1.0 Assembly Language Specification

The Imagine Microcode Assembly Language (μasm) is to be used for programming kernels to be run on the array of arithmetic clusters in Imagine. The language is meant not only to be a straightforward assembly language using a C-like expression syntax that can be handled by the Imagine microcode scheduler, but to also to include enough expressive power that programmers do not need to manage the hardware resources.

1.1 Types

μasm uses data types to reduce the number of operators, as operators are overloaded based on type, and to enable type checking.

1.1.1 Basic Types

Table 1 lists the currently supported basic types.

TABLE 1. Basic Types

(U)INT	32-bit (un)signed integer
(U)HALF2	2 packed 16-bit (un)signed half words
(U)BYTE4	4 packed 8-bit (un)signed bytes
FLOAT	IEEE format single precision, 32-bit floating point
CC	4-bit boolean

In the operation descriptions in Section 2, integer operators are assumed to operate on both signed and unsigned types unless otherwise specified.

1.1.2 Record Types

User-defined record types are also supported. Records are used to combine basic types for programming convenience and concise transfers using communication or streams. The record type declaration is as follows:

```
record name {
    type name;
    ...
};
```

Once a record type is defined in a file, it can be used as a new type. The "." operator is used to access the fields of a record for individual use.

1.1.3 Type Qualifiers

Type qualifiers modify a type for a special purpose. Type qualifiers cannot be combined, applied to the basic type CC, or applied to record fields. Only the stream qualifiers can be applied to record types. The type qualifier syntax is:

type_qualifier <type>

Table 2 lists the type qualifiers' syntax and their meanings.

TABLE 2. Qualifiers

(C)ISTREAM	(Conditional) input stream composed of the qualified type
(C)OSTREAM	(Conditional) output stream composed of the qualified type
UC	Microcontroller variable of the qualified type
DOUBLE	Two concatenated instances of the qualified type
ARRAY	Array of the qualified type

The (C)ISTREAM and (C)OSTREAM qualifiers specify input and output streams (series of elements analogous to vectors) composed of the qualified type. Values are sequentially read from or written to a stream using special operators. A new value is read from/written to a conditional stream only if a specified condition is true in the reading/writing cluster. The stream qualifiers can only be used for kernel parameters (see below).

The UC qualifier specifies that a variable is located in the microcontroller registers. There is only one instantiation of these registers, unlike the registers contained within the clusters. A UC variable may only be transferred to/from other types of variables by use of one of the communication instructions.

The DOUBLE qualifier specifies a variable consisting of two instances of the base type concatenated together. A double variable, consisting of two variables, X and Y, can also be constructed via the following syntax:

```
HI_LO(X, Y)
```

The ARRAY qualifier is used to specify an array of elements of the base type. The size of the array is specified in parentheses after the name of the array, for example:

```
ARRAY<INT> foo(10);
```

The PERSISTENT_ARRAY qualifier is also used to specify an array elements, but the created array persists in a single scratchpad location for the duration of a kernel. The order of allocations depends only on the order of the persistent array declarations in a kernel. By matching the order of declarations in multiple kernels, this qualifier allows separate kernels to pass data through the scratchpad.

1.2 File Structure

A µasm file is composed of any number of record type declarations followed by any number of kernel declarations.

1.2.1 Preprocessor Directives

Currently, there are no preprocessor directives. Future development to the microcode scheduler should add a C-like macro/include file facility.

1.2.2 Comments

µasm supports both C and C++ style comments. The “//” identifier signifies that all other text to the end of the line is a comment. The “/*” and “*/” identifiers signify the start and end, respectively, of a potentially multiline comment.

1.3 Kernels

A Kernel is essentially a function that operates on streams. Kernels are defined using µasm, but are called from the Imagine *Macro*code Assembly Language. Thus, unlike normal functions, kernels cannot be nested. The basic structure of a kernel is as follows:

kernel *name* (*parameter_type*[&] *name*, ...)
{
    body
}

Parameter type may be any valid type. If a stream qualifier is not used as part of the type, it must be followed by an &. The kernel body may be composed of arbitrary variable declarations, control flow, and operations. In practice, a kernel usually has one or more input and output stream parameters and consists of a some initial and final code surrounding a central loop which iterates over the input streams and produces the output streams.

1.4 Variable Declarations

Variable declaration syntax is as follows:

type name [= *initial value*];

where *type* is any valid type (excepting stream qualifiers).

1.4.1 Initial Values

If *type* is a microcontroller (UC) qualified basic type an initial constant value may be specified, which will be stored in the microcode word as an immediate, for example:

```
    UC<UBYTE4> VAL = 0xA580;
```

For basic types, initial values may be any valid expression. Therefore, the following two expressions are equivalent:

INT VAL = *a + b -3;*

and

INT VAL*;*
VAL = *a + b - 3;*

1.5 Control Flow

All µasm control flow is in the form of loops. All loops check the looping condition before the loop is entered, and after every loop iteration thereafter. There are no conditional branch instructions. There are three kinds of loops available:

1.5.1 Count Loops

In this form of looping, the loop is repeated until a counter (which must be of type UC<INT>), which is post-decremented once for each iteration, becomes zero. Syntax is:

loop_count(*counter*) { *loop body* }

1.5.2 Stream Loops

In this form of looping, the loop continues an input stream (a parameter qualified by (C)ISTREAM) has transferred all of its data to the clusters. This ensures that the loop body will be executed for all of the elements of input stream. Syntax:

loop_stream(*input_stream)* { *loop body* }

1.5.3 Conditional Loops

In this form of looping, the loop continues until some combination of values is achieved in a CC variable. Syntax:

loop_while_any(*loopcc*) { *loop body* }

loop_while_all(*loopcc*) { *loop body* }

loop_until_any(*loopcc*) { *loop body* }

loop_until_all(*loopcc*) { *loop body* }

The combination in each case should be obvious, e.g. loop_while_any executes loop body *while* loopcc is true in *any* cluster.

1.5.4 Loop Optimization

The scheduler supports the UNROLL hint for all loops. Right before the open brace of any loop, the hint UNROLL(*n*) can be placed to tell the scheduler to unroll the loop body *n* times. For count loops, it is important to adjust the loop counter accordingly, as each iteration will now actually execute the loop body *n* times.

The scheduler also supports the PIPELINE hint for all loops. Right before the open brace of any loop, the hint PIPELINE(*n*) can be placed to tell the scheduler to software pipeline the loop body with *n* iterations. Within the loop body the directive "STAGE (*i*);" must be used to tell the scheduler that all instructions following this directive should be placed in pipeline stage *i*, up until the next STAGE directive. Note that the STAGE directives do not need to occur in numerical order, and valid stage numbers are from 1 to *n*. All instructions inside a software pipelined loop default to stage 1 until the first STAGE directive.

1.5.5 Inline Functions

See *iscd-functions.txt* for preliminary information.

1.6 Operations

All operations are overloaded based on variable type. For the INT types, the operations perform full 32-bit integer operations, if applicable. For the FLOAT type, the operations perform full 32-bit floating point operations including alignment and normalization unless otherwise specified. For the HALF2 type, the operations perform two completely separate operations on each of the two 16-bit half words, if applicable. For the BYTE4 type, the operations perform four completely separate operations on each of the four 8-bit bytes.

No implicit type conversion is performed, i.e. an INT cannot be added to a BYTE4 or even an UNSIGNED INT. However, explicit type casting (with no conversion) is allowed between integer types. Other type conversion can only be performed with an explicit conversion operation (i.e. ITOF, CCTOI, etc.).

Unless otherwise specified all operations which take inputs of type INT, HALF2, and/or BYTE4 can also operate on their unsigned counterparts.

Refer to the *Imagine Instruction Set Architecture* for more specific details on the semantics of each operation.

1.6.1 ADD

Format	x + y
Input Types	INT, FLOAT, HALF2, BYTE4
Output Type	matches input type

Description. ADD computes the sum of x and y based on their types. For integer and floating point types, a modulo addition is performed. For the half word type, the bottom half words are added together, and the 16-bit result forms the bottom half word of the result. Independently, the top half words are added together to form the top half word of the result. For the byte type, four independent byte-wide additions are performed to produce four independent byte-wide results to make up the full 32-bit result.

1.6.2 SATURATING ADD

Format	addsat(x, y)
Input Types	INT, HALF2, BYTE4
Output Type	matches input type

Description. SATURATING ADD computes the sum of x and y with saturation based on their types. For the integer type, a saturating addition is performed. For the half word type, the bottom half words are added together with saturation, and the 16-bit result forms the bottom half word of the result. Independently, the top half words are added together with saturation to form the top half word of the result. For the byte type, four independent byte-wide saturating additions are performed to produce four independent byte-wide results to make up the full 32-bit result.

1.6.3 SUB

Format	x - y
Input Types	INT, FLOAT, HALF2, BYTE4
Output Type	matches input type

Description. SUB computes the difference of x and y based on their types. For integer and floating point types, a modulo subtraction is performed. For the half word type, the bottom half words are subtracted, and the 16-bit result forms the bottom half word of the result. Independently, the top half words are subtracted to form the top half word of the result. For the byte type, four independent byte-wide subtractions are performed to produce four independent byte-wide results to make up the full 32-bit result.

1.6.4 SATURATING SUB

Format	subsat(x, y)
Input Types	INT, HALF2, BYTE4
Output Type	matches input type

Description. SATURATING SUB computes the difference of x and y with saturation based on their types. For the integer type, a saturating subtraction is performed. For the half word type, the bottom half words are subtracted with saturation, and the 16-bit result forms the bottom half word of the result. Independently, the top half words are subtracted with saturation to form the top half word of the result. For the byte type, four independent byte-wide saturating subtractions are performed to produce four independent byte-wide results to make up the full 32-bit result.

1.6.5 ABS

Format	abs(x)
Input Types	INT, FLOAT, HALF2, BYTE4
Output Type	matches input type

Description. ABS computes the absolute value of x.

1.6.6 ABD

Format	abd(x, y)
Input Types	INT, HALF2, BYTE4
Output Type	matches input type

Description. ABD computes the absolute difference of x and y.

1.6.7 Bitwise Logical Operations (AND, OR, XOR, NOT)

Format	x & y, x | y, x ^ y, ~x
Input Types	INT, HALF2, BYTE4
Output Type	matches input type

Description. These four operations perform bitwise logical and, or, xor, and not respectively. The result is the same, regardless of the input type, as the operations are performed on a bitwise basis. Note that if all inputs are the output of a comparison operation (EQ, NEQ, LT, LE) then these functions are effectively normal logical operations that respect the integer input type boundaries.

1.6.8 Comparison Operations (EQ, NEQ, LT, LE, GT, GE)

Format	x == y, x != y, x < y, x <= y, x > y, x >= y
Input Types	INT, FLOAT, HALF2, BYTE4
Output Type	INT, HALF2, and BYTE4 => matches input type, FLOAT => INT

Description. These six operations perform the comparsons for equality, inequality, less than, less than or equal to, greater than, and greater than or equal to, respectively. For integer and floating point inputs, the result of this test is a bitmask of all 0’s or all 1’s, where all 0’s indicates false, and all 1’s indicates true. For the halfword and byte types, the comparison is performed on a component by component basis, and each component of the result will contain a bitmask of all 0’s or all 1’s, based on the test for that component.

1.6.9 SELECT

Format	select(p, x, y)
Input Types	p: CC x, y: INT, FLOAT, HALF2, BYTE4
Output Type	matches type of inputs ‘x’ and ‘y’

Description. SELECT chooses x if p is true and y if p is false. Both x and y must be of the same type. If x and y are of type HALF2 or BYTE4, then the individual components are selected seperately, using the multiple bits of the CC p.

1.6.10 MUL

Format	x * y
Input Types	INT, FLOAT, HALF2, BYTE4
Output Type	DOUBLE integer input type, or FLOAT

Description. MUL produces the result of multiplying x and y. For integer and floating point types, the obvious function is performed. For the halfword and byte types, multiplication is performed on a component by component basis and the products are packed together into the result, with the components of the high word of the result being the packed high halves of the multiplications, and the components of the low word of the result being the packed low halves of the multiplications.

1.6.11 MULD

Format	muld(x, y)
Input Types	HALF2, BYTE4
Output Type	DOUBLE INT, DOUBLE HALF2

Description. MULD produces the result of multiplying x and y. Multiplication is performed on a component by component basis and the products are packed together into the result. The difference from the MUL instruction is the output word format. The high and low halves of the result of the multiplications are stored in the same word, rather than being split across the high and the low word. So, the result of a MULD on half words is two full words, with the high word being the 32-bit result from multiplying the high half words together, and the low word being the 32-bit result from multiplying the low half words together. For bytes, the high 16-bits of the high word of the result is the product of the high bytes of the input words, and so on.

1.6.12 MULRND

Format	mulrnd(x, y)
Input Types	INT, HALF2, BYTE4
Output Type	DOUBLE integer input type

Description. MULRND produces the result of multiplying x and y. For integers, the high word of the result is the high rounded 32-bits of the 64-bit product, and the low word of the result is the low saturated 32-bits of the 64-bit product. For the halfword and byte types, multiplication is performed on a component by component basis and the high rounded halves of the result are packed into the high word, and the low saturated halves of the result are packed into the low word. The RNDM() and SATM() functions can be used to select the appropriate result from the DOUBLE return value.

1.6.13 DIV

Format	x / y
Input Types	INT, FLOAT, HALF2, BYTE4
Output Type	DOUBLE integer input type, or FLOAT

Description. DIV produces the double precision result of dividing x and y. For integer formats, a 64-bit result is produced as a 32-bit quotient and a 32-bit remainder.For the halfword and byte types, division is performed on a component by component basis, and the quotients of the results are packed together into the high word produced by this instruction, while the remainders of the results are packed together into the low word produced by the instruction.

1.6.14 FSQRT

Format	fsqrt(x)
Input Types	FLOAT
Output Type	FLOAT

Description. FSQRT produces the floating point square root of x.

1.6.15 SHIFT (logical)

Format	shift(x, y)
Input Types	x: INT, HALF2, BYTE4 y: INT
Output Type	matches type of input ‘x’

Description. SHIFT shifts x by y bits. If y is a positive number, x is shifted left; if y is negative, x is shifted right. For the integer types, the result is the 32-bit value resulting from x being shifted by y. For the halfword and byte types, each component of the result is the value obtained by shifting the corresponding component of x by y bits.

1.6.16 SHIFTA (arithmetic)

Format	shifta(x, y)
Input Types	x: INT, HALF2, BYTE4 y: INT
Output Type	matches type of input 'x'

Description. SHIFTA shifts x left by y bits to the left if y is positive, and to the right if y is negative. The sign of the result is maintained by shifting 1's into the upper bits of the result on right shifts if the input was negative. For the integer types, the result is the 32-bit value resulting from x being shifted right by y. For the halfword and byte types, each component of the result is the value obtained by shifting the corresponding component of x by y bits.

1.6.17 ROTATE

Format	rot(x, y)
Input Types	x: INT, HALF2, BYTE4 y: INT
Output Type	matches type of input 'x'

Description. ROTATE rotates x by y bits. If y is positive then x is rotated left; if y is negative then x is rotated right. For the integer types, the result is the 32-bit value resulting from x being rotated by y. For the halfword and byte types, each component of the result is the value obtained by rotating the corresponding component of x by y bits.

1.6.18 SHUFFLE

Format	shuffle(x, y)
Input Types	x: any integer type y: BYTE4
Output Type	matches type of input 'x'

Description. SHUFFLE performs a byte reordering operation on the input x based on the control information in the input y. The component bytes of the output word can be independently selected from the input word x with the following options:

0. byte 0 of the input x

1. byte 1 of the input x

2. byte 2 of the input x

3. byte 3 of the input x

4. fill with msb of byte 0 of the input x

5. fill with msb of byte 1 of the input x
6. fill with msb of byte 2 of the input x
7. fill with msb of byte 3 of the input x
8. zero

This operation allows a variety of functions, such as sign extend from a smaller type to a larger type, permute, broadcast, pack from a larger type to a smaller type, and many others. Byte 0 corresponds to the low order byte, and byte 3 is the high order byte. Each byte of the output word is selected by the corresponding control byte of the input y.

1.6.19 SHUFFLED

Format	shuffled(x, y)
Input Types	x: any integer type y: BYTE4
Output Type	DOUBLE type of input 'x'

Description. SHUFFLED performs a byte reordering operation on the input x based on the control information in the input y, just as SHUFFLE does. The difference is that shuffled produces two outputs. The low nibble of each control byte controls the result of the low output word, and the high nibble of each control byte controls the result of the high output word. The control values are exactly the same as in the SHUFFLE instruction.

1.6.20 FTOI

Format	ftoi(x)
Input Types	FLOAT
Output Type	INT

Description. FTOI takes a 32-bit floating point number and converts it to an integer using truncation. If the magnitude of the floating point number is too large to be represented as an integer, then the result is clamped at +/- MAXINT depending upon the sign of the input.

1.6.21 FRAC

Format	frac(x)
Input Types	FLOAT
Output Type	FLOAT

Description. FRAC takes a 32-bit floating point number and returns the fraction that would be left over if x were converted to an integer. In other words, FRAC(x) = x - ITOF(FTOI(x)).

1.6.22 ITOF

Format	itof(x)
Input Types	INT
Output Type	FLOAT

Description. ITOF takes a 32-bit integer and converts it to a 32-bit floating point number. The input cannot be unsigned.

1.6.23 ITOCC

Format	assign an INT to a CC
Input Types	INT, HALF2, BYTE4
Output Type	CC

Description. When a variable of type CC is assigned to from a signed or unsigned integer, the low bit of each byte of the input is used to set the value of the CC variable.

1.6.24 CCTOI

Format	assign a CC to an INT
Input Types	CC
Output Type	INT

Description. When a variable of type CC is assigned to a signed or unsigned integer, each bit of the input is replicated to mask each byte of the output.

1.6.25 Type Cast Operators

Format	type(x), AsInt(y), AsFloat(z)
Input Types	x: INT, HALF2, BYTE4 y: FLOAT z: INT, HALF2, BYTE4
Output Type	type

Description. The type cast operators change the type of an integer variable to a different integer type. No data conversion is performed. AsInt(y) and AsFloat(z) can be used to treat floats as ints and ints as floats with no data conversion.

1.6.26 HI

Format	hi(w)
Input Types	w: DOUBLE<BASIC TYPE>
Output Type	BASIC TYPE

Description. HI returns the high word of a double type.

1.6.27 LO

Format	lo(w)
Input Types	w: DOUBLE<BASIC TYPE>
Output Type	BASIC TYPE

Description. LO returns the low word of a double type.

1.6.28 RNDM

Format	rndm(w)
Input Types	w: DOUBLE<HALF2 or BYTE4>
Output Type	HALF2 OR BYTE4

Description. RNDM returns the rounded multiply result of a double type that was the result of a MULRND instruction. This is an alias to the HI instruction for greater program readability.

1.6.29 SATM

Format	satm(w)
Input Types	w: DOUBLE<HALF2 or BYTE4>
Output Type	HALF2 OR BYTE4

Description. SATM returns the saturated multiply result of a double type that was the result of a MULRND instruction. This is an alias to the LO instruction for greater program readability.

1.6.30 CHECK_OVF

Format	check_ovf(x)
Input Types	x: FLOAT
Output Type	INT

Description. CHECK_OVF returns true if overflow has occured, false otherwise.

1.6.31 CHECK_UNF

Format	check_unf(x)
Input Types	x: FLOAT
Output Type	INT

Description. CHECK_UNF returns true if underflow has occured, false otherwise.

1.6.32 COMMUCPERM

Format	commucperm(perm, x) commucperm(perm, x, srcidx, y)
Input Types	perm: UC INT x: any integer type, FLOAT srcidx: constant y: UC of same type as ‘x’
Output Type	same type as input ‘x’

Description. COMMUCPERM performs intercluster communication as determined by *perm*. *perm* specifies which cluster’s ‘x’ input will be read by each cluster. The least significant nibble of *perm* contains the source index for cluster 0, and so on up to the most significant nibble, which contains the source index for cluster 7. Each source index is a number between 0 and 7, which corresponds to the source cluster. If specified, srcidx specifies which cluster’s data will be stored in ‘y’.

1.6.33 COMMCLPERM

Format	commclperm(perm, x)
	commclperm(perm, x, y)
	commclperm(perm, x, srcidx, z)
	commclperm(perm, x, y, srcidx, z)

Input Types	perm: INT x: any integer type, FLOAT y: UC of same type as ‘x’ srcidx: constant z: UC of same type as ‘x’
Output Type	same type as input ‘x’

Description. COMMCLPERM performs intercluster communication as determined by *perm*. *perm* specifies which cluster’s ‘x’ input will be read by each cluster. Only the least significant nibble of this value is used by each cluster. Each source index is a number between 0 and 8, which corresponds to the source cluster -- 8 indicates the micro-controller, or input ‘y’. If specified, srcidx specifies which cluster’s data will be stored in ‘z’.

1.6.34 INPUT

Format	istr >> x;
	istr: ISTREAM<TYPE>
Input Types	x: TYPE
Output Type	none

Description. INPUT reads a value from input stream ‘istr’ and stores it in ‘x’. The “>>” operator can be cascaded to input multiple values, similar to the C++ operator>> semantics.

1.6.35 OUTPUT

Format	ostr << x;
	ostr: OSTREAM<TYPE>
Input Types	x: TYPE
Output Type	none

Description. OUTPUT writes a value to output stream ‘ostr’ from ‘x’. The “<<” operator can be cascaded to output multiple values, similar to the C++ operator<< semantics.

1.6.36 CONDINPUT

Format	cistr(p, ccend) >> x;
	istr: CISTREAM<TYPE>
	p: CC
Input Types	ccend: CC
	x: TYPE
Output Type	ccend: CC

Description. CONDINPUT reads a value from input stream ‘istr’ and stores it in ‘x’. Each cluster can evaluate the validity of the data received with the ‘ccend’ value. ‘ccend’ is true if ‘p’ is true and valid data was not stored to ‘x’, otherwise it is false. In other words, ‘ccend’ is false in a cluster if that cluster didn’t request a piece of data or if the stream delivered valid pieces of data. The “>>” operator can be cascaded to input multiple values, similar to the C++ operator>> semantics.

1.6.37 CONDOUTPUT

Format	costr(p) << x;
	costr: COSTREAM<TYPE>
	p: CC
Input Types	x: TYPE
Output Type	none

Description. If ‘P’ is true, CONDOUTPUT writes a value to output stream ‘ostr’ from ‘x’. The “<<” operator can be cascaded to output multiple values, similar to the C++ operator<< semantics.

1.6.38 FLUSH

Format	flush(costr, x);
	costr: COSTREAM<TYPE>
Input Types	x: TYPE
Output Type	none

Description. FLUSH pads ‘costr’ to force the stream to have a length that is a multiple of 8. The value of ‘x’ is used to pad the stream. FLUSH must be called as the last operation on a conditional output stream. For predictable results, the value of ‘x’ should be the same in all clusters.

1.6.39 CID

Format	cid();
Input Types	none
Output Type	INT

Description. The special function *cid()* can be used to obtain the cluster’s number. This function will return a different number (0-7) for each of the eight clusters, and can be used to perform calculations that require knowledge of which cluster they are being computed on.

1.6.40 UCID

Format	ucid();
Input Types	none
Output Type	INT

Description. The special function *ucid()* can be used to obtain the microcontroller’s index. This function will return a number that can be used in communication permutations to address the microcontroller.

2.0 Sample Microprogram

The following is one possible unoptimized implementation of the FFT algorithm, to illustrate the use of the microcode assembly language:

```
// Vector radix two FFT
// 1. assumes twiddle factors are in order
//
// Ujval Kapasi
// 10/15/97
// 12/1/97
// 5/21/98

// This program needs two operands for the butterfly, a and b.  These should
//   correspond to elements 0..(N/2-1) and (N/2)..N of the input respectively
//   for the first pass.  Subsequently, the streams a,b should be the same
//   parts of the stream output by the previous pass of this loop.
// Twiddle factors must also be provided evey loop iteration in the correct
//   order

// Let a, b, and w be the input vectors.
//   ISTREAM[0] = a[0..511]    in_a
//   ISTREAM[1] = b[0..511]    in_b
//   ISTREAM[2] = w[0..511]    in_w

// Let c be the output vector
//   OSTREAM[0] = c[0..1023]   out

// cluster        :  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
// first output   : dr3 | cr3 | dr2 | cr2 | dr1 | cr1 | dr0 | cr0 |
// second output : dr7 | cr7 | dr6 | cr6 | dr5 | cr5 | dr4 | cr4 |

record COMPLEX {
  FLOAT r;
  FLOAT i;
};

kernel fft8c(istream<COMPLEX> in_a,
             istream<COMPLEX> in_b,
             istream<COMPLEX> in_w,
             ostream<COMPLEX> out)
{
  UC<INT> perm_e = 0x37261504;
  UC<INT> perm_f = 0x73625140;

  INT odd = cid() & 1;
  INT even = 1 - odd;

  CC low = itocc(cid() < 4);
```

```
ARRAY<FLOAT> sel_array_real(2), sel_array_imag(2);

loop_stream(in_a) pipeline(5) {

  stage(1);

  // Read in 2 complex data points

  COMPLEX a, b, w;

  in_a >> a;
  in_b >> b;
  in_w >> w;     // twiddle factor

  // Perform butterfly calculations

  complex c, d;

  c.r = a.r + b.r;
  c.i = a.i + b.i;

  b.r = a.r - b.r;
  b.i = a.i - b.i;

  stage(2);

  FLOAT wrbr = w.r*b.r;
  FLOAT wrbi = w.r*b.i;
  FLOAT wibr = w.i*b.r;
  FLOAT wibi = w.i*b.i;

  stage(3);

  d.r = wrbr - wibi;
  d.i = wrbi + wibr;

  stage(4);

  float e, f;

  // want this eventually
  // cluster        :  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
  // first output   : dr3 | cr3 | dr2 | cr2 | dr1 | cr1 | dr0 | cr0 |
  // second output : dr7 | cr7 | dr6 | cr6 | dr5 | cr5 | dr4 | cr4 |

  e = select(low, d.r, c.r);
  sel_array_real[even] = commucperm(perm_e, e);
  f = select(low, c.r, d.r);
  sel_array_real[odd] = commucperm(perm_f, f);
```

```
      e = select(low, d.i, c.i);
      sel_array_imag[even] = commucperm(perm_e, e);
      f = select(low, c.i, d.i);
      sel_array_imag[odd] = commucperm(perm_f, f);

      stage(5);

      COMPLEX out0, out1;

      out0.r = sel_array_real[0];
      out0.i = sel_array_imag[0];
      out1.r = sel_array_real[1];
      out1.i = sel_array_imag[1];

      out << out0 << out1;
    }
  }
```

Note that this kernel takes 3 input streams and produces one output stream, which must be passed as parameters to the kernel. All record declarations must precede the kernel declaration.

Two arrays and a set of communication instructions are used to reorganize the output of the radix-2 butterfly such that this kernel can be used as-is for subsequent passes over the data. Therefore, to perform an *N*-point FFT, the original two input streams, in_a and in_b, are passed through the kernel, and the results recirculated through the kernel for a total of log *N* passes to produce the final results. The results must be stored back to memory using a bit-reversed store, as they are in bit-reversed order.

The loop is software pipelined for illustration. There are five pipeline stages, which are enumerated with “stage” directives inside the body of the loop.