solution to the data dependency problem posed. Several Q primitive functions (*e.g.* `resample`, see sec. 5.2.3) support access patterns which vary as functions of another function input parameter.

3.2 Pipelined Function Application

There are a number of common functions that operate in a data locality which is changed only incrementally between invocations. If the function implementation has the data machine store locally input values which will be reused, and only requires that new data (not previously used) be input, the function implementation is *pipelined*[5]. Pipelining is very attractive when using specialized stream processors, where local state is relatively cheap and memory bandwidth is expensive; but may also be used to support software optimizations such as loop unrolling.

The size of the overlap along each dimension between function invocations is indicated by the absolute magnitude of the extent minus the absolute magnitude of the step. If the overlap is non-zero, the control machine will attempt to pipeline the task, thereby amortizing the cost of transferring the overlap over as many applications as possible. Pipelining is possible when the function has an implementation which supports it, and it can be scheduled to a data machine that has sufficient local storage to store the input and intermediate values that it will reuse. When partitioning a demand, the runtime system should take into account the amount of local memory available on a data machine, and partition it so as to allow the use of pipelined function implementations if possible.

Both software and hardware pipelining are supported by Q. If a function implementation is marked as supporting pipelining, it will be executed once to generate a stream fragment, instead of being executed once for each output access pattern in the stream fragment. The stream fragment may only span dimensions which the function has explicitly declared (bound dimensions). It is the responsibility of the function to apply itself as necessary to generate all the locations specified in its output stream fragment(s).

3.3 Q Primitives within Functions

In order to access stream data from within a sequential function implementation, several primitive operations are supplied. The simplest of these primitive operations (`curr`) takes a stream fragment pointer and provides a pointer to the value of the "current" stream element in an input or output stream. These primitive operations all operate exclusively using the information provided to a function in the stream fragments describing its input

[5] I'm considering changing this term to *serialized*, due to the differences with the established definition of "pipelined". Comments ?

and output stream parameters.

3.3.1 Intensional Data Access

Unlike the streams traditionally used in programming, which only support access to one stream element, a Q function may access the set of stream elements described by its stream access pattern. In order to select particular elements within this set, several primitive operators are provided.

The `next` intensional operator, which references data advanced one step from the current position along a specified dimension, is supported within sequential functions. A more general form, `rel`, equivalent to multiple serial applications of the `next` operator, and a `prev` operator, which references data one step earlier along a dimension, are also provided.

3.3.2 Context Query

A context query operator (`context` or `#`) is supported for use within sequential functions. The absolute context (or position) along a specified dimension is returned. The dimension must be one of the function's bound dimensions.

3.3.3 Direct Addressing

A direct addressing operator (`at` or `@`) is used to index an arbitrary element of a stream along a specified dimension. In its elemental form, it provides stream programming with data dependent data access — any data access may be to any element of the source stream. It is available within a sequential function in a limited form: a function must bound the application of the operator, communicating the bounds to the control machine in its predefined stream access patterns.

4 Graphs

A graph is a compositing operator, allowing complex operations or those which require non-strict interpretation to be specified as a dataflow graph composed of streams and tasks. All functions with a sequential implementation may also be specified using a graph representation. But not all graphs can have a semantically equivalent sequential function representation, due to their strict application.

Sequential function implementations internally use temporary local variables which are not shared between invocations. A function implemented with a graph instead uses streams for internal variables. In order to allow multiple invocations of a graph to be executed simultaneously (and minimize overhead), these streams persist between invocations.

Maintaining the identity of data stored in streams internal to a graph is handled two ways, both performed before a graph may executed. A unique copy (or instantiation) of a graph is used for each task which references it. Multiple invocations of a single task calling a graph are differentiated by extending the multidimensional space of each stream internal to the graph. A graph which has been prepared for execution in this manner is referred to as *expanded*.

Input and output stream parameters to a graph are represented by "dummy" streams in the dataflow graph structure. All references to these are replaced with the references to the streams actually bound to them by a task when a graph is expanded.

A graph declares all dimensions, both bound and internal, which are used by its component stream and tasks. Dimensions are represented in graph components by indices into a graph dimension list. When a graph is expanded, each stream must have additional dimensions defined to represent the function "invocation space". These are the dimensions over which the function representing the graph is being applied, with the exception of the ones already bound to the function.

When a program is loaded into the global namespace, functions implemented solely with a graph are expanded (*i.e.* the program is "flattened".) In the absence of cyclic data dependencies, this produces a single graph with maximal functional parallelism. If a graph calls itself (recurses), it may be automatically transformed into a non-recursive graph of higher dimensionality when expanded [3][4].

5 Tasks

In Q, a *task* represents the application of a particular function (or graph) to a particular set of bound input and output parameters. Multiple tasks may apply the same function, with a different set of input and output parameters.

A task is usually executed multiple times, to produce output stream elements with varying context. If the function referenced by a task produces one stream element per invocation, and five stream elements are demanded from one of the task's output streams, it will be executed five times.

A normal task consists of the following:

- The name of the function — its tag once a program is loaded.
- A list of dimensions to apply the function over
- A list of function parameters (both input and output)
- A list of any non-data forward dependencies

Most tasks represent the application of a user or library defined function. There are, however, two special classes of tasks defined in the Q ISA. One is tasks which perform application input or output (I/O), and the other class is primitive tasks which provide functionalities not available to ordinary tasks

— such as conditional execution or data remapping. These special classes differ from normal tasks in how the function name is specified and how they are handled by the control machine.

5.1 I/O Tasks

Support for application input and output is provided in Q through the mechanism of I/O tasks. These identify functions which must be scheduled with care. They have three relevant characteristics:

1. Unlike typical Q functions, they modify external state over which the runtime system has no control — they have side-effects.
2. A number of I/O tasks are synchronous: they represent devices which require known amounts of stream elements at specific real-time rates.
3. Most I/O tasks need to be executed on a particular set of nodes in a system.

Output tasks force the eduction of a program. Since the output of a program identifies what is "useful" computation, a program's evaluation is driven by periodic demands for its output tasks' input streams. The demand period is determined by the output device.

5.2 Primitive Tasks

There are a small number of tasks proposed in order to provide dataflow graph semantics not possible with normal tasks. These *primitive tasks* provide conditional evaluation, and stream subset specification, as well as regular linear remappings along a dimension.

The primary justification for identifying these as tasks, when they might be better classified as functions, is that in most cases they do not actually result in any computation being done on data machines. They are instead applied by the control machine as filters to demands for data during the evaluation of a dataflow graph.

5.2.1 MCond

The Multiple Conditional, or `mcond`, task provides a mechanism for conditional evaluation of input data. In a normal task, all the input parameters must be present in memory readily accessible by a data machine before the task will be allowed to execute. An mcond task requires that all the parameters used to evaluate the conditional expressions be defined (present in memory) before determining the winning condition and demanding the required input stream. Input streams not required for the winning condition are never demanded[6]. It is equivalent to the following expression:

[6] This assumes that speculative execution of conditional branches is never desired.

```
mcond = if( cond1 ) then stream1;
            else if( cond2 ) then stream2;
            else if ...
          fi;
```

The conditions are currently limited to the following form: `param1 <op> param2` where the parameters may be either constants or streams. The monadic context query constant (`#`) may be one of the parameters. The operation must be an arithmetic comparison operation — equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal. The parameters being compared must be of equivalent type, there is no automatic type casting. There is no explicit default assertion. The programmer can include one as the last conditional pair, with a condition of true (`1 eq 1`).

The `mcond` task provides a means of combining multiple streams into a single one. If the conditions all have the form `#.d <op> constant`, the output is predefined and the `mcond` operation has little impact on performance.

5.2.2 Select

The `select` task provides a dual to the `mcond` task. It is used to select a subset of a stream along a single dimension, and also, through the use of special values for offset, to synchronize a stream along two dimensions.

A `select` task specifies a multidimensional access pattern to be applied to a single stream, producing a single stream. In a common form, `select` is used to choose a single location along a dimension (direct addressing), eliminating the dimension from further streams and processing.

The range is either the following parameters passed as constants, or a stream with the following information provided as separate integers along a specified dimension :

- *offset* - The beginning of the region, along this dimension
- *extent* - The extent of the region, in elements (not units)
- *step* - The step, or number of units between elements along this dimension. A *step* of zero along a dimension indicates that the data at *offset* should be replicated *extent* times.

A range is simply another multidimensional access pattern descriptor. Several "special" values are available for use when specifying a range that aren't typically used in normal function access patterns. They are:

- An *offset* of negative infinity (STREAM_NEG_INFINITY) indicates that the "earliest" data along that dimension is desired.
- An *offset* of positive infinity (STREAM_POS_INFINITY) indicates that the "latest" data along that dimension is desired.

- A infinite *extent* (STREAM_INFINITE) may be indicated.

These special values are generally used for one of two purposes. The first is referencing the limits of a data stream along a dimension for which the limits are relatively static yet unknown at compile-time. Once the amount of data along the dimension is specified, it doesn't change. The second use is referencing the limits of a data stream which is growing over time. In this case, the `select` task synchronizes one dimension with another. Further discussion of these infinite values along with examples is provided in a separate article[7].

5.2.3 Resample

The Q `resample` task provides a limited form of the generic stream indexing operator (`@`). It is useful for either shifting or decimating a stream along a particular dimension. The output stream consists of one out of every *scale* elements in the input stream. The offset and scale parameters must be integers. More formally:

```
resample.d( x, scale, offset )  = x @.d ((#.d * scale) + offset);
```

An example of the resample function would be:

```
pipe1:    orig_stm = src( prev_stm );
          new_stm = resample.x( orig_stm, 2, 0 );
          ... = dst1( orig_stm );
          ... = dst2( new_stm );

pipe2:    orig_stm = src( prev_stm );
          new_stm = resample.x( orig_stm, 2, 0 );
          ... = dst1( new_stm );
          ... = dst2( new_stm );
```

In `pipe1`, the `resample` primitive (in this case simply taking every even sample along dimension x) will be incorporated into the input of the `dst2` task. In `pipe2`, the filter will be incorporated into the output of `src`, in which case it might affect the demands made on `prev_stm`.

Several common intensional operators are implemented at the DFG level using the `resample` task:

```
first.d x = resample.d( x, 0, 0 );
next.d x  = resample.d( x, 1, 1 );
prev.d x  = resample.d( x, 1, -1 );
rel.d( x, n ) = resample.d( x, 1, n );
```

[7] *A Discussion of Positive and Negative Infinity*

The `resample` and `upsample` tasks have stream access patterns that vary depending on other parameters. For simplicity this capability is currently limited to these primitives, although it could be utilized by user-defined functions. The output stream access pattern is scalar, as are the scale and offset parameter accesses. The data input parameter stream access pattern for `resample` has an offset equal to the demanded output location along the specified dimension times the function's scale parameter plus the offset parameter ((`#.d` * scale) + offset), an extent of one, and step equal to the scale parameter.

5.2.4 Upsample

The Q `upsample` task provides another limited form of the generic stream indexing operator (`@`) operator. It is useful for interpolating a stream along a particular dimension. It may be viewed as performing the same basic operation as `resample` with a scale which is the reciprocal of the integer provided. This task produces an upsampled stream with either the samples replicated, or with "new" samples zeroed (zero-padded).
A more exact definition follows. The divides are integer (as is all the math).

```
upsample.d( x, scale, offset, ctl )  =
      if( scale == 0 )
          then x @.d offset;
      else if( ctl == REPLICATE )
          then  x @.d ((#.d / scale) + offset);
      else if(((#.d - 1) / scale) == (#.d / scale))
          then 0;              // ZERO-PAD
      else x @.d ((#.d / scale) + offset);
      fi;
```

The `upsample` primitive has a stream access pattern that varies depending on other input parameters. The `x` input parameter stream access pattern for `upsample` has an offset equal to the demanded output location along the specified dimension divided by the function's scale parameter plus the offset parameter ((`#.d` / scale) + offset), an extent of one, and step equal to the scale parameter. The output stream access pattern is scalar, as are the scale and offset parameter accesses.

6 An Example Program

A simple program illustrating the use of the Q instruction set architecture may be represented by the dataflow graph in Fig. 4. This program inputs an image from a file, and outputs a single level of a gaussian pyramid decomposition of the image. This is a repeated filtering and downsampling along

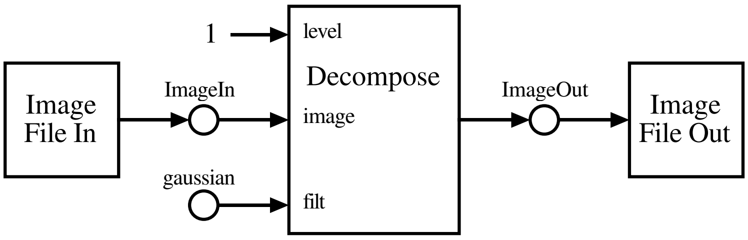

Figure 4: An Example Program

both image dimensions. Assuming an image size of $M \times N$ for the zeroeth level (the input image), the first level of the pyramid will have a size of $M/2 \times N/2$, with succeeding levels having sizes of $M/4 \times N/4$, $M/8 \times N/8$, etc. . .

In the dataflow graphs, circles are used to represent streams, and rectangles represent tasks, usually labeled with the name of the function they apply. The `Decompose` function, which calculates a particular level of the pyramid, may be described by the dataflow graph shown in Fig. 5. The `Decompose` graph is shown in Fig. 5. It applies the `Down` function along each spatial dimension at each level to generate the next level of the pyramid.

A text version of this program is shown in Fig. 6, written in the Q language described in Appendix A. There are four dimensions declared. The first two (x and y represent the spatial dimensions of the image. The third (f) represents filter coefficients, and the c dimensions spans the characters in a filename.

Five streams are declared at the top level of the program. Two of these represent the images being input and output, and only span the image spatial dimensions. The third, `gaussian`, represents the filter being used for decomposition and is statically initialized. It only spans the filter coefficient dimension. The last two streams, `InFile` and `OutFile` represent the names of the files used for input and output.

The `Decompose` graph declares one internal dimension, d, which represents a level within the gaussian pyramid. The `Output` is defined as being the input image at position zero along the d dimension, followed (at higher positions along d) by a decomposition of the previous image along d[8].

The `Down` graph, shown in Fig. 7 separably filters then decimates the image along each of the spatial dimensions[9].

The `filt1D` function is declared at the top level, but not defined at this time. It represents a Q sequential function in this program.

[8]The `fby` operator can be confusing. It is important to remember that the second term is evaluated at the current position along the dimension to yield the next value along the dimension.

[9]While this separable filter does not exactly implement a two-dimensional gaussian filter, this is a common image processing optimization. More importantly, it allows me to show a one-dimensional filter routine being applied along different dimensions.

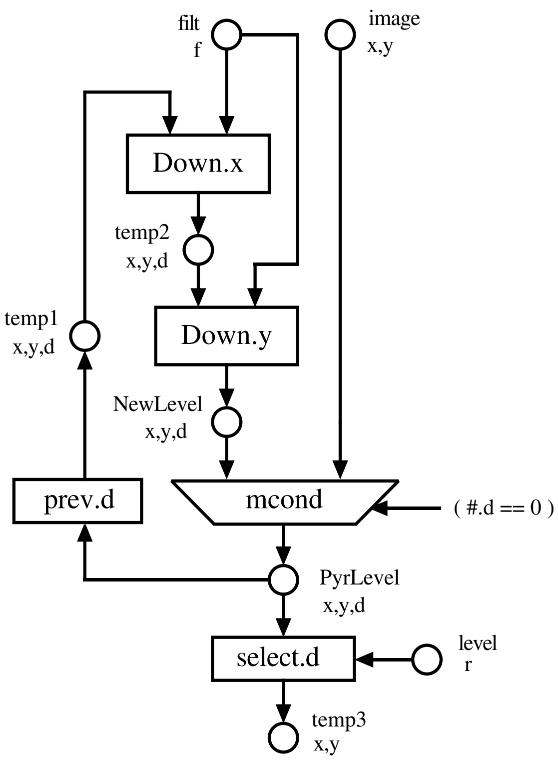

Figure 5: The Decompose Graph

6.1 The Expanded Program

An expanded version of this program is shown in Fig. 8. In this version, the `Decompose` and `Down` graphs have been flattened. Their formal parameters have been replaced by their actual ones, and the dimensionality of their internal streams has been increased. In addition, as the first operand of the `fby` operator was not a constant, it has been replaced with an `mcond` operation. The constant passed as the level argument to the `Decompose` function has been converted into the `level` stream.

6.2 Program Eduction

Although the book-keeping associated with the eduction of even a simple program can appear voluminous, it is worth following to its conclusion, as it eventually reduces down to a simple set of required tasks. We will start with the assumption that nothing has been executed — there are no stream elements (other than constants) defined. The eduction of the ex-

```
dimension  x, y, f, c;
int16 ImageIn.x,y;
int16 ImageOut.x,y;
int16 gaussian.f = {{f=0+3} {f=0+3} {48, 160, 48}} ;
int8  InFile.c = {{c}{c} {"test_image.raw"}};
int8  OutFile.c = {{c}{c} {"out_image.raw"}};

access_pattern filt1D_In   { fixed, { -1+3, 1, x }};
access_pattern filt1D_filt { fixed, {  0+3, 0, f }};
int16 filt1D.x,f( int16 filt1D_In, int16 filt1D_filt );

ImageIn = ImageFileIn.x,y,c( InFile );
ImageOut = Decompose.x,y,f( ImageIn, 1, gaussian );
void = ImageFileOut.x,y,c( OutFile, ImageOut );

Decompose.a,b,e( image, level, filt ) = PyrLevel @.d level
  where
    dimension  d;
    int16  NewLevel.a,b;
    int16  PyrLevel.a,b;

    PyrLevel = image fby.d NewLevel;
    NewLevel = Down.y,f( Down.x,f( PyrLevel, filt ), filt );
    Down.m,p( In, filter ) = resample.m( Low, 2, 0 );
      where
        int16  Low.m;
        Low = filt1D.m,p( In, filter );
      end;
  end;
```

Figure 6: An Example Program

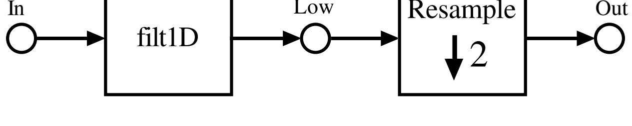

Figure 7: The Down Graph

```
dimension  x, y, d, f, c;
int16  ImageIn.x,y;
int16  NewLevel.x,y,d;
int16  PyrLevel.x,y,d;
int16  ImageOut.x,y;
int16  Low1.x,y,d;
int16  Low2.x,y,d;
int16  temp1.x,y,d;
int16  temp2.x,y,d;
int32  level.r = {{r=0+3} {r=0+3} { 1, 1, 1 }};
int16  gaussian.f = {{f=0+3} {f=0+3} {48, 160, 48}} ;
int8   InFile.c = {{c}{c} {"test_image.raw"}};
int8   OutFile.c = {{c}{c} {"out_image.raw"}};
access_pattern filt1D_In   { fixed, { -1+3, 1, x }};
access_pattern filt1D_filt { fixed, {  0+3, 0, f }};
int16 filt1D.x,f( int16 filt1D_In, int16 filt1D_filt );

ImageIn = ImageFileIn.x,y,c( InFile );
temp1 = resample.d( PyrLevel, 1, -1 );      //  prev.d Output
Low1 = filt1D.x,f( temp1, gaussian );
temp2 = resample.x( Low1, 2, 0 );
Low2 = filt1D.y,f( temp2, gaussian );
NewLevel = resample.y( Low2, 2, 0 );
PyrLevel = mcond ((( #.d == 0 ) ImageIn )   //  In fby.d NewLevel
                 (( 1 == 1 )   NewLevel ));
ImageOut = select.d,r( PyrLevel, level );   //  PyrLevel @.d level
void = ImageFileOut.x,y,c( OutFile, ImageOut );
```

Figure 8: The Flattened Example Program

panded program (see Fig. 8) begins with a demand for a stream fragment, say `[ x=10+100, y=10+100 ]`[10], of the `ImageOut` stream.

If all the stream elements demanded of a stream are already defined, the demand is satisfied. If demanded stream elements are undefined, the task sourcing of the stream (if it has one) needs to be executed. The `ImageOut` stream is output by a `select` task. The demand for the task output "translates" first into a demand for `[ r=0+3 ]` of the `level` stream. If such an operator "control" input is unavailable the required computation

[10] The context, or stream fragment describing a demand will be denoted using a shorthand notation: a list of dimension descriptors, separated by commas and enclosed by brackets. Each dimension descriptor has a syntax of: *dimension* '=' *offset* '+' *extent* '/' *step*. If the *extent* or *step* are one, they may be omitted along with the preceding separator.

Task	Demanded Context
ImageFileIn(InFile)	`[ x=9+200, y=9+200 ]`
filt1D.x,f(temp1, gaussian)	`[ x=10+200/2, y=9+200, d=1 ]`
filt1D.y,f(Low1, gaussian)	`[ x=10+100, y=10+200/2, d=1 ]`
ImageFileOut(OutFile, ImageOut)	`[ x=10+100, y=10+100 ]`

Table 1: Actual Program Tasks

is scheduled, and the eduction of the second demand deferred pending its availability. In this case, it is readily available as it has been pre-defined as a program constant. The second demand is for `[ x=10+100, y=10+100, d=1 ]` of `PyrLevel`. The `mcond` operator sourcing `PyrLevel` has a simple set of conditions, comparing a constant (the context operator) against another constant. It may be completely evaluated when a demand is made. As this demand is for stream elements with a non-zero position along the d dimension, it becomes a demand on `NewLevel` for `[ x=10+100, y=10+100, d=1 ]`

The `NewLevel` stream is sourced by a `resample` task, which is decimating its input by a factor of two along the y dimension. The demand on its input stream, `Low2`, is `[ x=10+100, y=10+200/2, d=1 ]`. This stream is generated by filtering `temp2` from which `[ x=10+100, y=9+200, d=1 ]` is demanded. A demand is also made of `gaussian` for the filter kernel `[ f=0+3 ]`.

We now repeat these operations, applying them over the x dimension instead: The `resample` task sourcing `temp2` requires `[ x=10+200/2, y=9+200, d=1 ]` of `Low1`. This in turn requires `[ x=9+200, y=9+200, d=1 ]` of `temp1`, and `[ f=0+3 ]` of `gaussian` again.

The source of `temp1` is a `prev` operation (implemented using `resample`) applied along the d dimension. It transforms the demand into one for `[ x=9+200, y=9+200, d=0 ]` of `PyrLevel`. As discussed above, its source is an `mcond` task. In this case, a demand is made for `[ x=9+200, y=9+200 ]` of the `ImageIn` stream.

After all those demands have been calculated, the actual work required can be identified. In this case, there are only four tasks that will actually be executed (shown in Table 1), and only three streams (`ImageIn`, `Low1`, and `Low2`) are actually utilized. Any stream which is sourced by a conditional or intensional operator (*e.g.* `mcond`, `select`, or `resample`) will probably be used only for eduction, and then ignored along with the associated operator during execution. Such a stream will not have any stream elements defined at runtime.

7 Q and Lucid

Now that the Q programming model has been introduced using a virtual machine model, the formal underpinnings can be discussed. It is possible to describe the Q instruction set architecture without resorting to a machine description. Q is in many ways an architecture for implementing a high-level programming language, Lucid. Q provides an implementation of Lucid which is reduced is some ways and extended in others. Examination of these differences may suggest possible areas for improvement.

The multidimensional Lucid language proposed by Ashcroft, Faustini, Jaggannathan, and Wadge [2] is a intensional, stream-based, multidimensional programming language, derived from Landin's ISWIM [3]. It is an extension of earlier work with the original one-dimensional version of Lucid [5], *ferd* Lucid, and an early attempt at multidimensionality [1]. Hereafter, the Lucid referred to is the "new" multidimensional Lucid [2] unless otherwise noted.

Lucid extends the simple applicative expressiveness of ISWIM with an intensional logic using a multidimensional space. The single variable of a simple algebra is extended into a stream which spans any dimensions over which the variable changes. The use of Lucid as a language for describing stream-based programming is warranted for several reasons. It provides a firm theoretical foundation for streams — their definition and manipulation. While Q makes little use of it, much work has been done on transformation and proof of Lucid programs.

Equally important is its use of intensional referencing, which promotes development of algorithms supporting stream (data independent) processing. Limited access to variable values "at other times and places" is allowed through intensional (relative) addressing primitives.

The requirements that Q places upon functions are a minor divergence from Lucid. In Lucid, functions are allowed to access streams and dimensions which are defined either globally or in `where` clauses including the function definition. This difference facilitates proper implementation of the hybrid dataflow model, and is mainly a syntactical difference.

7.1 Supported Lucid Operators

The following section describes three basic Lucid operators: stream initialization, context query and indexing, along with their Q equivalents.

7.1.1 Followed By

The Lucid "followed-by" (`fby`) operator is replaced with two different mechanisms. In the case that the first parameter (the "first" value) is a constant, the basic stream definition suffices, with the second parameter specified as the source of the stream. If the first parameter contains a variable (stream), an `mcond` statement must be used.

7.1.2 Context Query

The Lucid context query operator (`#`) is supported for use within Q functions. See section 3.3.2 for a description. Currently there is no Q DFG equivalent to `#`, although it may be used in `mcond` conditional statements.

7.1.3 Direct Addressing

The Lucid direct addressing operator, `@`, is used to index a particular subset of a stream, along a specified dimension. It is available within a Q function in a limited form, see section 3.3.3. It is not directly supported at the Q DFG level, except in the form of special cases such as `resample` (see section 5.2.3).

7.1.4 Intensional Data Access

The Lucid `next` intensional operator, which references data one step further along a specified dimension, is supported for use within Q functions. A more general form, `rel`, equivalent to multiple serial applications of the `next` operator, is also provided.

Q operators available within functions may be defined in terms of the Lucid `#` and `@` primitives as follows:

```
next.d x   = x @.d (#.d + 1);
prev.d x   = x @.d (#.d - 1);
rel.d( x, n)  = x @.d (#.d + n);
```

The Lucid `first` operator always returns the first value (or subspace) along the specified dimension. As such, it isn't really intensional! It is not supported within Q functions.

All of these primitives (`first`, `next`, `rel`, and `prev`) may be used in Q DFGs. They are obtained as special cases of the resample task (see section 5.2.3).

7.2 Unsupported Lucid Operators

The following Lucid operators are currently unsupported by the Q instruction set architecture:

7.2.1 As Soon As

The "as soon as" operator, `asa`, returns the "first" data value along the specified dimension for which the control value is non-zero. Note that it returns a constant. The value returned is the same in all contexts, once the control value is non-zero in the "first" (in an implementation-dependent manner) value. Here is a theoretical recursive Lucid implementation:

```
data asa.d control = if control then data;
                        else next.d data asa.d next.d ctl fi;
```

The `asa` operator has largely been replaced in Q by support for positive and negative infinity in the range of a `select` task.

7.2.2 Where-ever

The "wherever" operator, `wvr`, returns a stream of data values selected by an associated non-zero control value. An intensional implementation using the `@` operator is defined below. In it, `t1` is a stream containing the location of the next selected value, and `t2` is a stream containing a single entry for each selected value.

```
data wvr.d ctl = data @.d t2
                          where
                            t1 = if ctl then #.d;
                                        else next.d t1  fi;
                            t2 = t1 fby.d t1 @.d t2 + 1;
                           end;
```

A limited form of the `wvr` operator is provided by the Q `select` task, where `range` is a vector of parameters defining a stream subset along a particular dimension:

```
select.d( data, range ) =
                  data wvr.d is_contained( #.d, range );

select.d,e,f( data, range_d, range_e, range_f ) =
                ((data wvr.d is_contained( #.d, range_d ))
                       wvr.e is_contained( #.e, range_e ))
                       wvr.f is_contained( #.f, range_f );
```

7.2.3 Upon

The "advanced upon" operator, `upon`, returns a stream of data values selected by an associated non-zero control value. The input/output rate is unbounded…

Here is an implementation using the `@` operator:

```
data upon.d ctl = data @.d t1
                       where
                         t1 = 0 fby.d if ctl then t1 + 1;
                                             else t1  fi; end;
```

It should be possible to implement both `upon` and `wvr` operators as C functions, which operate within bounds...

8 Future Directions

These are the areas of the Q ISA which will possibly change in the future.

8.1 Variable Data Types

The strong type restrictions may be alleviated to some extent by preprocessing which includes type checking and coercion. A polymorphic version of the architecture could be specified, with an increase in implementation complexity. Perhaps automatic type casting during task execution when appropriate would be a compromise. This is left for further work.

8.2 Direct Addressing

The Lucid direct addressing operator, `@`, is used to index a particular subset of a stream, along a specified dimension. While it is available within a Q function in a limited form (see section 3.3.3) it is not directly supported at the dataflow level, except in special cases such as `resample`.

A direct addressing primitive in the dataflow graph level might to be needed, but an unbounded version is impractical with a demand driven evaluation model. Instead a relatively general, yet bounded version may be adopted.

References

[1] Edward A. Ashcroft, Anthony A. Faustini, and Rangaswamy Jagannathan. An intensional language for parallel applications programming. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 2, pages 11–49. Addison-Wesley, Reading, MA, 1991. QA76.642.P37 1991 in Barker Lib.

[2] Edward A. Ashcroft, Anthony A. Faustini, Rangaswamy Jagannathan, and William W. Wadge. *Multidimensional Programming.* Oxford Univ. Press, New York, 1995.

[3] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, March 1966. On microfiche in Barker Lib.

[4] David Skillicorn. Stream Languages and Data-Flow. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 16, pages 439–454. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[5] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Harcourt Brace Jovanovitch, Pubs., New York, 1985. QA76.73.L83.W33 1985 in Barker Lib.

[6] John A. Watlington and V. Michael Bove, Jr. A system for parallel media processing. In *Proc. of the Workshop on Parallel Processing in Multimedia: IPPS '97*, Geneva, April 1997.