Imagine Programming System
User's Guide

Peter Mattson

Ujval Kapasi

John Owens

Scott Rixner

Table of Contents

1.0 Introduction

This document describes how to develop applications for the Imagine Media Processor using the Imagine Programming System. Imagine is a stream processor: it operates on sequences of data records called streams. Applications written for Imagine have two parts: kernels that define operations on streams and a stream program that defines the streams and the high-level control and data flow between kernels.

1.1 Roadmap

This document is divided into chapters that address each part of the development process for an Imagine application:

Section 2.0 of this document describes the components of the Imagine toolset, how to configure Microsoft Visual C++ for use in developing an Imagine application, and how to structure the source files that compose the application.

Section 3.0 describes the KernelC language used to write kernels.

Section 4.0 describes the StreamC language used to write stream programs.

Section 5.0 describes the IScd kernel scheduler used to compile kernels for execution on Imagine.

Section 6.0 gives an overview of the run-time tools: the simulators and the profile compiler.

Section 7.0 describes the IDebug functional simulator used in conjunction with Microsoft Visual C++ to debug the application.

Section 8.0 describes the ISim cycle-accurate simulator used to execute the application.

Section 9.0 described the IStream profile compiler built in to the Imagine simulators.

Section 10.0 describes the SchedViz application visualizer used to analyze and optimize the application.

Section 11.0 deals with advanced topics, including dealing with kernels that fail register allocation and using IStream with Verilog.

Each section concludes with one step in a short exercise that develops a simple application for Imagine.

2.0 Setup

2.1 Toolset Files

To ready an Imagine for toolset for use, unpack (or check out from sourcesafe) the toolset into a “working” directory. The resulting directory structure is shown below. The directory im_apps contains all imagine applications. Paths used within this document assume that any new application is located in its own directory within im_apps. The tools directory contains the source files for the Imagine tools, each in its own directory.

```
working
    im_apps
         <one directory for each application>
    tools
        iscd
        isim
          isimdll
          isimhostdll
          isimexe
              blank_headers
        SchedViz
```

The Imagine toolset contains the essential files listed in Table 1. These files are located at the path given in Table 1.

To ready the toolset for use, do the following (in this order):

1. Set the environment variable “iscd_preproc” equal to a fully qualified path to the file CL.EXE, e.g. “C:\Program Files\Microsoft Visual Studio\VC98\Bin\cl.exe”, and “iscd_includes” to a fully qualified path to the blank_headers directory, e.g. “c:\working\tools\isim\isimexe\blank_headers”.
2. Add the tools\isim\isimexe\Release, tools\isim\isimexe\Debug, and tools\iscd\Release directories to your system path.
3. Add paths to bison.exe and flex.exe in the Visual C++ GUI under “Tools->Options->Directories->Executable Files.” These executables can be obtained from the Cygwin distribution (www.cygwin.com). Note: these executables are only necessary for tool development -- application developers need not provide these files.
4. Open the project tools\tools.dsw in Visual C++ and build the Release and Debug versions of isimexe.exe and the Release version of iscd.exe.

TABLE 1. Essential Files

File	Description	Location(s)
gold8.md	Machine description for the Imagine architecture, used by kernel scheduler and simulator	im_apps
main.cpp	The standard main file required for all Imagine applications	tools/isim/isimexe
*.hpp	Header files required for developing Imagine applications	tools/ tools/isim/ tools/isim/isimdll tools/isim/isimhostdll
*.hpp	Blank version of some header files used by preprocessor for kernel scheduler	tools/isim/isimexe/blank_headers
iscd.exe	The kernel scheduler (IScd)	tools/iscd/Release
isimhost.dll	The combined functional simulator, cycle-accurate simulator, and profiling compiler (IDebug, ISim, and IStream).	tools/isim/isimexe/Release
isimcore.dll		tools/isim/isimexe/Release
isimhost.lib	Libraries required to link application for use with above dlls	tools/isim
isimcore.lib		tools/isim
isimhostdbg.dll	Debug versions of the above.	tools/isim/isimexe/Debug
isimcoredbg.dll		tools/isim/isimexe/Debug
isimhostdbg.lib		tools/isim
isimcoredbg.lib		tools/isim
SchedViz.exe	The schedule visualizer, written in and compiled using Visual Basic	tools/SchedViz/SchedViz.exe

2.2 Creating a project

To develop an Imagine application, create a new “Win 32 Console Application” project in Visual Studio located in a new directory within the im_apps directory. Then, change the following project settings:

1. C/C++, General, Preprocessor Definitions:
   add _USRDLL_IMP,_USRDLL2_IMP
2. C/C++, Preprocessor, Additional Include Directories
   add ..\..\tools\isim\isimdll, ..\..\tools\isim\isimhostdll, ..\..\tools\isim
3. Link, Input, Additional library path:
   add ..\..\tools\isim
4. Link, Input, Object/Library Modules:
   add isimcore.lib isimhost.lib for “Release”
   add isimcoredbg.lib isimhostdbg.lib for “Debug”
5. Copy main.cpp from tools\isim\isimexe into the project

2.3 File Structure

Imagine applications consist of the files described in Table 2. The file main.cpp is always the same. The arbitrary C++ files that do not involve KernelC or StreamC have no restrictions. The StreamC and KernelC files all have specific formats, including a REQUIRED ordering of header files, described in the remainder of this section. The simulator script files are introduced in Section 6.1 and described fully in Section 8.0.

TABLE 2. File Types

File	Description
main.cpp	The standard main file included in the toolset
*_kc.cpp	A KernelC file containing a kernel
*_sc.cpp	A StreamC file containing a stream program, or functions that contain StreamC
*_kc.hpp	A shared header file containing kernel declarations and record definitions shared by both KernelC and StreamC files
*.hpp, *.cpp	Arbitrary C++ files that contain portions of the application that do not involve KernelC or StreamC
*.sim	simulator script files used to execute the application on a simulator

The Imagine tools also use/produce several related file types, the most important of which are summarized in the following table:.

TABLE 3. Related File Types

File	Description
*.md	Machine description file, describes the Imagine architecture
*.uc	Human readable microcode format, output of kernel scheduler
.raw/.lis	Binary microcode formats used to verify decode logic and for actual hardware (.lis is a special binary format for use with verilog)
*.pro	Profile recorded for profiling compiler
*.viz	Schedule visualizer file, used to visualize an iscd schedule, istream resource allocation, or isim run

2.3.1 Shared header file format (*_kc.hpp)

These files contain record definitions and kernel declarations. Records are essentially structures defined using Imagine types. Kernel declarations are essentially function declarations for kernels. Each kernel declaration is followed by two special lines containing the KERNELDECL and KERNELCALL macros. The file must end by including the "idb_undeftypes.hpp" header file.

The file format is as follows:

```
#include "idb_types.hpp"
#include "idb_deftypes.hpp”

record recordName
{
    type  fieldName;
    ...
};

kernel kernelName(type parameterName, ...);
KERNELDECL(kernelName);
#define kernelName KERNELCALL(kernelName);

#include "idb_undeftypes.hpp"
```

2.3.2 KernelC file format (*_kc.cpp)

These files contain one or more kernels (one kernel per file is recommended). The file must begin with the order of headers shown. Each kernel must be preceded by a KERNELDEF macro that takes the name of the kernel and the file name (and path) of the .uc file produced by the kernel scheduler as arguments.

```
#include “idb_kernelc.hpp”
#include “sharedHeaderFileName”;
...
#include “idb_kernelc2.hpp”

KERNELDEF(kernelName, “kernelUCFileName”);
kernel kernelName(type parameterName, ...)
{
    // kernelC code here
    ...
}
```

2.3.3 StreamC file format (*_sc.cpp)

These files contain a mix of C++ and StreamC. Each file must begin with the order of headers shown. At least one of these files must contain a special function called a stream program. A stream program is somewhat like the “main” function of a stream application, though each stream program has a unique name and there may be more than one. A stream program can have only two arguments, the special argument STREAM_SCHEDULER, and a String called args. It must be preceded by the STREAMPROG macro with the name of the stream program as an argument. Any function that uses StreamC must have a STREAM_SCHEDULER argument, and be passed the special variable “scd” as shown.

```
#include “idb_streamc.hpp”
#include “sharedHeaderFileName”;
...

STREAMPROG(streamProgramName);

void streamProgramName(STREAM_SCHEDULER, String args)
{
    // C++ or StreamC code here
    ...
    otherFunctionName(scd, ...);
}
void otherFunctionName (STREAM_SCHEDULER, ...)
{
    // more C++ or StreamC code here
    ...
}
```

2.4 Exercise, Step 1

This is the first step in creating a simple Imagine application. Do the following:

1. Ready the toolset for use a described in Section 2.1.
2. Start Visual C++ and create a “Win32 Console Application” project called “test” in a directory within “im_apps” called “test” as described in Section 2.2.
3. Add the file main.cpp to the project.
4. Add the new (blank) files test_kc.cpp, test_sc.cpp, and test_kc.hpp

3.0 KernelC Language Specification

The KernelC language is to be used for programming kernels to be run on the arithmetic clusters in Imagine. The language uses a C-like expression syntax that can be handled by the Imagine kernel scheduler, which includes enough expressive power that programmers do not need to manage the hardware resources.

3.1 Types

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

3.1.1 Basic Types

Table 4 lists the currently supported basic types.

TABLE 4. 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 3.6, integer operators are assumed to operate on both signed and unsigned types unless otherwise specified.

3.1.2 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;
  ...
};

Where *type* can be any basic type or previously defined record type. 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.

3.1.3 Type Qualifiers

Type qualifiers modify a type for a special purpose. The type qualifier syntax is:

type_qualifier <*type*>

Table 5 lists the type qualifiers' syntax and their meanings.

TABLE 5. 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 (allocated in the scratchpad)
EXPAND	Similar to Array (allocated in the LRF)

Type qualifiers cannot be combined, applied to the basic type CC, or applied to record fields. Only the stream (istream, cistream, ostream, costream) qualifiers can be applied to record types.

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 can only be read or written in a kernel using a communication operation (e.g. commclperm described in Equation 3.6.33).

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 and the EXPAND qualifier are used to specify an array of elements of the base type. The only difference is that while the former gets allocated in the scratchpad, the later is allocated in the local register files. The size of the array or expand is specified in parentheses after the name of the array, for example:

ARRAY<INT> foo(10);

EXPAND<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 declaring the same persistent arrays in the same order in multiple kernels, this qualifier allows multiple kernels to maintain persistent data in the scratchpad, such that one kernel can write some data to the array, and the next kernel can read/and or change that data, and so on.

3.1.4 Preprocessor Directives

Standard C preprocessor directive are supported.

3.1.5 Comments

KernelC 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.

3.2 Kernels

A Kernel is essentially a function that operates on streams. Kernels are defined using KernelC. 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 some initial and final code surrounding a central loop which iterates over the input streams and produces the output streams.

3.3 Variable Declarations

Variable declaration syntax is as follows:

type name [= *initial value*];

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

3.3.1 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 = 0x01020304;
```

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;
```

3.4 Control Flow

All KernelC 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:

3.4.1 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* }

3.4.2 Stream Loops

In this form of looping, the loop continues until 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* }

3.4.3 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.

3.4.4 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 to modulo software pipeline any non-nested loop. Software pipelining is a technique in which a loop is divided into *n* stages and different stages of *n* iterations are executed at once. Modulo software pipelining is an algorithm for producing a software pipelined loop that relies on determining the shortest possible schedule length for the loop, called the minimum iteration interval or *minII*, then attempting to construct a software pipelined loop with that length. If the attempt fails, an attempt is made to construct a loop of length *minII + 1*, and so on until a valid iteration interval (loop length) is found.

To use modulo software pipelining, the hint PIPELINE(*startII*) is placed just before the open brace. If *startII* is 1 or less than the minimum iteration interval, the search for a valid iteration interval begins with the minimum iteration interval, otherwise it begins with *startII*.

This form of pipelining is for kernels and should not be confused with the software pipelining described in Section 9.5, which is for stream programs and completely separate.

3.5 Inline Functions and Methods

KernelC and the Imagine stream processor do not support kernel function calls. However, for the convenience of programmers, it is useful to use function semantics in defining kernels. Functions are cleaner than macros in that they have scope, are more modular, and are easier to debug. Functions allow better analysis of the impact of new functions to the Imagine instruction set. Record member functions also have the advantage of transparent access to record members. Like macros, these functions are all inlined when invoked, but still maintain function semantics. Repeating that, since it's the most important idea here: KernelC functions, as defined in this document, are inlined when invoked.

We support two kinds of functions:

- Record member functions, with semantics identical to a class/structure member function in C++;

- Global standalone functions, with semantics identical to a regular function definition and call in C++.

Functions can take an arbitrary number of arguments. These arguments can be either Imagine native data types like float and int, or they can be records. Arguments can be passed by value or by reference. Return values from functions can also be Imagine native datatypes or records.

3.5.1 Examples

```
record xyz {
  float x, y, z;
  float norm() {
    return fsqrt(x*x + y*y + z*z);
  }
```

The "norm" function is a member of the xyz class. It is invoked as

```
xyz v;
float f = v.norm();
```

Functions can return variables or the results of operations.

```
void scale(float s) {
  x = x * s;  y = y * s;  z = z * s;
}
void normalize() {
  scale(1.0 / norm());
}
```

which can be invoked as

```
v.normalize();
```

As in normalize() above, functions can call other functions which have already been defined.

Prototypes are permitted for member functions and are necessary in cases such as the one below:

```
record xyz {
  float x, y, z;
  void AddToMe(xyz v2);
};

inline void xyz::AddToMe(xyz v2) {
  x = v2.x + x;
  y = v2.y + y;
  z = v2.z + z;
}
```

which is invoked as

```
xyz va, vb;
va.AddToMe(vb);
```

The reason prototypes are necessary in this case is because AddToMe uses a datatype (xyz) that is not yet fully defined at the time the prototype is needed.

Global standalone functions (not members of any record) are also permitted:

```
inline void ClampBetween(float & f, float clamp_lo, float clamp_hi) {
  f = select(itocc(f <= clamp_hi),
             select(itocc(f >= clamp_lo), f, clamp_lo),
             clamp_hi);
}

inline float ClampBetween2(float f, float clamp_lo, float clamp_hi) {
  return select(itocc(f <= clamp_hi),
                select(itocc(f >= clamp_lo), f, clamp_lo),
                clamp_hi);
}
```

which is invoked as

```
  float f, f2;
  float flo = 0.0;
  float fhi = 1.0;
  ClampBetween(f, flo, fhi);             // f is passed by reference
  f2 = ClampBetween2(f2, flo, fhi);      // f2 is passed by value
```

You are also permitted to prototype global standalone functions. This is not necessary in KernelC, but is necessary for idebug, because idebug requires the functions to be defined once per executable, but KernelC needs them to be defined in each kernel. To gracefully handle this difficulty, put the prototypes in the headers, link the corresponding .cpp files into the debug environment project, and for KernelC only, conditionally include the .cpp files into each kernel (probably easiest by including them at the bottom of the headers).

```
/* at the end of standalone_functions.hpp (for instance): */
#ifdef _IMAGINE_BUILD
#include "standalone_functions.cpp"
#endif
```

Note that both the global standalone and externally defined member functions have the "inline" keyword. This keyword is not mandatory but it is intended to remind the programmer that these functions will be directly inlined and are not real function calls.

3.5.2 Limitations

The space of supported functions is not nearly as great as C++. It is important to keep in mind that the KernelC parser is single pass, which reduces the breadth of capabilities of KernelC functions.

In particular:

- Function support has not been tested as extensively as other KernelC features.

- Circular function calls (where a() calls b() and b() calls a()) are not possible. In fact, circular record usage where something defined as part of record x uses record y and

something defined as part of record y uses record x is also not possible. The specific restriction in KernelC is that things must be defined before they're used, whereas in C++ it is often sufficient to only declare things before they're used. (In other words, this isn't allowed:

```
inline int foo();
inline int bar() { return foo(); }
```

instead you'd have to define foo first, not just prototype it.)

- Recursion is not permitted.

- There is no concept of constructors (or destructors).

- Function argument testing between prototype and definition is nonexistent. Return value checking is also primitive, so avoid prototyping a function with one set of arguments then defining it with another. (In this case, the definition set will be accepted and the prototype set will be thrown away.)

- Streams are not permitted as function arguments.

- There's no capability for overloading a function name: i.e. you can't have both foo(int a) and foo(float b) and expect them to be properly resolved.

Most of the capabilities of functions can be achieved by judicious use of macros, so if your use of functions causes problems, falling back to macros is the preferred solution.

3.6 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.

3.6.1 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.

3.6.2 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.

3.6.3 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.

3.6.4 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.

3.6.5 ABS

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

Description. ABS computes the absolute value of x.

3.6.6 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.

3.6.7 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.

3.6.8 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.

3.6.9 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 separately, using the multiple bits of the CC p.

3.6.10 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.

3.6.11 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.

3.6.12 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.

3.6.13 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.

3.6.14 FSQRT

Format	fsqrt(x)
Input Types	FLOAT
Output Type	FLOAT

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

3.6.15 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.

3.6.16 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.

3.6.17 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. Signed integers are treated the same as unsigned integers; the

sign bit is rotated normally. 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.

3.6.18 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.

3.6.19 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.

3.6.20 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.

3.6.21 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)).

3.6.22 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.

3.6.23 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.

3.6.24 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.

3.6.25 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.

3.6.26 HI

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

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

3.6.27 LO

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

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

3.6.28 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.

3.6.29 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.

3.6.30 CHECK_OVF

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

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

3.6.31 CHECK_UNF

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

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

3.6.32 COMMUCPERM

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

Description. COMMUCPERM performs intercluster communication as determined by *perm*, which must be a uc variable. *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 value of ‘x’ will be stored in ‘y’.

Examples:

```
// swap values in adjacent clusters
// cluster 0 gets value in cluster 1,
// cluster 1 gets value in cluster 0, etc.
uc<int> perm = 0x67452301;
x = commucperm(perm, x);

// foo in all clusters and the uc variable uc_foo get the value of
// bar in cluster 3
uc<int> perm = 0x33333333;
foo = commucperm(perm, bar, 3, uc_foo);
```

3.6.33 COMMCLPERM

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

Description. COMMCLPERM performs intercluster communication as determined by *perm*, which cannot be a uc variable. *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 clus-

ter. Each source index is a number between 0 and 8, which corresponds to the source cluster -- 8 indicates the microcontroller, or input ‘y’. If specified, ‘srcidx’ specifies which cluster’s data will be stored in ‘z’. The value of ‘srcidx’ must indicate a cluster, it cannot indicate the microcontroller (it cannot be 8).

The first two forms of commclperm duplicate the two forms of commucperm shown above except that a uc variable is not used for the permutation. The remaining two are illustrated by these examples:

```
// assign value of uc variable uc_bar to variable foo in all clusters
foo = commclperm(8, x, uc_bar);

// as above, and assign value of x in cluster 1 to uc_foo
foo = commclperm(8, x, uc_bar, 1, uc_foo);
```

3.6.34 INPUT

Format	istr >> x;
Input Types	istr: ISTREAM<TYPE> 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.

3.6.35 OUTPUT

Format	ostr << x;
Input Types	ostr: OSTREAM<TYPE> 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.

3.6.36 CONDINPUT

Format	cistr(p, ccend) >> x;
Input Types	cistr: CISTREAM<TYPE> p: CC ccend: CC
Output Type	x: 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, it is true only if data was requested but none was available. The “>>” operator can be cascaded to input multiple values, similar to the C++ operator>> semantics, all of which use the same predicate.

3.6.37 CONDOUTPUT

Format	costr(p) << x;
Input Types	costr: COSTREAM<TYPE> p: CC 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, all of which use the same predicate.

3.6.38 CONDALLINPUT

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

Description. CONDALLINPUT functions like CONDINPUT except that the value of p must be the same in all clusters; either all or no records are read from the input stream. For this reason, it can and must be used with a normal, non-conditional input stream. Thus, it does not have a ccend argument, but can be used in a loop_stream. Note: in the above syntax, ALL is a literal keyword not an argument.

3.6.39 CONDALLOUTPUT

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

Description. CONDALLOUTPUT functions like CONDOUTPUT except that the value of p must be the same in all clusters; either all or no records are written to the output stream. For this reason, it can and must be used with a normal, non-conditional output stream. Note: in the above syntax, ALL is a literal keyword not an argument.

3.6.40 FLUSH

Format	flush(costr, x);
Input Types	costr: COSTREAM<TYPE> 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.

3.6.41 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. For example, in an eight cluster machine, *cid*() would return zero in cluster zero, one in cluster one, ..., and seven in cluster seven.

3.6.42 UCID

Format	ucid();
Input Types	none
Output Type	INT

Description. The special function *ucid()* can be used to obtain the microcontroller’s index, which is normally equal to the number of clusters. This function will return a

number that can be used in communication permutations to address the microcontroller. For example, in an eight cluster machine, *ucid*() would return eight in all clusters.

3.6.43 SYNCH

Format	synch();
Input Types	none
Output Type	NONE

Description. The *synch* operation is never required, but it can be used enable IStream to reorder a kernel that uses uc parameters, which is otherwise impossible. In order for a kernel to be reordered it needs to have the following:

- synch() if it reads any UC arguments
- read UC arguments, if any
- write UC arguments, if any
- synch() if it writes any UC arguments

Other code can be arranged around these steps in any order.

3.7 Exercise, Step 2

1. Open the file “test_kc.cpp” and add the following code:

```
// headers must always be in this order
#include "idb_kernelc.hpp"
#include "test_kc.hpp"
#include "idb_kernelc2.hpp"

// this tells StreamC where to find microcode
KERNELDEF(addAndSum, "test/test_kc.uc");

// this kernel takes two stream of integers
// adds each pair of integers and
// sums all of the integers
// it has the following arguments:
//   a, input stream of integers
//   b, input stream of integers
//   c, output stream of addition results
//   uc_sum, the sum of all integers
kernel addAndSum(istream<int> a,
                 istream<int> b,
                 ostream<int> c,
                 uc<int>& uc_sum)
{
  // read initial value of uc_sum into sum
  // dummy is an unused argument
  int sum;
  int dummy;
  sum = commclperm(ucid(), dummy, uc_sum);
  // only keep sum in cluster 0 (otherwise, would be x8)
  sum = select(itocc(cid() == 0), sum, 0);

  uc<int> perm_a = 0x67452301;
  uc<int> perm_b = 0x44660022;
  uc<int> perm_c = 0x00004444;

  // loop over input stream a
  loop_stream(a) {
    int a1, b1, c1;
    // read a value from a and b
    a >> a1;
    b >> b1;
    // add the values
    c1 = a1 + b1;
    // output the result
    c << c1;
    // update sum
    sum = sum + c1;
  }

  // compute sum across all clusters
  // adjacent clusters...
  sum = sum + commucperm(perm_a, sum);
  // adjacent pairs of clusters...
  sum = sum + commucperm(perm_b, sum);
  // adjacent "quads" of clusters ...
  sum = sum + commucperm(perm_c, sum);

  // write the value of sum in cluster 0 into uc_sum
```

```
  // dummy is an unused argument and result
  dummy = commclperm(dummy, sum, 0, uc_sum);
}
```

2. Open the file “test_kc.hpp” and add the following code:

```
#ifndef _TEST_KC
#define _TEST_KC

// header files must be in this order
#include "idb_types.hpp"
#include "idb_deftypes.hpp"

// declare the kernel
kernel addAndSum(istream<int> a,
                 istream<int> b,
                 ostream<int> c,
                 uc<int>& uc_sum);
// these are required for each kernel
KERNELDECL(addAndSum);
#define addAndSum KERNELCALL(addAndSum)

// don’t forget this line at the end!
#include "idb_undeftypes.hpp"

#endif
```

4.0 StreamC Language Specification

The StreamC language is used in combination with C++ for writing stream programs. It includes definitions of stream and microcontoller variables, calling kernels, copying streams, loading and saving data to/from streams, and transferring streams over a network. StreamC also includes special annotations required to use IStream, the profiling compiler described in Section 9.0.

4.1 Imagine Basic Types in StreamC

Imagine basic types should be used in StreamC only to declare variables of stream types and microcontroller (UC) qualified types. To avoid name conflicts with normal C++ basic types, Imagine basic types are distinguished by an “im_” prefix in StreamC. For instance, the Imagine “int” basic type is “im_int”. Note: this prefix is not required in shared header files as defined in Section 2.3.1.

4.2 Streams

Stream variables are like pointers that refer to a sequence of data records. Stream variables are declared as follows:

im_stream <*type*> *name*;

where type is an imagine basic type or user-defined record.

4.2.1 NAMED Streams

The NAMED macro, used in a stream declaration as shown below, exposes the name of a stream variable to the application enabling it to be used in ISim messages, error messages, etc. It is recommended that all streams be “NAMED.”

im_stream <*type*> NAMED(*name*) = ...

4.2.2 Null Streams

Stream variables are initially null -- they do not refer to any data. Like a null pointer, any attempt to use a null stream results in an error. The following method returns true if a stream is not null:

```
x.isValid();
```

4.2.3 newStreamData

A stream variable can be assigned to refer to new data as follows:

x = newStreamData<*type*> (*size, [optional] data dependence* = im_fixed);

The newStreamData functional allocates storage for *size* records. Use of the *data dependence* parameter is described in Section 4.9.2.

4.2.4 Simple Assignments

A stream variable can be assigned to refer to the data referred to by another stream variable using standard assignment syntax as shown below. It is important to note that this does not copy any data, it merely makes both streams refer (or "point") to the same data.

```
y = x;
```

4.2.5 Derivations

A stream variable can be assigned to refer to a subset of the data referred to by another stream variable. Such a stream is said to be derived from the other stream.

The simplest derivation refers to a range. The following syntax is used to assign y to refer to every record in x starting with *start index* up to but not including *end index*:

y = x(*start index*, *end index*, *[optional] data dependence* = im_fixed);

For instance:

```
y = x(0, 8);
```

assigns y to refer to records 0 through 7 of x.

More complex derivations can be achieved using one of three access types:

Strided access can be used to refer to every *stride*th record in the range, as follows:

y = x(*start*, *end*, *variableLength*, im_acc_stride, *stride*);

For instance:

```
y = x(0, 8, im_fixed, im_acc_stride, 2);
```

assigns y to refer to records 0, 2, 4, and 6 of x.

Bit-reversed access refers to every record in the range, but with the bits in the index rearranged using four steps:

1. divide by modulus
2. reverse order of first n bits in quotient, where n = log2(length of stream)
3. multiply bit-reversed quotient by modulus
4. add remainder of original division

Bit-reversed access is specified as follows:

y = x(*start*, *end*, *variableLength*, im_acc_bit_reverse, *modulus*);

For instance:

```
y = x(0, 7, im_fixed, im_acc_bit_reverse, 2);
```

assigns y to refer to records 0, 1, 4, 5, 2, 3, 6, and 7 of x (in that order).

Indexed access refers to elements constrained within the range with indices given by the elements of a stream of unsigned integers, as follows:

y = x(*start*, *end*, im_fixed, im_acc_index, *index*);

For instance, if the im_stream<im_int> z refers to integers with values 5, 6, 5, 4, 2 then:

```
y = x(0, 7, im_fixed, im_acc_index, z);
```

assigns y to refer to elements (words) 5, 6, 5, 4, and 2 of x (in that order). Note that the indices refer to the words and not the records in x, thus allowing indexing inside records.

4.2.6 Overriding Record Size and Specifying Coordinates in Words

All such assignments also accept two additional, optional parameters: *recordSize* and *coordinatesInWord*s.

y = x(..., *[optional] recordSize* = 1, *[optional] coordinatesInWords* = false);

recordSize (specified in words) can be used to override the default record size so that that multiple records are treated as a single record when distributing the data to the clusters. Specifying a *recordSize* of 3 would result in cluster 0 of 8 clusters receiving records 0, 1, 2, then 24, 25, 26, while cluster 1 received records 3, 4, 5, then 27, 28, 29 and so on.

If *coordinatesInWords* is true, then the range start, end, stride, bit-reverse modulus, and record size are specified in words instead of in records (the default). This could be used to derive a stream referring to one field in each record of a stream. For instance, given a stream of 256, ten-word Foo records, a stream containing the third field could be extracted as follows:

```
// start at word 3,
// end at word 256 * 10 = 2560,
// stride of 10 words,
// record length of 1 word
a = b(3, 2560, im_acc_stride, 10, 1, true);
```

4.2.7 Countup Streams

The length of some streams may vary depending on the data being processed. When allocating space for such a stream or deriving it from another stream, the data dependence parameter should be set to im_countup.

x = newStreamData<*type*> (*size*, im_countup);

or

```
y = x (0, 8, im_countup);
```

A countup stream initially contains zero records. When written to by a kernel or other operation, the length of the stream is set to the number of records written. The maximum length is the allocated *size* specified in the newStreamData call. The current length of a countup stream can be determined as follows:

```
x.getLength();
```

An additional function, getSize() returns the maximum number of records the stream could contain.

4.2.8 Stream Derivation Restrictions

Certain hardware specific restrictions apply to these derivations. In particular, almost all derivations must be from sequential streams. A sequential stream is a stream with strided access and a stride equal to the record length. Streams with strided access and a stride other than the record length, or a bit-reversed or indexed access pattern, or coordinates in words that are not evenly divisible by record length can only be derived from a sequential stream.

4.3 Microcontroller Variables

Microcontoller variables may be declared in StreamC as follows:

im_uc<*type*> name;

where type is an Imagine basic type (with the “im_” prefix).

Microcontoller variables may be read and written as follows:

```
int i;
im_uc<im_int> uc_i;
uc_i = i;// assign value of i  to uc_i
i = ucRead(uc_i);// assign value of uc_i to i
```

Microcontroller variables are only useful in StreamC because they allow single values to be passed as kernel arguments, for example if the kernel “sum” computes the sum of a stream of integers, it could be called as follows:

```
im_stream<im_int> x = ...;
...
im_uc<im_int> sumOfX = 0;
sum(x, sumOfX);
cout << ucRead(sumOfX);
```

4.4 Kernels

A Kernel is essentially a function that takes streams and microcontroller variables as arguments. See the KernelC documentation for information on writing kernels. Kernels are called in StreamC just like any other function. A kernel is called as follows:

kernel(*argument0, ..., argument n*);

4.4.1 Kernel Stream Restarts

Kernels can be made to read inputs from or write outputs to a series of streams by restarting the kernel with each additional stream as follows:

kernel(... , **restart**(*stream argument i*), ...);
kernelRestart(*i*, **restart**(*another stream*));
...
kernelRestart(*i*, *final stream*);

In this syntax, *restart* indicates that a kernel argument will be restarted (any number and combination of arguments may be restarted), and each *kernelRestart* specifies the stream used to restart one argument and actually triggers the restart of that argument. A series of restarted streams will be treated exactly the same as one large input stream by the kernel, for example a loop_stream would loop over the records in all of the streams. The kernel will not terminate until the end of the last stream and all internal state will stay the same between streams.

For example, a kernel might require ten passes over an input stream, which could be written as:

```
foo(in1, ...);
for (int i = 0; i < 9; i++) {
  kernelRestart(0, in1);
}
```

4.5 Copying Streams

StreamC includes a special function that copies records from one stream to another.

void **streamCopy**(im_istream<T1> in1, im_ostream<T2> out1);

It is important to note that a derived stream refers to part of another stream, whereas streamCopy makes a copy of the records in the stream. For instance, the following two statements are not equivalent:

```
// refer to part of A
b = a(0, 100);

// copy part of A
streamCopy(a(0, 100), b);
```

Following the first statement, a change to the records in a *a* will also change the records in *b*. Following the second statement, *b* will still contain a copy of the old records even if the records in *a* are changed.

4.6 Loading/Saving Streams

StreamC includes several functions to load data to streams from binary pointers, a vector template class, or a file (see Section 6.4 for data file formats).

void **streamLoadBin**(void* bin, int length, im_ostream<T> out1);

void **streamLoadVect**(vector<Tv>& vect, im_ostream<T> out1);

void **streamLoadFile**(char* file, String type, String args, im_ostream<T> out1);

StreamC also includes an additional load function that replicates the records in a file number-of-clusters times so that each cluster receives one copy of each record, which is useful for loading streams of constants.

void **streamLoadFileReplicated**(char *file, String type, String args, im_ostream<T> out1);

Conversely, streamC includes several functions to save data from streams:

void **streamSaveBin**(void* bin, im_istream<T> in1);

void **streamSaveVect**(vector<Tv>& vect, im_istream<T> in1);

void **streamSaveFile**(char *file, String type, String args, im_istream<T> in1);

Lastly, StreamC includes a special function that compares data from a stream to that saved in a file:

bool **streamCompareFile**(char *file, im_istream<T> in1, float threshold, String args);

where args are typically “a” for absolute comparison (difference is measured by subtracting one value from the other) or “r” for relative comparison (difference is measured by subtracting one value from the other then dividing by their sum), and threshold is the maximum absolute or relative difference.

4.7 Network Operations

StreamC includes network operations to transfer data between processors using the following functions:

void **streamSend**(StreamRoute route, im_istream<T> in1);

void **streamReceive**(StreamRoute route, im_istream<T> in1);

4.7.1 Stream Routes

A StreamRoute variable defines a route between two imagine processors, using the syntax shown below. All StreamRoute variables must be declared as global variables (outside of a function).

StreamRoute *name*(int source, int dest, char* route);

where *source* and *dest* are the indices of the source and destination processors and *route* is a string of in which each digit indicates a hop according to the following table:

TABLE 6.

Digit	First hop direction	Subsequent hop direction
0	north	eject
1	south	right
2	west	left
3	east	straight
4	loopback	--

4.7.2 Network Stream Restarts

Network operations may be restarted the manner shown below:

streamSend or streamReceive(*route*, **restart**(*stream*));
streamSend or streamReceive(*route*, **restart**(*another stream*));
...
streamSend or streamReceive(*route*, *final stream*);

Note that all restarts for a particular transfer must use the same StreamRoute. Network transfers intended to occur in parallel should not use the same StreamRoute.

For example, two streams produced by different kernels could be sent over the network and received as one stream using the following, which could be useful for accumulating the results without tying up SRF space on the sending processor:

```
// on sending processor
networkSend(r1, restart(a));
...
networkSend(r1, b);

// on receiving processor
networkReceive(r1, ab);
```

4.8 Controlling Multiple Imagines with One Stream Program

A single stream program can be used to control multiple Imagines with the same host processor. Such a stream program must be preceded by a STREAMPROG_MULTIPLE statement instead of a STREAMPROG statement. STREAMPROG_MULTIPLE takes two arguments, the name of the function that implements the stream program and the number of Imagines it controls, for instance:

STREAMPROG_MULTIPLE(testProgram, 2)

In such a stream program, all profile and data-dependent control-flow blocks apply to all Imagines controlled by the stream program, but all StreamC operations (e.g. kernels) are executed only on the current Imagine. The current Imagine is initially that with index 0, and can be set using the following syntax:

setImagine(*index of new current Imagine*);

The current Imagine is set back to 0 upon entering or leaving a profiled section of code.

For good performance, operations sent to the two Imagines should alternate unless one takes significantly more time than the other, and calls to getLength and ucRead should occur as late as possible. For any network transfers between Imagines controlled by the same stream program, the streamSend must precede the streamReceive.

For a good example of a stream program that uses multiple imagines, see the im_apps\sortmulti_sc application (not reprinted here for brevity).

4.9 Profiling Annotations

IStream, the tool used to compile StreamC, is a profile compiler. Rather than compile the source code directly, it compiles a *profile* of the program made during a run using the functional simulator or naive compilation. This profile is essentially a recording of the sequence of stream operations performed by the program. If the sequence of stream operations (or any of their arguments) varies depending on the data being processed, this variation must be communicated to the profile compiler using special annotations. These annotations fall into two main categories, *data dependence* annotations to streams, which are used to note that some aspect of a stream such as its start or end varies depending on the data being processed, and *data dependent control flow*, if-statements or loops that depend on the data being processed. These annotations are described in the following sections. These annotations are not required in order to use IDebug, so initial development can be done without them, but they are required for use of IStream (and hence ISim).

4.9.1 Designating which part of the application to profile

IStream only profiles a section of the application enclosed as follows:

```
profile(profile name) {
  ...
}
```

where IStream appends various suffixes to the *profile name* (which must be the start of a valid file name, including the path, if any) to produce file names for data, visualization, and information files related to the profile (see Section 9.0 for more detail on those files).

It is possible to produce multiple profiles for a stream program under varying parameters (e.g. different profiles for an image-processing program applied to different image sizes) by simply giving each such profile a different name.

4.9.2 Data-dependent stream derivations

The *data dependence* argument in a newStreamData call or stream derivation is used to mark countup streams, or stream derivations where the start index or end index depend on the data being processed (within a profile, these are the only parts of stream derivation that are allowed to vary). The data dependence argument can be some combination of the following flags OR'ed together:

TABLE 7.

Flag	Meaning
im_fixed	(default), stream is not data dependent, cannot be combined with other flags
im_countup	is a countup stream, see Section 4.2.7
im_var_size	start index is zero, end index is data-dependent, no data past end
im_var_pos	start and end index are data-dependent
im_var_align	as im_var_pos, in addition start index times record size is always evenly divisible by 32
im_var_incr	as im_var_pos, in addition successive accesses will always access incrementing, disjoint portions of a larger stream
Stripmining flags:	
im_strip_none	These flags are binary OR'ed into the data-dependence parameter to provide extra information about the stripmining behavior of the stream. For details on their usage, see Section 9.4.
im_strip_ignore	
Special hardware flags:	
im_cacheable	Used for hardware experiments, see Section 11.4

The following examples illustrate the use of these flags:

im_fixed: A fixed stream is used whenever the start and end are always the same e.g.:

```
im_stream<Foo> out_data = newStreamData<Foo>(100);
im_stream<Foo> out_firstHalfEven = out_data(0, 50, im_fixed,
im_acc_stride, 2);
myKernel(in, out_firstHalfEven);
```

im_countup: A countup, also called variable length, stream is used whenever the end varies depending the number of records produced by a stream operation. Note: all countup streams are also inherently variable size.

```
im_stream<Foo> out = newStreamData<Foo>(maxFoo, im_countup);
myKernel(in, out);
recordsInOut = out.getLength();
```

im_var_size: A variable size stream is used whenever the end varies, most often to contain the output of kernel that consumes a stream with a data-dependent number of records and produces the same number of records, e.g.:

```
im_stream<Foo> out_data = newStreamData<Foo>(maxFoo);
im_stream<Foo> out = out_data(0, in.getLength(), im_var_size);
myKernel(in, out);
```

IStream assumes that there is no valid data past the end of such a stream.

im_var_pos, im_var_align, im_var_incr: A variable position stream is used whenever the start or end varies, most often to iterate over parts of a stream or do some sort of lookup into a particular stream. It is important to note that im_var_pos does not place any restrictions on the size of the stream since both the start and end can vary. In case where the stream size is bounded, copying the stream into a fixed size stream is recommended. For instance:

```
im_stream<Foo> out = newStreamData<Foo>(maxFoo);
for_VARIABLE(i = 0; i < in.getLength(); i+= 32) {
   im_stream<Foo> outIter = newStreamData<Foo>(32, im_countup);
   streamCopy(out(i, max(i + 32, in.getLength()),
        im_var_pos | im_var_align | im_var_incr), outIter);
   myKernel(in, outIter);
}
```

In the case of this example, the im_var_align flag can be used because the start is always divisible by 32. This flag is important for good performance because it allows multiple variable position accesses to use the same buffer in the SRF. Further, the im_var_incr flag can be used because the definition always refers to incrementing, disjoint sections of a larger stream. This flag is important because it avoids dependencies between successive accesses.

4.9.3 Data-dependent control flow

Data-dependent control flow occurs any time a branch depends on the data being processed. Only data-dependent if-statements and loops are supported. If a loop iterates a large number of times (more than ten), it is recommended that it be marked as a data-dependent loop even if it is not data-dependent since this will reduce the size of the profile and make compiling it significantly faster. Data-dependent if-statements within a profile must be annotated as follows (else is not supported):

```
if_VARIABLE(...) {
   ...
}
```

Data-dependent loops must be annotated as follows:

```
while_VARIABLE ( ... ) {
    loopIter();
   ...
}
```

or

```
for_VARIABLE ( ... ) {
    loopIter();
   ...
}
```

The loopIter call has two optional parameters that can be used can be used to handle data-dependent loops that differ from iteration to iteration

```
loopIter([optional] preOrPost = false, [optional] newPeriod = true);
```

preOrPost can be set to true for one or more iterations at the start and/or end of the loop to “peel” those iterations off of the loop (for instance, if they execute special operations). newPeriod can be set to true only every *n* iterations to unroll the loop *n* times (for instance, if the loop toggles between two sets of streams). For, example, the following code executes a loop with the first and last two iterations peeled, unrolled three times, to implement three rotating row buffers with padding:

```
int peeledStart = 2;
int peeledEnd = 2;
int unrolledTimes = 3;

for_VARIABLE (int i = 0; i < numLoops; i++) {
    loopIter(i < peeledStart || i >= numLoops - peeledEnd,
       ((i - peeledStart) % unrolledTimes) == 0);
   temp = buffer2;
   buffer2 = buffer1;
   buffer1 = buffer0;
   buffer0 = temp;
   if (i < 2 || i >= numLoops - 2) {
     producePadding(buffer0);
   } else {
     streamCopy(inImage(width * (i - 2), width * (i - 1), im_var_pos),
       buffer0);
   }
   if (i >= 2) {
     processData(buffer0, buffer1, buffer2, output);
     streamCopy(output, outImage(width * (i - 2), width * (i - 1),
       im_var_pos);
   }
}
```

4.9.4 Restrictions on newStreamData

Within a profile the size argument to newStreamData must be a constant, and one newStreamData call cannot be used to allocate multiple chunks of streamData used at the same time. For example, the following is not legal within a profile:

```
im_stream<Foo> y = newStreamData(x.getLength());
im_stream<Foo> z[10];
for_VARIABLE(int i = 0; ...) {
    z[i] = newStreamData(100);
    ...
}
```

In addition, a stream that refers to data allocated using newStreamData within a profile may not be assigned to a stream variable from outside the profile. For example, the following is not legal:

```
im_stream<Foo> a;
profile(...) {
    a = newStreamData<Foo>(100);
    ...
}
```

4.10 Exercise, Step 3

1. Open the file “test_sc.cpp” and add the following code:

```
#include "idb_streamc.hpp"
#include "test_kc.hpp"

// this defines the function "testProgram"
// as a stream program
STREAMPROG(testProgram);

// this stream program is a simple example
// that calls the addAndSum kernel twice
// all stream programs must have only these arguments
void testProgram(STREAM_SCHEDULER, String args)
{
  // print the arguments (otherwise unused)
  cout << args << endl;

  // declare two streams of 32 integers
  // note that these are streams of "im_int"
  // the Imagine integer type not "int"
  im_stream<im_int> NAMED(s1) =
    newStreamData<im_int>(32);
  im_stream<im_int> NAMED(s2) =
    newStreamData<im_int>(32);

  // initialize the input stream from an array
  int data[32];
  for (int i = 0; i < 32; i++) {
      data[i] = i;
  }
  streamLoadBin(data, s1);

  // declare a microcontroller variable
  im_uc<im_int> uc_sum = 0;

  // profile this portion of the application
  profile("testProgram") {

    // declare a stream of 32 integers
    im_stream<im_int> NAMED(temp) =
    newStreamData<im_int>(32);

    // call the addAndSum kernel twice
    addAndSum(s1, s1,   temp,   uc_sum);
    addAndSum(s1, temp, s2,     uc_sum);
  }

  // save the output and display it
  streamSaveBin(data, s2);
  for (int i = 0; i < 32; i++) {
    cout << data[i] << " ";
  }
  cout << endl;
  cout << "sum = " << ucRead(uc_sum) << endl;
}
```

5.0 IScd, the Kernel Scheduler

This section describes how to use IScd to compile kernels for execution on the Imagine processor. IScd must be used to compile all kernels before using the cycle-accurate simulator, and in order to use the functional simulator to create a profile. However, it is recommended that the application first be debugged using the functional simulator in non-profiling mode as described in Section 7.1.

5.1 Optimizations

There are two key optimizations that should be used in the scheduling process. First, almost all loops should be software pipelined as described in Section 3.4.4. Many of the terms introduced in that section are used in this section. Second, Scheduling can be repeated multiple times using different random seeds to perturb the heuristics used in the process, in order to find a better schedule. By default, random seeds are not used. They are controlled by the -r option (which tries a range of random seeds) and -rs option (which uses a specific random seed) described below in Section 5.3. In general, scheduling a kernel that does achieve the best possible schedule length with 10 random seeds may improve schedule length by one or two cycles.

5.2 Input Files

The kernel scheduler takes one or more KernelC files (_kc.cpp) as input. Each such file can contain one or more kernels, written using KernelC as described in Section 3.0.

OBSOLETE: Some older applications include .i files instead of _kc.cpp files. The present kernel scheduler uses a preprocessor to produce .i files from _kc.cpp files, but it can also handle this type of file as a direct input.

The kernel scheduler always takes one machine description (.md) file as an input that describes the target architecture. If binary microcode is a desired output, it also takes an opcode encoding file (JDO?) as an input. These file formats are documented elsewhere.

5.3 Command Line

The kernel scheduler has the following basic command line:

iscd -m <*machine description file*> <*global options*> <*kernel file 1*> <*local options for kernel file 1*> ... <*kernel file n*> <*local options for kernel file n*>

Which compiles all of the kernels in kernel file 1 through *n* for the target architecture described in the machine description file. The following is a typical command line for compiling the file test_kc.cpp:

iscd -m gold8.md test/test_kc.cpp

The kernel scheduler accepts the command line options shown in Table 8. Some options are preceded by [n], indicating that appending n to the start of the option turns it off. Some options have one or more arguments, denoted as "<*argument name*>". Options

can be used as either global options, which are specified before any kernel and affect all kernels or as local options, which are specified after a kernel and affect only that kernel. Some options can be only global or local, specified by “G” or “L” in the “Use” column.

TABLE 8. IScd Command Line Options

Option	Use	Description
Required options:		
-m <.md file>	G	machine description file name, default is gold8.md
Input options:		
-e <file>	G	opcode encoding file name used to generate raw microcode
-ne	L	suppress raw ucode generation for a file
-k <n>	L	schedule *n*th kernel only in file with multiple kernels, indexed from 0
-pre <command line> -pre_end	G	preprocess each C++ file using the command enclosed between -pre and -pre_end, the default is: -pre %iscd_preproc% /I %iscd_includes% /D "_IMAGINE_BUILD" -pre_end
-[n]syn	G	check the syntax of the kernel files only, do not schedule them, default = false
-[n]up <dir>	G	only schedule the kernel file for which there is not an up-to-date corresponding object file in object dir, default = false
Output file options:		
-o <name>	L	override the actual file name so that all output files are named by appending a suffix to *name*
-od <dir>	G	write all output files to specified directory
-[n]sum <file>	-	append summary line to text file showing schedule lengths of all blocks, default is true with file set to “summary.txt”
-uc <type>	G	set raw ucode output type, either “isim” for use with the cycle accurate simulator or “rtl” for use with the verilog simulator, default is “rtl”
-[n]v	-	generate SchedViz file, default is true
Optimization options:		
-[n]cons	-	use conservative software pipelining which can help performance for some kernels with many loop carried variables, default is false
-r <n> <s>	-	schedule kernel repeatedly with *n* random seeds used to perturb heuristics, starting with seed *s*, and output the best schedule

TABLE 8. ISCd Command Line Options

Option	Use	Description
-rf <n>	-	conserve registers using the *n*th of three levels: level 1 tries to load balance registers between the two register files attached to each functional units inputs, level 2 tries to load balance the register files attached to different functional units, level 3 actually rejects scheduling operations on heavily loaded functional units
-rs <n>	-	run with *n*th random seed
-b<II> -b<n>,<II>		sets first pipelined block to start with II sets nth (indexed from 0) pipelined block to start with II
Information display options:		
-[n]cd	-	show which operations are eliminated by copy propagation and dead code elimination, default is true
-[n]ops	-	print operations list to stdout, default is false
-[n]postops	-	print operations list after optimizations to stdout, default is false
-[n]scopes	-	print scopes list to stdout, default is false
-[n]t	-	show scheduling pass titles such as “COPY PROPIGATION”, default is true
-[n]vars	-	print variable list to stdout, default is false
Rarely used options:		
-file <file>	-	include contents of text file in command line
-io <mode>	-	OBSOLETE: use an alternate input/output mode
-mp <p>	-	minimize copy operations by randomly rejecting p% of routes that could be completed only with copy operation(s)
-lat <r> <w> <i>	-	add extra cycles of latency to all operations that read from a register file, write to a register file, or are stream inputs as given by integers *r, w,* and *i*, respectively
-opt <n>	-	special purpose optimizations, not for general use
-p <n>	-	set maximum number of copy operations used to complete a route, can be set lower to decrease run time with worse performance and loss of completion guarantee, or higher for architectures in which more than two copies are required to move values from one register file to another, default is 2
-s <method>	-	OBSOLETE: use alternate scheduling method such as list scheduling
-stat <n>	-	schedules kernel *n* times, where for *i*th schedule a maximum of *i* input or output operations can occur on a cycle

5.4 Command Line Output

The kernel scheduler displays which scheduling pass it is executing, a progress indicator when actually scheduling operations, and other important information directly to the command line. For example, when scheduling the example file test_kc.cpp it displays:

```
Starting scheduler in working directory: D:\working\im_apps                (1)
Parsing machine description file : gold8.md
Preprocessing program file: D:\working\im_apps\test\test_kc.cpp            (2)
%iscd_preproc% /I "%iscd_includes%" /D "_IMAGINE_BUILD" /P D:\work-
ing\im_apps\test\test_kc.cpp
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

test_kc.cpp
Parsing program file: D:\working\im_apps\test\test_kc.i                    (3)
************************************************************               (4)
Scheduling microassembly file: D:\working\im_apps\test\test_kc.i
BASIC BLOCKS:
DATA TRANSFERS:                                                            (5)
!!! WARNING: No write reaches read of: dummy !!!
  for: line:   28  COMMUCDATA        inputs: hw_mctrl#id dummy uc_sum  outputs:
sum
!!! WARNING: No write reaches read of: dummy !!!
  for: line:   54  COMMUCDATA         inputs: dummy sum  outputs: dummy
COPY PROPAGATION:                                                          (6)
DEAD CODE ELIMINATION:
  line:   19  INIT_COSTATE       inputs:  outputs: jb_c#2
  line:   18  INIT_CISTATE       inputs:  outputs: jb_b#1
  line:   17  INIT_CISTATE       inputs:  outputs: jb_a#0
COPY PROPAGATION:
DEAD CODE ELIMINATION:
INIT FEASIBLE FUS:                                                         (7)
CONSTRUCT DAG:
INITIALIZE REGISTERS:
COMPUTE SLACK:
SCHEDULE:                                                                  (8)
***** BLOCK: 1 ops: 7 (seed: 0 min/cur II: 0/-1a)                          (9)
.......                                                                    (10)

***** BLOCK: 0 ops: 9 (seed: 0 min/cur II: 0/-1a)
.........

***** BLOCK: 2 ops: 8 (seed: 0 min/cur II: 0/-1a)
.......
COMPACT BLOCK:                                                             (11)
  block: 0 best: 5 achieved: 7
  block: 1 best: 6 achieved: 6
  block: 2 best: 11 achieved: 11
ALLOCATE REGISTERS:                                                        (12)
// maximum register allocation:
// idx:   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
// max: 256  32   2  16  16  16  16  16  16  16  16  16  32  32  32  32  16  16
// use:   0   4   2   1   2   2   2   2   1   2   2   2   1   1   1   1   1   4

// idx:  18  19  20  21  22  23  24  25  26  27  28  29  30
// max:  16  16  16  16  16   1   1   1   1   1   1   1   1
// use:   1   1   1   1   1   0   0   0   0   0   0   0   0

// rf/functional unit mapping:
//  0:  0: SP_SC0   1: -1: COMM_0   2: -1:    MC0   3:  1: SP_SC0   4:  2: COMM_0
//  5:  3: ADDER0   6:  4: ADDER1   7:  5: ADDER2   8:  6: DIVID0   9:  7: ADDER0
// 10:  8: ADDER1  11:  9: ADDER2  12: 10: MULTI0  13: 11: MULTI1  14: 12: MULTI0
// 15: 13: MULTI1  16: 14: DIVID0  17: 15: COMM_0  18: 16: SP_SC0  19: 17: SP_SC0
// 20: 18: JUKEB0  21: 19: JUKEB0  22: 20: VALID0  23: 21: INOUT0  24: 22: INOUT1
// 25: 23: INOUT2  26: 24: INOUT3  27: 25: INOUT4  28: 26: INOUT5  29: 27: INOUT6
```

```
// 30: 28: INOUT7
*** Current best seed is: 0                                                    (13)
*** Best random seed: 0 ***                                                    (14)
Generating microcode file: D:\working\im_apps\test\test_kc.uc for microassembly
file: D:\working\im_apps\test\test_kc.i                                        (15)
Generating viz file: D:\working\im_apps\test\test_kc.viz for program file:
D:\working\im_apps\test\test_kc.i
```

Each of the numbered components of this output is explained in more detail below:

1. The working directory and machine description used:

```
Starting scheduler in working directory: D:\working\im_apps
Parsing machine description file : gold8.md
```

2. The preprocessor command line (note default uses environment variables):

```
Preprocessing program file: D:\working\im_apps\test\test_kc.cpp
%iscd_preproc% /I "%iscd_includes%" /D "_IMAGINE_BUILD" /P D:\work-
ing\im_apps\test\test_kc.cpp
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

test_kc.cpp
```

3. The kernel scheduler parser. KernelC syntax errors are displayed here:

```
Parsing program file: D:\working\im_apps\test\test_kc.i
```

4. The kernel scheduler passes. Unless the -nt option is used, the title of each pass is displayed, for example “BASIC BLOCKS”:

```
************************************************************
Scheduling microassembly file: D:\working\im_apps\test\test_kc.i
BASIC BLOCKS:
```

5. The data transfer pass. Unreached variable warnings are displayed here. In the example, on lines 28 and 56 of test_kc.cpp, ‘dummy’ is only used as a dummy argument so it is never written.

```
DATA TRANSFERS:
!!! WARNING: No write reaches read of: dummy !!!
  for: line:   28  COMMUCDATA        inputs: hw_mctrl#id dummy uc_sum  outputs:
sum
!!! WARNING: No write reaches read of: dummy !!!
  for: line:   56  COMMUCDATA         inputs: dummy sum  outputs: dummy
```

6. The iterative copy propagation and dead code elimination passes. Unless the -ncd option is used, the operations which removed due to copy propagation or dead code elimination are displayed here:

```
COPY PROPAGATION:
DEAD CODE ELIMINATION:
  line:   19  INIT_COSTATE       inputs:  outputs: jb_c#2
  line:   18  INIT_CISTATE       inputs:  outputs: jb_b#1
  line:   17  INIT_CISTATE       inputs:  outputs: jb_a#0
COPY PROPAGATION:
DEAD CODE ELIMINATION:
```

7. More scheduler passes, which do not produce messages:

```
INIT FEASIBLE FUS:
CONSTRUCT DAG:
INITIALIZE REGISTERS:
COMPUTE SLACK:
```

8. The actual scheduling pass. If multiple random seeds are used, this pass and all later passes are repeated for each seed:

```
SCHEDULE:

***** BLOCK: 1 ops: 7 (seed: 0 min/cur II: 0/-1a)
.......
```

9. The scheduling header for a basic block, showing the basic block number, the number of operations in the basic block, the random seed used, the minimum and current iteration interval in the case of software pipelining, and an “a” or “c” indicating aggressive (default) or conservative (using -cons option) software pipelining:

```
***** BLOCK: 1 ops: 7 (seed: 0 min/cur II: 0/-1a)
```

10. The scheduling progress for a block, consisting of “.” representing a scheduled operation, “p” representing a a scheduler induced copy operation, or “x”, indicating that a particular operation placement was abandoned due to excessive computation:

```
.......
```

The progress above is for a very simple basic block containing seven operations. The progress below is for a more complex basic block, showing software pipelining failing for a particular iteration interval. It displays zero or more “limit operations,” which particularly restrict placement of the current operation due to software pipelining restrictions, and the current operation. Scheduling then resumes with an iteration interval one higher.

```
***** BLOCK: 1 ops: 128 (seed: 5 min/cur II: 32/34a)
.........................p..............p.............................
.................p.p..p........p...xxxx.......p.p..
Limit op: line: 146 COMMUCDATA  inputs: srcAcur A.id  outputs: A.id  instr: 7

Limit op: line: 147 COMMUCDATA  inputs: srcBcur B.id  outputs: B.id  instr: 11

SWP failure: line:  117  ILE32  inputs: A.id B.id  outputs: tmp#63765
```

In rare cases, the scheduler may need to backtrack to a previous operation, possibly in an earlier basic block, and reschedule it and all later operations, which it indicates as follows:

```
***** BLOCK: 13 ops: 17 (seed: 0 min/cur II: 0/-1a)
......<
backtracking...

***** BLOCK: 11 ops: 35 (seed: 0 min/cur II: 0/-1a)
```

11. The compact block pass, which eliminates any empty cycles and displays the best possible (critical path length or minimum iteration interval) and achieved schedule length for each basic block:

```
COMPACT BLOCK:
  block: 0 best: 5 achieved: 7
  block: 1 best: 6 achieved: 6
  block: 2 best: 11 achieved: 11
```

12. The register allocation pass, which displays the maximum and used number of registers in each register file by index, then the mapping of indices to register file names. For instance, register file 17 at the end of the first set of lines contains 16 registers, 4

of which were used. It maps to the register file attached to the communication unit's input port 0 as shown by 17: 15: COMM_0. (The 15 indicates hardware index 15, and is used for tools debugging -- it can be ignored.)

```
ALLOCATE REGISTERS (new):
// maximum register allocation:
// idx:   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
// max: 256  32   2  16  16  16  16  16  16  16  16  16  32  32  32  32  16  16
// use:   0   4   2   1   2   2   2   2   1   2   2   2   1   1   1   1   1   4

// idx:  18  19  20  21  22  23  24  25  26  27  28  29  30
// max:  16  16  16  16  16   1   1   1   1   1   1   1   1
// use:   1   1   1   1   1   0   0   0   0   0   0   0   0

// rf/functional unit mapping:
//  0:  0: SP_SC0   1: -1: COMM_0   2: -1:    MC0   3:  1: SP_SC0   4:  2: COMM_0
//  5:  3: ADDER0   6:  4: ADDER1   7:  5: ADDER2   8:  6: DIVID0   9:  7: ADDER0
// 10:  8: ADDER1  11:  9: ADDER2  12: 10: MULTI0  13: 11: MULTI1  14: 12: MULTI0
// 15: 13: MULTI1  16: 14: DIVID0  17: 15: COMM_0  18: 16: SP_SC0  19: 17: SP_SC0
// 20: 18: JUKEB0  21: 19: JUKEB0  22: 20: VALID0  23: 21: INOUT0  24: 22: INOUT1
// 25: 23: INOUT2  26: 24: INOUT3  27: 25: INOUT4  28: 26: INOUT5  29: 27: INOUT6
// 30: 28: INOUT7
```

Register allocation is not guaranteed to succeed. In the case of a register allocation failure, the kernel scheduler displays the overflowing register file and the variable that will not fit, for example:

```
REGISTER ALLOCATION FAILURE! Unable to allocate register in RF: UNITRF_1_0
(index: 8) for variable: x
```

The kernel scheduler does not generate any output if it fails register allocation for all random seeds. For more information on dealing with a register allocation failure, see Section 11.1.

13. The best random seed found so far that passes register allocation, displayed after attempting to schedule the kernel with each random seed:

```
*** Current best seed is: 0
```

14. The best random seed found overall:

```
*** Best random seed: 0 ***
```

15. The output files generated:

```
Generating microcode file: D:\working\im_apps\test\test_kc.uc for microassembly
file: D:\working\im_apps\test\test_kc.i
Generating viz file: D:\working\im_apps\test\test_kc.viz for program file:
D:\working\im_apps\test\test_kc.i
```

5.5 Output Files

The kernel scheduler produces output files for each kernel it schedules. These files are placed in the same directory as the input file that contains the kernel unless the -od option is used. If the input file only contains one kernel, the output files for that kernel have the same name as the input file with different suffixes unless the -o option is used. If the input file contains more than one kernel, the output files for each kernel are named

by appending an underscore, the name of the specific kernel, and a different suffix to the name of the input file.

The kernel scheduler produces up to four output files for each kernel. First, it always generates a human-readable microcode (.uc) file. This file is parsed and executed by cycle accurate simulator, and some information is extracted from it by the functional simulator for profiling. Second, unless the -nviz option is used it generates a schedule visualizer (.viz) file that is used by the SchedViz tool to show a graphical representation of the scheduled kernel. Third, if the -e option is used, it generates a binary microcode (.raw) file and a special binary microcode file directly readable by Verilog (.lis) file.

5.6 Exercise, Step 4

1. Open a command prompt window and change to the im_apps directory
2. Schedule the kernel in test_kc.cpp using the following command line:

   iscd -m gold8.md test\test_kc.cpp

   This will generate the files test_kc.uc and test_kc.viz.

6.0 Run-time Tools Introduction

A stream application is linked at run time to two dynamic link libraries (.dlls), isim-host.dll and isim.dll. These libraries contain the run-time tools for Imagine: a functional simulator called IDebug, a cycle-accurate simulator called ISim, and a profile compiler for StreamC called IStream, each of which is described in detail in one of the following three chapters. This section provides a brief introduction to using these tools.

6.1 Simulator Script Files

Using the functional simulator or the cycle-accurate simulator requires a simulator script (.sim) file. These files can contain a wide variety of commands. Simulator script files are described in detail in Section 8.0. However, the following simulator script file suffices for most purposes:

```
t im
p /it/im/
run streamprogram ../hp "arguments"
go
```

Where *streamprogram* is the name of a stream program, the special function that serves as the “main” function of a stream application as described in Section 2.3.3, and *arguments* is an arbitrary, non-null string passed to that function as the second of its two arguments.

6.2 Using a Simulator

Both simulators are available by simply executing the stream application with particular arguments. The basic command line is as follows:

stream application -m <*machine description file*> -s <*simulator script file*> [-idb] <*options*>

Which simulates the stream application on the architecture described in the machine description file using the specified simulator script. If the command line includes the option -idb, the application is executed using IDebug, the functional simulator. Otherwise, it is executed using ISim, the cycle-accurate simulator.

The following is a minimal command line for executing the example application using IDebug:

test/Release/test.exe -m gold8.md -s test/test.sim -idb

The same command line without the -idb option executes the application using ISim:

test/Release/test.exe -m gold8.md -s test/test.sim

It is recommended that the -fht (fast host transfers) command line option always be used when running ISim, since it eliminates simulation of tedious host-Imagine data transfers.

The following is the typical progression of application development with the recommended command line options:

For debugging with IDebug use Debug build with -idb -np

For building a profile with IDebug use Release build with -idb -arp

For executing on ISim use Release build with -fht

6.3 Command line options

The run time tools have several command line options, described in the following table:

TABLE 9. Run-time Tools Command Line Options

Option	Description
Required options:	
-m <.md file>	machine description file name, default is gold8.md
-s <.sim file>	simulator script file name
IDebug options:	
-idb	run in iDebug mode
-np	disable profiling
ISim options:	
-h	print usage information
-s	specify file to read commands from
-m	specify machine description language file
-n	do not print debugging messages
-e	specify op encodings file
-d (obsolete)	specify base name for implicit -s, -m
-i (obsolete)	specify interactive mode override .sim file when using -d option
-q	terminate simulator on completion of .sim file
-0	suspend memory system during kernel execution
-lu	simulate loading of kernels from off-chip memory
-lu+file	don't simulate loading of kernels from memory
-lu+sim	simulate loading of kernels from memory from .lis files
-lu+real	simulate loading of kernels via the host processor using .raw and .uc files

TABLE 9. Run-time Tools Command Line Options

Option	Description
-fht	enable fast (not cycle-accurate) data transfers between the host and Imagine, recommended
-sls	enable .sim file loads and saves for StreamC to support verilog, see Section 11.3 for more details
IStream options:	
-arp	always rebuild profiles regardless of whether source file containing the profile statement has changed (recommend after any changes to other files)
-nrp	never rebuild profiles even if the source file containing the profile statement has changed (dangerous, use only when, for instance you have just changed a comment)
-nostrip	do not suggest stream sizes for stripmining as described in Section 9.4, may make profile generation faster
-stripsp <n>	make stripmining suggestions such that largest n% of countup or variable size/position streams recorded in the profile overflow
-pg	specify specially tagged source file containing a loop to be software pipelined as described in Section 9.5
-pg#	specify # specially tagged source files, where the first contains a loop to be software pipelined and successive files contain tagged functions called from within that loop
-ptags	show tags in pipelined source file
-sdb <root> <mode1> <mode2>	enable IDebug vs. ISim stream comparison, see tool developers documentation for more information

6.4 Data File Formats

The Imagine simulators support a variety of data file formats. UJK/JDO (graphics)?

Imagine data files in text format (used by streamLoadFile, streamCompareFile and streamSaveFile) are specified in the following manner. The files consist of one or more regions. Each region starts with a **:**, followed by a **T** as the first two characters on the line, and then a whitespace (**:T**).

Then the datatype is specified as in the following string:

```
:T F
```

where **F** is a single character data format specifier (identical to those used in the C library function *scanf*). For decimal integer data, use **d**; for hexidecimal integer data, use **X**; for floating-point data, use **E**.

Next, the data is specified, with each word separated by whitespace (either spaces, tabs, or newlines). For example, an integer stream with 8 data values from 0-7 would be specified as:

```
# data file with 8 integers, 0-7
:T d
0 1 2 3
4 5 6 7
```

- Note:

  1. Comments in data files are preceded by a **#** and are terminated by a newline.

  2. The file is read row-wise, i.e first we read the entries (going from left to right) in the the first row and then the second row and so on.

- To specify a data file with mixed datatypes (for example, a record with one integer and one floating point member), use multiple **:T** datatype specifiers:

```
# data file with 8 records (int/float)
:T d
1
:T E
100.0
:T d
2
:T d
200.0
...
```

- Also, an optional number can come right after the formatting character (no intervening space), to indicate that the number of bytes required to represent the numbers. Thus we use a '2' for 16-bit and a '4' for 8-bit numbers. stream operations on files check each 8-bit or 16-bit values to the correct length.
- Mixing the file's datatype and the datatype of the input record is not recommended. However, if this is done, the data is processed in the following way:

  1. The data is parsed using the *scanf* operator with the specified **:T** directive as the format argument to *scanf*.

  2. The data is written into an Imagine data word, which is a union of a *float* and an *int*.

  a. If that directive specifies a floating point format, the data is written as a *float*.

  b. If that directive does not specify a floating point format, the data is written as an *int*.

  3. When that data is then used as an *int* or a *float* (or an int or a float in a record), it appears as it is declared.

  As an example, if the data file is

```
:T E
1 2 3 ...
```

  and that data is read into a stream of *int*s, the value "1" will be written as a floating-point value 1.0 into the Imagine data word. When the stream of *int*s is actually used, that floating-point 1.0 will be treated as an int: 0x3f800000.

  Users who write data as one format and then use it as another format for computation or for saving to files will likely be surprised at the results.

6.5 Example, part 5

1. Create a text file called test/test.sim that contains the following:

```
t im
p /it/im/
run testProgram ../hp "no arguments"
go
```

7.0 IDebug, the Functional Simulator

IDebug is a functional simulator built into isimhost.dll and isimcore.dll. It includes a set of classes and functions that allow the direct execution of a stream application without simulated or actual Imagine hardware. It can be used in conjunction with a debugger such as that built into Visual C++ to debug a stream application, or to generate a profile for IStream. IDebug can be used for debugging without scheduled kernels as long as the -np (no profiling) command line option is used.

NOTE: Since the kernels are being used with Visual C++ support, and not IScd, they are not parsed and hence `unroll(n)` is ignored. This will be a matter of concern when you use `loop_count(len)`. The loop will always behave as if it has not been unrolled.

7.1 Using IDebug with a debugger

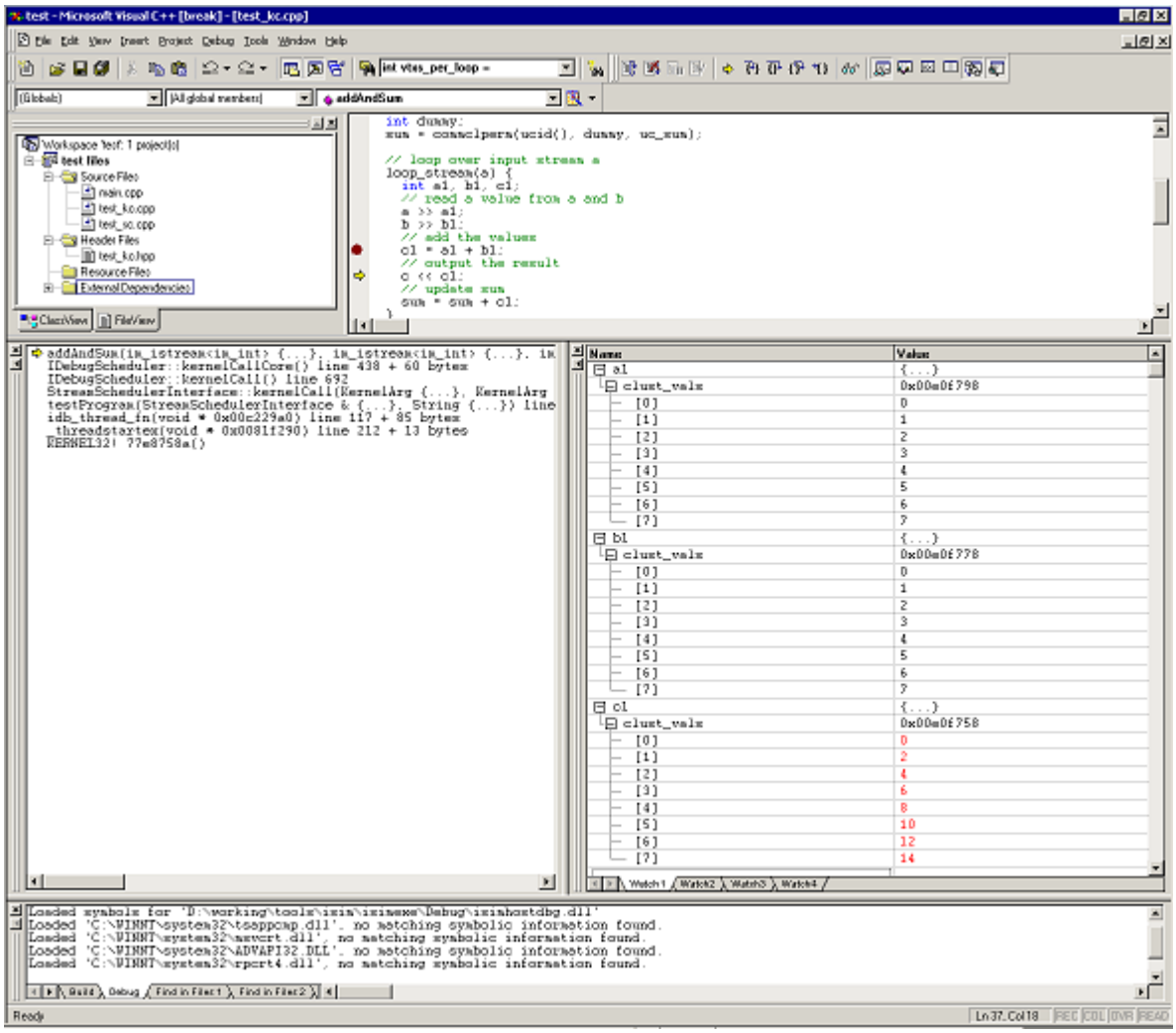
Using a debugger on a stream application simulated with IDebug is much like debugging any other application. The application must be compiled as a “debug” build. All of the conventional debugging tools can be used: breakpoints, single-stepping instructions, watches, stack traces, etc.

Under IDebug kernels work like normal functions, with two important exceptions. First, kernels are not called directly, and may be called in their own threads. Single-stepping into a kernel is laborious, and it is better to simply place a breakpoint at the start of a kernel and execute to that point. Second, all basic data types within a kernel (such as int) are actually classes with a property called *clust_vals*. *Clust_vals* is an array of eight values, the value of that variable in each of the eight clusters of Imagine.

For instance, the following screen shot shows Visual C++ debugger being used to debug the example stream application. A breakpoint is set on the line that computes the value of *c* inside the kernel *test*. The program has been single-stepped past this instruction, and the watches in lower right window show the values of the variables *a*, *b*, an *c* in the eight clusters.

It is important to note that IDebug allows the full range of C++ syntax to be used for printing debugging information, validating results, etc. Such code should always be enclosed in “#ifndef _IMAGINE_BUILD” / “#endif” preprocessor directives. This code is only executed under IDebug, and is not compiled by the kernel scheduler. For instance, the value of *a* inside the example kernel could be displayed to stdout using the following syntax:

```
#ifndef _IMAGINE_BUILD
  for (int i = 0; i < 8; i++) {
    cout << a.clust_vals[i] << “ ”;
  }
  cout << endl;
#endif
```



7.2 Exercise, part 6

1. Use IDebug to explore the functionality of the test program. Place a breakpoint at the start of the stream program, then compile and execute the application with the command line:

   test/Debug/test.exe -m gold8.md -s test/test.sim -idb -np

   Place a breakpoint as shown in the above figure. Run to that breakpoint. Examine the values of a, b, and c. Single step. Examine them again.

8.0 ISim, the cycle accurate simulator

ISim is the cycle accurate simulator used to gather performance results. The remainder of the section will discuss how to use the simulator command line interface, and how to debug applications as well as extract useful statistics.

8.1 ISim Semantics

usage: *isim <options>*

Command-line options can be found in Section 6.3.

8.2 ISim Commands:

These commands can be called from the command line interface of ISim.

Command	Shortcut	Description
assert		*check that a variable has a specified value* Format: assert <var> "<val>" The assert command is used to ensure that the variable <var> has the specified value <val>. If so, the assert command has no effect. If not, the assert command prints an informative error message.
debug		*set the current debug printing mode* Format: debug <mode> The debug command sets the amount of debug printing desired. <mode> is one of the valid debug printing modes (off, on, log, etc.) Modes: off: do not print any spurious information on: print information about the various states of pieces of the hardware to the screen log: same as "on", but also store the output in a buffer so that it can be displayed later using printlog or writelog
display	d	*display the value of a signal, module, or state variable* Format: display <object> The display command displays the specified signal, module, or state variable.

dump		*dump a range of a memory to a file* Format: dump <type> <object> "<filename>" <start> <length> "<arguments>" The dump command sends the range of memory <object> starting at <start> and continuing for <length> elements to the file named <filename>. The <type> parameter should be a string describing the data type (txt, bin, etc.). The <arguments> parameter is different for each <type> (and may be omitted for some types). For "txt", <arguments> should be a single character specifying the format in which the data should be written, using the C language printf '%' format printing codes.
go	g	*Step the simulator until the program completes* Format: go [<n>] Go will step the simulator until either ctrl-c is typed by the user, or the host interface signals that the current macrocode program is complete. The current cycle is printed every <n> cycles. <n> defaults to 10,000 if not specified.
help	h	*print help information* Format: help [<command>] Used by itself, help shows a list of available commands with a brief description of each. Help for a specific command provides more in depth information
include		*include command file* Format: include "<filename>" The include command is useful for running a bunch of commands stored in a file either interactively or within another file. All of the commands in the file named <filename> will be exectuted before the next command in the current file or at the prompt. The isim filename suffix convention for include files is ".sim".
printlog		*prints the specified log to the screen* Format: printlog "<log_name>" The printlog command can be used to display the "debug" or "error" log to the screen, so that it can be reviewed. (<log_name> must be "debug" or "error".)
prefix	p	*set the variable name prefix* Format: prefix [<string>] The prefix command sets a string which will be prepended to all variable names. If there are no arguments, the prefix is displayed. The prefix can be cleared by setting it to "".
quit	q	*terminates the simulator program* Format: quit Terminates the simulator program.

read		*read a file into a range of a memory* Format: read <type> <object> "<filename>" <start> "<arguments>" The read command reads a file into a range of memory <object> starting at <start> and continuing for the length of the file. <type> specifies the type of file (txt, bin, etc.). The <arguments> parameter is different for each <type> (and may be omitted for some types, such as "txt").
run		*run an application* Format: run <streamprogram> <host object> ["<args>"] The run command loads a streamprogram to be run. <streamprogram> is the name of a registered streamprogram (see Section 2.3.3). <host object> must be a Host_Processor object. <args> are optional arguments that will be passed directly to the streamprogram.
set		*set the value of a signal or state variable* Format: set <var> "<val>" The set command sets the signal or state variable <var> to the value <val>. <val> is interpreted differently for each type of variable.
show		*show all visible symbols with the current or given prefix* Format: show [<path>] Shows all visible symbols at a given level in the hierarchy. If no argument is given, the current prefix is used as the root. Otherwise, the given path is used to construct the root. Symbols ending with a '/' are not terminal symbols, they are another level of symbol path names - they can be seen by calling "show" with the extended path.
step	s	*steps the simulator one or more cycles* Format: step [[until] <n> [<m>]] The step command steps the simulator <n> cycles. The default is to step one cycle if <n> is not specified. The current cycle is printed every <m> cycles. <m> defaults to 10,000 if not specified. If "until" is specified, the simulator is stepped until cycle <n> instead of stepping <n> cycles.
test	t	*sets up the given test configuration* Format: test <test_name> The test command runs predefined test routines. The mapping from <test_name> to functions can be found in test.hpp.

vcmp		*compare an internal vector to the contents of a file* Format: vcmp <object> "<filename>" <start> [<threshold>] The vcmp command provides an internal interface to the vector compare program. It compares the vector starting at <start> in <object> to the vector contained in the file named <filename>. The number of elements compared is determined by the number of elements in the file. If this causes the simulator to read past the end of <object>, an error occurs. <threshold> sets the allowable relative difference between elements in the vector and elements in the file; if omitted, <threshold> defaults to 0.
writelog		*writes the specified log to the specified file* Format: writelog "<log_name>" "<file_name>" The writelog command can be used to write out the "debug" or "error" log to a file for later perusal. (<log_name> must be "debug" or "error"

8.3 Debugging

‘debug_info’ allows the programmer to look at the register state within isim, the Imagine simulator. (Note: the ‘-n’ command line option to ISim must not be used if debugging information is required.) Debug_info is a module and can be viewed just like any other module, using the "d" (display) command in isim. There are two types of debug_info modules. Each cluster has a debug_info module of its own that stores register state in that module. The cluster array also has an "umbrella" module that has pointers to each of the cluster modules. The two modules have identical semantics and calling a command from the umbrella module simply calls the same command on each of the cluster modules. However, the "#watch" family of commands should only be called on the umbrella module; don't call it on individual cluster modules.

The two module types live in /it/im/clust_array/debug_info (the umbrella module) and /it/im/clust_array/clusterX/debug_info (the cluster modules).

These are string arguments to debug_info; i.e.

```
isim> d ./clust_array/debug_info "string"

The valid strings are below (and error checking is spotty, so watch
it):

  /*
   * "#all":             Print out all values in RF, even hardwired
ones.
   * "#vars":            Print out all values in RF, except hardwired
ones.
   * "#regs":            Print out all values in RF, except hardwired
ones.
   * "#watch varname":   Watch the variable named varname
   *                     (i.e. print out each time the var is written)
   * "#unwatch varname": Stop watching the variable named varname
   * "#watch":           Print out all watched varnames
```

```
   * "varname":          Print out all vars that have that varname
   * "rf":              Print out all values in RFrf (rf is a number)
   * "rf reg":          Print out value in RFrf[reg] (rf and reg are
numbers)
   */
```

There are 2 kinds of printing:

#vars prints in var alphabetical order, starting with var name;

#regs prints in numerical order by regfile number (then reg number).

If you provide no argument it will do "#vars" by default. Printing "rf reg" will print the value of the register even if that value is no longer alive, although the variable name info will be lost.

8.4 Statistics

Statistics on ISim runs can be gathered using a variety of “stats” modules. These modules have a similar structure to debug_info; displaying them shows the statistics that the module has compiled. (Note: the ‘-n’ command line option to ISim must not be used if statistics are required.)

Four statistics modules are currently supported: clusters/functional units, the microcontroller, the SRF, and the memory system. As an example, the .sim file usually used to gather simulation statistics is:

```
displaylog "clear"
setlog "display" logonly

d /it "-cycle"
d /it/im/mct "stats -num_stalls -busy_cycles -all"
d /it/im/ms "stats"
d /it/im/clust_array/cluster0/stats
d /it/im/srf/stats

// writelog "display" ".txt"
```

The last line is commented out, but running it with a real file name (and removing the comment line //) saves the statistics to a file.

8.4.1 Stats for clusters and function units

Stats gives instruction counts and percentages on a per-function-unit basis. Like debug_info, cluster stats can be printed from one of two places: in the cluster from an umbrella module (each cluster will work, but they should all be the same) or from a functional unit. The cluster stats umbrella just calls each functional unit's stats. Examples of both are below.

These are dynamic instruction counts. Static instruction counts are at the bottom of the .uc file. Note also that the stats include the time for memory initialization, etc. (so there's an awful lot of "NONE"s), as I suppose they should; running several kernels will only generate cumulative stats. It is currently implemented quite simply and ticks once each time the functional unit's "evaluate()" function is called, so it doesn't handle stalls correctly at this time.

The command

```
d ./stats "reset"
```

will reset stats, called from either the cluster or a functional unit, so the programmer can have a warm start for the stats info.

```
[ 28737] isim> p ./clust_array/cluster0/
p ./clust_array/cluster0/
prefix: /it/im/clust_array/cluster0/

[ 28737] isim> d ./fu/fu0/stats
d ./fu/fu0/stats
/it/im/clust_array/cluster0/fu/fu0/stats (module): Instr count: 28737
    NONE 26657 (92.8%)
    FADD 320 (1.1%)
    FSUB 800 (2.8%)
    ISELECT32 960 (3.3%)


[ 28737] isim> d ./stats
d ./stats
/it/im/clust_array/cluster0/stats (module):
FU0: Instr count: 28737
    NONE 26657 (92.8%)
    FADD 320 (1.1%)
    FSUB 800 (2.8%)
    ISELECT32 960 (3.3%)
FU1: Instr count: 28737
    NONE 26657 (92.8%)
    FADD 640 (2.2%)
    FSUB 960 (3.3%)
    ISELECT32 480 (1.7%)
FU2: Instr count: 28737
    NONE 26817 (93.3%)
    PASS 160 (0.6%)
    FADD 960 (3.3%)
    FSUB 160 (0.6%)
    ISELECT32 640 (2.2%)
FU3: Instr count: 28737
    NONE 27137 (94.4%)
    ISELECT32 320 (1.1%)
    FMUL 1280 (4.5%)
FU4: Instr count: 28737
    NONE 27297 (95.0%)
    ISELECT32 160 (0.6%)
    FMUL 1280 (4.5%)
FU5: Instr count: 28737
    NONE 27617 (96.1%)
    PASS 1120 (3.9%)
FU6: Instr count: 28737
```

```
    NONE 28737 (100.0%)
FU7: Instr count: 28737
    NONE 28737 (100.0%)
FU8: Instr count: 28737
    NONE 23617 (82.2%)
    COMMUCI 5120 (17.8%)
```

8.4.2 Stats for SRF

The SRF's stats module is located in /it/im/srf/stats. It is a module much like the cluster stats and can be displayed using the following command:

```
d /it/im/srf/stats
```

It displays the total number of reads and writes from and to the SRF as well as which streambuffer was responsible for the reads and writes. Dividing the total number of reads and writes by the total number of cycles gives the SRF achieved bandwidth. Sample output is:

```
/it/im/srf/stats (module):
  SRF Total Reads: 5426112 [ cycles reading: 169566 (5.09823%) ]
    SB0 (/it/im/srf/sbuf0): 2647712
    SB1 (/it/im/srf/sbuf1): 1678176
    SB2 (/it/im/srf/sbuf2): 349952
    SB3 (/it/im/srf/sbuf3): 308736
    SB4 (/it/im/srf/sbuf4): 0
    SB5 (/it/im/srf/sbuf5): 0
    SB6 (/it/im/srf/sbuf6): 0
    SB7 (/it/im/srf/sbuf7): 0
    SB8 (/it/im/srf/sbuf8): 21664
    SB9 (/it/im/srf/sbuf9): 0
    SB10 (/it/im/srf/sbuf10): 212896
    SB11 (/it/im/srf/sbuf11): 71456
    SB12 (/it/im/srf/robuf12): 72864
    SB13 (/it/im/srf/robuf13): 62656
    SB14 (/it/im/srf/sbuf14): 0
    SB15 (/it/im/srf/sbuf15): 0
    SB16 (/it/im/srf/sbuf16): 0
    SB17 (/it/im/srf/sbuf17): 0
    SB18 (/it/im/srf/sbuf18): 0
    SB19 (/it/im/srf/sbuf19): 0
    SB20 (/it/im/srf/sbuf20): 0
    SB21 (/it/im/srf/sbuf21): 0
  SRF Total Writes: 4520896 [ cycles writing: 141278 (4.24772%) ]
    SB0 (/it/im/srf/sbuf0): 0
    SB1 (/it/im/srf/sbuf1): 504064
    SB2 (/it/im/srf/sbuf2): 1145408
    SB3 (/it/im/srf/sbuf3): 1107808
    SB4 (/it/im/srf/sbuf4): 800000
    SB5 (/it/im/srf/sbuf5): 119040
    SB6 (/it/im/srf/sbuf6): 119040
    SB7 (/it/im/srf/sbuf7): 51520
    SB8 (/it/im/srf/sbuf8): 0
    SB9 (/it/im/srf/sbuf9): 0
    SB10 (/it/im/srf/sbuf10): 0
    SB11 (/it/im/srf/sbuf11): 0
    SB12 (/it/im/srf/robuf12): 644288
    SB13 (/it/im/srf/robuf13): 29728
```

```
    SB14 (/it/im/srf/sbuf14): 0
    SB15 (/it/im/srf/sbuf15): 0
    SB16 (/it/im/srf/sbuf16): 0
    SB17 (/it/im/srf/sbuf17): 0
    SB18 (/it/im/srf/sbuf18): 0
    SB19 (/it/im/srf/sbuf19): 0
    SB20 (/it/im/srf/sbuf20): 0
    SB21 (/it/im/srf/sbuf21): 0
  SRF Cycles Idle: 3015131 (90.654%)
```

8.4.3 Stats for the Microcontroller

The microcontroller statistics are accessed by displaying the microcontroller module “mct” with the argument “stats”. It contains statistics on the total number of busy cycles, the number of cluster stalls, and the busy cycles and stalls as well as the number of invocations for each kernel. Stalls are generated using the “-num_stalls” argument; the number of busy cycles, with “-busy_cycles”; and the kernel statistics with “-all”.

```
d /it/im/mct "stats -num_stalls -busy_cycles -all"

/it/im/mct (module):
# Stalls : 82576
# Busy Cycles : 2740028

Statistics for kernel : render_sc/rtsl_advs/vertex_program_kc.uc
---------------------------------

Total Statistics :
# invocations : 218
Total Cycles : 109436
Total Stalls : 436


Statistics for kernel : render_sc/rtsl_advs/perbegin_program_kc.uc
---------------------------------

Total Statistics :
# invocations : 1
Total Cycles : 64
Total Stalls : 3

...
```

8.4.4 Stats for the Memory System

The memory system statistics are accessed by displaying the memory system module “ms” with the argument “stats”. Among the statistics are found the number of loads and stores on each address generator (which can be used to calculate memory bandwidth) and statistics for each memory bank and reorder buffer. An example output is displayed below:

```
d /it/im/ms "stats"

/it/im/ms (module):
 Live: 3325975

 AG 0
```

```
  loads: 638584 stores: 67392 active: 929848 srf stalls: 180553 bank
stalls: 43319
  load: stride: streams: 403 count: 503752  indirect: streams: 432
count: 134832  bitrev: streams: 0 count: 0
  store: stride: streams: 2 count: 352  indirect: streams: 369 count:
67040  bitrev: streams: 0 count: 0

 AG 1
  loads: 28584 stores: 60872 active: 152960 srf stalls: 20492 bank
stalls: 43012
  load: stride: streams: 52 count: 20712  indirect: streams: 78 count:
7872  bitrev: streams: 0 count: 0
  store: stride: streams: 2 count: 56  indirect: streams: 151 count:
60816  bitrev: streams: 0 count: 0

 Bank 0

  Bank Buffer:   32/32  stalled: 144970
  Return Buffer: 4/32  stalled: 207
  Hold Buffer 0: 2/2  stalled: 9667
  Hold Buffer 1: 2/2  stalled: 9805

  store latency:  min: 2 max: 753 avg: 114 (total: 33548)
  max mshr packets: 38

 Bank 1

  Bank Buffer:   32/32  stalled: 143911
  Return Buffer: 7/32  stalled: 838
  Hold Buffer 0: 2/2  stalled: 9580
  Hold Buffer 1: 2/2  stalled: 9815

  store latency:  min: 2 max: 921 avg: 117 (total: 31770)
  max mshr packets: 40

 Bank 2

  Bank Buffer:   32/32  stalled: 143251
  Return Buffer: 8/32  stalled: 317
  Hold Buffer 0: 2/2  stalled: 7289
  Hold Buffer 1: 2/2  stalled: 6997

  store latency:  min: 2 max: 937 avg: 109 (total: 31588)
  max mshr packets: 45

 Bank 3

  Bank Buffer:   32/32  stalled: 144236
  Return Buffer: 2/32  stalled: 550
  Hold Buffer 0: 2/2  stalled: 7726
  Hold Buffer 1: 2/2  stalled: 8712

  store latency:  min: 2 max: 1385 avg: 112 (total: 31358)
  max mshr packets: 38

 ROB 0

  Buffer 0: 4/4
  Buffer 1: 4/4
  Buffer 2: 4/4
```

```
 Buffer 3: 4/4

 load latency:  min: 5 max: 337 avg: 36 (total: 638584)

ROB 1

 Buffer 0: 4/4
 Buffer 1: 4/4
 Buffer 2: 4/4
 Buffer 3: 4/4

 load latency:  min: 6 max: 317 avg: 62 (total: 28584)
```

8.5 Microcode Breakpoints

Microcode breakpoint commands can be entered from the command-line interface in the simulator by setting the mc_store with a string. So, a sample command would look like :

```
set ./mct/mc_store "command"
```

The command formats are :

```
s_line  name  #
c_line  name  #
s_idx   name  #
c_idx   name  #
c_all
```

The *c_all* command clears all previously set breakpoints. For the other commands, the name field refers to the name of the microprogram. This is simply the name of the microcode file stripped of the the path and extension; i.e., sort/sort8.uc would have the name "sort8".

The *s/c_line* commands will only work if the microcode file was produced by scheduling in sequential mode (and will give an error message as a reminder if this is not the case). This is necessary because the # field refers to a line number from the original microassembly file. This is useful because it saves the time spent looking for certain op in the microcode file. The *s* prefix stands for set and the *c* prefix for clear.

The *s/c_idx* commands are similar except the # field in these commands refer to a relative index into the microcode program as stored in the microcode store. In other words, this number is exactly the instr # which can be found in the microcode file. One must refer to the microcode file to reliably ascertain where to set the breakpoint for these commands. These two commands can be used with any scheduling method.

Let’s look at a quick example. The following shows an example of how to set a breakpoint in at instruction #5 in the *mpeg_sc/rle_kc.uc* kernel when executing the *mpeg_sc/mpeg2_regress.sim* application.

```
[ 48598] /it/im/sc :          Finished : Y dct
[ 48605] /it/im/sc : Starting          : CrCb dct

   ( <Ctrl-C> hit here. )
```

```
[ 48757] isim> set ./mct/mc_store "s_idx mpeg_sc/rle_kc.uc 5"
```

Now, we can start execution again with the *g* command, and as is shown in the output below, the simulator stops execution at the breakpoint.

```
[ 48757] isim> g
[ 48914] /it/im/sc :          Finished : CrCb dct
[ 48925] /it/im/sc : Starting          : RLE
[ 48938] /it : Microcode breakpoint reached
```

To verify we are at the right point, we can display the status of the microcontroller:

```
[ 48938] isim> d ./mct
/it/im/mct (module):

idle : 0  |  stall : 0  |  MPC : 165  |  end_cnt : 0  |  cached_pc :
   4294967295
prime_ctr : 0  |  drain_ctr : 0  |  chk_stage : 0  |  last_stage : 0
Ucode Load : ---

mcode_store[MPC] : MicroInstructions :
Instruction #  -- Absolute:  165
Instruction #  -- Relative:    5  of program : mpeg_sc/rle_kc.uc
Valid Bit         : 1
Breakpoint status : set
MC_instr          :   op : UC_DATA_IN  |  IMM : 1732580128  |  UCRF
   Write Port 0 : 4  |  END_FLAG : FALSE  |  RF[2][0] Stage : -1  |  In
   2 Stage : -1  |  UCRF 0 Stage : -1
Cluster_Microcode :
Reg Control for register files:
  R2: { RdAddrs; WrAddrs; InBuses; OutBuses }:  { ( x ) ( 1 ) ( 24 ) (
   ( ) ) }
InCtrl Mappings: I2 => B24;
OutCtrl Mappings:
```

To remove the breakpoint, the following command would be used. After that, execution will no longer stop were the breakpoint was once set.

```
[ 48938] isim> set ./mct/mc_store "c_idx mpeg_sc/rle_kc.uc 5"
```

8.6 Simulator Example

So, how is the simulator actually used? Let's look at "fft8c.sim", the simulator file we use to test regression on our 1024-point complex 8 cluster FFT implementation. Here (in its entirety) is the file:

---- *begin sim file* ---

```
t im
p /it/im/

// FFT
read txt ./ms/data "fft\input.vfft" 0
read txt ./ms/data "fft\twiddle1-10.vfft" 0x1000
run fft ../hp "8c"
go
vcmp ./ms/data "fft\bitrev.vfft" 0x800 0.005
```

```
printlog error
q
```

---- *end sim file* ----

Let's break it down, since most sim files are very similar. User input (either manually or from a sim file) is indicated by "isim> " below (an actual isim> prompt is preceded by the global cycle number); isim's response is preceded by a "> ".

```
isim> t im
```

This command instantiates an Imagine simulated processor.

```
isim> p /it/im/
```

This is the 'prefix' command. Since all pieces of the Imagine simulator are instantiated as modules in a hierarchical manner, the 'p' command us used to navigate to different levels of the heirarchy. "/it/im" is, in some sense, the "root directory" of the Imagine processor which we're simulating. Isim responds to the "prefix" command with:

```
> prefix: /it/im/

isim> read txt ./ms/data "fft\input.vfft" 0
> Read 2048 values.
isim> read txt ./ms/data "fft\twiddle1-10.vfft" 0x1000
> Read 10240 values.
```

Now, we'd like to take a FFT of an input data set so we'll read it into memory at memory address 0. The input file with our data set is in text format ("txt") and it is in source file "fft\input.vfft" (files are accessed relative to the current working directory). The destination for this input data set is our memory system's data ("./ms/data"). Similarly, the twiddle factors for the FFT are read into memory location 0x1000. The length of the read is determined by the length of the source file, and the simulator responds to the read command with the length read.

(If we had not called the "p /it/im/" command earlier, the read command would have had to have looked like this:

```
  isim> read txt /im/it/ms/data "fft\input.vfft" 0
```

)

Now we'd like to run the fft kernel on this data set:

```
isim> run fft ../hp "8c"
isim> go
```

"fft" names a kernel instance in mcprog.h, which in turn maps to the macrocode we called "fft" defined in app_fft.cpp. The macrocode will run on the host processor ("../hp", or alternatively, "/im/hp") and take arguments "8c" (8 cluster). Then we kick off the kernel with "go", and the simulator responds with:

```
> [     1] Imagine Starting
> [ 10000]
> [ 12663] MEM LOADS FINISHED
```

```
> [ 17205] COMPUTATION FINISHED
> [ 20000]
```

"Go" runs the kernel to completion; the macrocode prints out progress messages, and by default the simulator prints its cycle number every 10000 cycles. (Use "go" with an argument to change that, as described above.) When we're done we'd like to check that it's run correctly:

```
isim> vcmp ./ms/data "fft\bitrev.vfft" 0x800 0.005
> The number of elements compared = 2048
> The maximum relative difference = 0.004925 (for element 1023, start-
ing at 1)
> The maximum absolute difference = 0.000006 (for element 3, starting
at 1)
> The average absolute difference = 0.000000
```

which tests the contents of memory starting at address 0x800 against the expected output contained in "fft\bitrev.vfft" which a tolerance of 0.005 (0.5%). The simulator indicates that the differences were within our tolerance; if the differences were above our indicated tolerance then an error message prints out.

Finally, if there are errors they'd be contained in the error printlog:

```
isim> printlog error
```

and at the end we quit.

```
isim> q
```

Now, let's say we want to do the same run but try a couple of different things. First, I'd like to make sure the correct data is coming into the twiddle factor stream, so I'd like to look at the first 200 cycles or so of the computation time as the first few twiddle factors are read into the clusters. To do this I can use the "watch" facility of the debug_info module. Also, I'd like to get statistics on functional unit utilization for only the calculation part of the fft kernel. Armed with the cycle counts from the run above I can add a few lines to the sim file above and check both.

The sim file and its output (which is a little long) are attached below.

---- *begin sim file* ----

```
t im
p /it/im/

// FFT
read txt ./ms/data "fft\input.vfft" 0
read txt ./ms/data "fft\twiddle1-10.vfft" 0x1000
run fft ../hp "8c"
s 12663
d ./clust_array/cluster0/stats "reset"
d ./clust_array/debug_info "#watch w.r"
d ./clust_array/debug_info "#watch w.i"
s 200
d ./clust_array/debug_info "#unwatch w.r"
d ./clust_array/debug_info "#unwatch w.i"
```

```
s 4342
d ./clust_array/cluster0/stats
go
vcmp ./ms/data "fft\bitrev.vfft" 0x800 0.005
printlog error
```

---- *end sim file* ----

---- *begin sim output* ----

```
Y:\im_apps>y:\tools\isim\release\isim.exe -s fft\fft8c.sim -m
gold8.md
y:\tools\isim\release\isim.exe -s fft\fft8c.sim -m gold8.md

Y:\im_apps>echo off

[     0] isim> t im

[     0] isim> p /it/im/
prefix: /it/im/

[     0] isim>

[     0] isim> // FFT

[     0] isim> read txt ./ms/data "fft\input.vfft" 0
Read 2048 values.

[     0] isim> read txt ./ms/data "fft\twiddle1-10.vfft" 0x1000
Read 10240 values.

[     0] isim> run fft ../hp "8c"

[     0] isim> s 12663
[     1] Imagine Starting
[ 10000]
[ 12663] MEM LOADS FINISHED

[ 12663] isim> d ./clust_array/cluster0/stats "reset"
/it/im/clust_array/cluster0/stats (module):

TOTAL: Instr count: 139293
    NONE 139293 (100.0%)
FU0: Reset.
FU1: Reset.
FU2: Reset.
FU3: Reset.
FU4: Reset.
FU5: Reset.
FU6: Reset.
FU7: Reset.
FU8: Reset.
FU9: Reset.
FU10: Reset.
SRF usage: Reads[0:4]: 0 0 0 0
         : Writes[0:4]: 0 0 0 0
```

```
[ 12663] isim> d ./clust_array/debug_info "#watch w.r"
/it/im/clust_array/debug_info (module):


[ 12663] isim> d ./clust_array/debug_info "#watch w.i"
/it/im/clust_array/debug_info (module):


[ 12663] isim> s 200
[ 12680] w.r: 0x 3f800000 3f7ffec5 3f7ffb11 3f7ff4e6 3f7fec43 3f7fe129
3f7fd398 3f7fc38e
[ 12681] w.i: 0x 00000000 bbc90f89 bc490e92 bc96c9b6 bcc90ab1 bcfb49b8
bd16c32b bd2fe007
[ 12687] w.r: 0x 3f7fb110 3f7f9c19 3f7f84ab 3f7f6ac8 3f7f4e6e 3f7f2f9d
3f7f0e58 3f7eea9d
[ 12688] w.i: 0x bd48fb2e bd621467 bd7b2b75 bd8a200a bd96a904 bda3308c
bdafb680 bdbc3ac4

... bunch of w.r, w.i lines removed for brevity ...

[ 12855] w.r: 0x 3eac7cd5 3ea986c4 3ea68f12 3ea395c4 3ea09ae3 3e9d9e76
3e9aa084 3e97a117
[ 12856] w.i: 0x bf710909 bf718f58 bf721352 bf7294f8 bf731448 bf73913f
bf740bdd bf748422
[ 12862] w.r: 0x 3e94a032 3e919ddc 3e8e9a22 3e8b9508 3e888e95 3e8586ce
3e827dc2 3e7ee6e0
[ 12863] w.i: 0x bf74fa0a bf756d98 bf75dec6 bf764d97 bf76ba08 bf772416
bf778bc6 bf77f110

[ 12863] isim> d ./clust_array/debug_info "#unwatch w.r"
/it/im/clust_array/debug_info (module):
CL0:
CL1:
CL2:
CL3:
CL4:
CL5:
CL6:
CL7:


[ 12863] isim> d ./clust_array/debug_info "#unwatch w.i"
/it/im/clust_array/debug_info (module):
CL0:
CL1:
CL2:
CL3:
CL4:
CL5:
CL6:
CL7:


[ 12863] isim> s 4342
[ 17205] COMPUTATION FINISHED

[ 17205] isim> d ./clust_array/cluster0/stats
/it/im/clust_array/cluster0/stats (module):
```

```
TOTAL: Instr count: 49764
    NONE 27215 (54.7%)
    FADD 1932 (3.9%)
    IADD32 2 (0.0%)
    FSUB 1932 (3.9%)
    ISUB32 1 (0.0%)
    AND 1 (0.0%)
    ILT32 1 (0.0%)
    SELECT 2580 (5.2%)
    NSELECT 6440 (12.9%)
    FMUL 2576 (5.2%)
    COMMUCPERM 2576 (5.2%)
    SPREAD 1932 (3.9%)
    SPWRITE 1932 (3.9%)
    SPRW 644 (1.3%)
FU0: Instr count: 4524
    NONE 1947 (43.0%)
    FADD 644 (14.2%)
    IADD32 1 (0.0%)
    FSUB 644 (14.2%)
    SELECT 644 (14.2%)
    NSELECT 644 (14.2%)
FU1: Instr count: 4524
    NONE 2590 (57.3%)
    FADD 1288 (28.5%)
    AND 1 (0.0%)
    ILT32 1 (0.0%)
    NSELECT 644 (14.2%)
FU2: Instr count: 4524
    NONE 1946 (43.0%)
    IADD32 1 (0.0%)
    FSUB 1288 (28.5%)
    ISUB32 1 (0.0%)
    SELECT 644 (14.2%)
    NSELECT 644 (14.2%)
FU3: Instr count: 4524
    NONE 1947 (43.0%)
    SELECT 645 (14.3%)
    NSELECT 644 (14.2%)
    FMUL 1288 (28.5%)
FU4: Instr count: 4524
    NONE 2591 (57.3%)
    SELECT 1 (0.0%)
    NSELECT 644 (14.2%)
    FMUL 1288 (28.5%)
FU5: Instr count: 4524
    NONE 1946 (43.0%)
    SELECT 646 (14.3%)
    NSELECT 1932 (42.7%)
FU6: Instr count: 4524
    NONE 4524 (100.0%)
FU7: Instr count: 4524
    NONE 16 (0.4%)
    SPREAD 1932 (42.7%)
    SPWRITE 1932 (42.7%)
    SPRW 644 (14.2%)
FU8: Instr count: 4524
    NONE 660 (14.6%)
    NSELECT 1288 (28.5%)
```

```
    COMMUCPERM 2576 (56.9%)
FU9: Instr count: 4524
    NONE 4524 (100.0%)
FU10: Instr count: 4524
    NONE 4524 (100.0%)
SRF usage: Reads[0:4]: 1280 1280 1280 0
         : Writes[0:4]: 2576 0 0 0


[ 17205] isim> go

[ 25381] isim> vcmp ./ms/data "fft\bitrev.vfft" 0x800 0.005
The number of elements compared = 2048
The maximum relative difference = 0.004925 (for element 1023, starting
at 1)
The maximum absolute difference = 0.000006 (for element 3, starting at
1)
The average absolute difference = 0.000000

[ 25381] isim> printlog error

[ 25381] isim>

[ 25381] isim>

[ 25381] isim>
```

---- *end sim output* ----

9.0 IStream, the Profile Compiler

IStream is a profile compiler built into the Imagine simulators. It records a trace of the stream operations that are executed by a stream program, called a profile, with special annotations for any variations in the operations that depend on the data being processed, then compiles that profile. IStream allocates resources such as the stream register file much more efficiently than a run-time method like that used by ISim for an un-profiled stream application.

9.1 Preparing an application for profiling

9.1.1 What to profile?

While it is possible to profile an entire stream program, a profile usually excludes loading initial input streams from the host and saving final output streams to the host. Since stream variables declared outside of the profile cannot refer to stream data allocated inside the profile, stream data for final output streams needs to be allocated before the start of the profile. Thus, a typical profiled stream program has the conceptual format:

```
stream<Foo> initialInput = newStreamData<Foo>(100);
stream<Bar> finalOutput = newStreamData<Bar>(100);
streamLoadBin(..., initialInput);
profile(“myProfile”) {
   firstKernel(initialInput, ...);
   ...
   lastKernel(..., finalOutput);
}
streamSaveBin(..., finalOutput);
```

9.2 How to annotate it?

Since a profile records the stream operations executed, any variations in the stream operations or streams must be explicitly annotated. See Section 4.9 for a description of these annotations.

9.2.1 What input data to use?

In the case of applications with data-dependent control flow, the profile must be generated using input data chosen so that all control-flow blocks (e.g. data-dependent if statements and loops) are executed at least once. If a control-flow block is not executed, IStream will generate a warning (see below). ISim will generate a fatal error if it attempts to enter that block when using the compiled profile. For example, if an application contains the following code:

```
stream<Foo> a = newStreamData<Foo>(100);
stream<Bar> b = newStreamData<Bar>(500, im_countup);
...
// one Foo converts to a data-dependent number of Bars
convertFoosToBars(a, b);
if_VARIABLE (b.getLength() >= 250) {
   compressBars(b, b);
}
```

Then the profile must be generated using input data such that the Foo records in *a* will be converted into at least 250 Bar records in *b*. If this code was inside a loop, this requirement would only have to be met on one iteration.

9.3 Generating a profile

To generate a profile, execute the application once using IDebug (or ISim, though IDebug is recommended). By default, a profile will only be generated if the _sc.cpp file containing the profile statement has changed, so it is best to always execute the application once with the -arp option in order to force rebuilding of the profile after any changes.

9.3.1 Command line output

When executing the application to generate a profile, the output like the following will be displayed at the start and end of the profiled section of the stream program, respectively:

```
****************************************
*                                      *
Starting profile 'bigtest/bigtestProgram'...
Always rebuild profiles (-arp) option enabled.
Generating new profile...
----------------------------------------

. . .

----------------------------------------
Ending profile 'bigtest/bigtestProgram_debug'...
Total execution time: 129865
Generating .pro file...
NOTE: Profile information has been saved as: 'bigtest/
bigtestProgram_dinfo.txt'.
Profile contains 3 spilled streams!
Profile contains 2 double-buffered streams!
Profile contains 1 sequential memory accesses!
Profile contains 1 empty data dependent control flow blocks! *
*
****************************************
```

The block displayed at the end of profiling contains important information. First, it contains the name of the profile. Second, it contains the total execution time of the profiled portion of the application. If the application was executed using IDebug this is only an estimate of total execution time of the kernels. Third, it displays the text “Generating .pro file...” followed by the name of the profile information file. Lastly, it displays important warnings about the various inefficiencies or problems in the profiled application including:

- Spilled streams, streams which could not be retained in the SRF between accesses due to size constraints and were therefore stored to memory and reloaded
- Double-buffered streams, streams which could not be fit in the SRF for at least one access and so were rotated though the SRF using double-buffering

- Sequential memory accesses, stream which were produced by one kernel and then used as an index to load an input to the next kernel, or used as part of an input stream to the next kernel which required the data being stored back to memory (because it was accessed with a different stride, for instance.)
- Empty control flow blocks, control flow blocks which do not contain any profiled stream operations, usually indicating that these blocks were not executed during the run.

9.3.2 Output files

Profiling an application results in multiple files. These files are named by appending different suffixes to the *profilename* argument passed to the profile statement.

- <profilename>.pro - Profile data file containing recorded operations
- <profilename>_info.txt - Important information about profile
- <profilename>_mar.viz - Memory access register (MAR) allocation
- <profilename>_mem.viz - Off-chip memory allocation
- <profilename>_nrr.viz - Network routing register (NRR) allocation
- <profilename>_sbr.viz - Stream buffer, receiving allocation
- <profilename>_sbs.viz - Stream buffer, sending allocation
- <profilename>_sdr.viz - Stream descriptor register (SDR) allocation
- <profilename>_srf.viz - Stream register file (SRF) allocation

The files consist of three main categories. First, a profile data file (.pro) that contains the recorded stream operations. This file is not human readable and is the only file that is actually used by the profile compiler for later runs of the application. Second, a human readable text file containing information about the profile. And third, a series of Sched-Viz files containing the resource allocations generated by IStream. The most important of these files is the SRF allocation (_srf.viz), which shows which streams (if any) were spilled or double-buffered as described in Section 10.3.

9.3.3 Profile information (_info.txt) file

The profile information file contains of the following parts.

1. Executive summary that repeats the inefficiency and problem counts shown at the command line, and lists the recorded stream operations with their input and output streams.

```
*****************************************
*                                       *
  EXECUTIVE SUMMARY:
  spilled streams: 0
  double-buffered streams: 0
  sequential memory accesses: 0
  empty control flow blocks: 0
-----------------------------------------
0 kernelLoad (  IN: 'addAndSum ucode'  )
1 addAndSum (  IN: 's1' 's1'  OUT: 'temp'  )
2 addAndSum (  IN: 's1' 'temp'  OUT: 's2'  )
*                                       *
*****************************************
```

2. Each stream operation with statistics for each kernel in the form: [minimum execution time (on iteration n of loop, if applicable) / average execution time / maximum execution time (on iteration n of loop, if applicable)] [time spent outside of main loop(s) in kernel time spent in prologue and epilogue of software pipelined loop(s)]

```
*****************************************
*                                       *
  PROFILE AND STATISTICS:
-----------------------------------------
...
1 addAndSum (  IN: 's1' 's1'  OUT: 'temp'  )
  [ 42(0) / 42 / 42(0) / calls: 1 total: 42 ]
  [ nonLoopTime: 0 setupTeardownCycles: 0 ]
```

3. The definition of each input and output stream:

```
    IN: s1 ( 0, 32, , stride, 1, 1 )
    IN: s1 ( 0, 32, , stride, 1, 1 )
    OUT: temp ( 0, 32, , stride, 1, 1 )
...
*                                       *
*****************************************
```

   The definitions of countup and variable size streams are followed by statistics in the form [minimum length (on iteration n of loop, if applicable) / average length / maximum length (on iteration n of loop, if applicable), strip/max end: stripmining (with spilling some percentage) and maximum end], for example:

```
OUT: pixel_rast_str ( 0, 6648, LS, stride, 3, 3 ) [ 368(1) / 388 /
408(0) /  strip/max end: 1224/1224 ]
```

4. A cumulative summary of the execution time, time spent outside of main loop(s) in kernel, and time spent in prologue and epilogue of software pipelined loop(s).

```
*****************************************
*                                       *
  CUMULATIVE STATISTICS:
-----------------------------------------
Total execution time (T):                  84
Total non-loop cycles (NL):                 0  (    0%)
Total loop setup/teardown cycles (ST):      0  (    0%)

          addAndSum: T:       84 NL:        0 ST:        0
                          --------     --------    --------
                              84            0           0

*                                       *
*****************************************
```

5. Stripmining and software pipelining information, see the Equation 9.4 and Equation 9.5 for more information.

```
*****************************************
*                                       *
  STRIPMINING:
Stripmining loops may improve performance.
Stream name (possibly one of many used to access stream data):
  suggested stream data size in records
-----------------------------------------
Stripmining spill: 0%
*                                       *
*****************************************
```

6. Each stream operation and the Imagine operations it compiles into:

```
*****************************************
*                                       *
  COMPILED OPS:
-----------------------------------------
...
-----------------------------------------
 HIGHOP: 1 addAndSum (  IN: 's1' 's1'  OUT: 'temp'  )
```

7. A typical control register write showing the register, high and low data words written (e.g. SRF block and length for SDRs), the stream the register is used for, the issue slot the operation is issued to (indicated by s:), and, if applicable, the index of the stream operation argument used to update the Imagine operation due to a data-dependent stream (indicated by u:)

```
    LOWOP: 4 Write SDR1 = 14 / 32 for 's1' (s: 4)
     RAW:
     WAR:
...
    LOWOP: 9 Run 'addAndSum' at MPC 0 with SDRs( 1 1 2 ) (s: 9)
```

8. The dependency mask for each Imagine operation

```
     RAW: 3 4 5 6 8
     WAR:
...
-----------------------------------------
...
*                                       *
*****************************************
```

9.3.4 Common Questions

The profile information file contains the answers to several common questions:

Q. Were there any problems with my program?

A. See the executive summary for a concise summary of important problems such as spilled streams.

Q. How long (on average) was a certain stream?

A. Look in the profile and statistics section to find the stream operation that produced the stream. Look at the statistics for that stream and look at the middle of the three numbers in brackets

```
OUT: pixel_rast_str ( 0, 6648, LS, stride, 3, 3 )
[ 368(1) / 388 / 408(0)  /  strip/max end: 1224/1224 ]
            ^ average length
```

Q. What size do I make a stream in a stripmined loop?

A. Stripmining is described in the next section. To determine how big to make a stripmined stream, look in the STRIPMINING section under the appropriate loop and find the name of the stream. The first number after the name is the recommend length:

```
foo: 80  (1600 words), currently 40 (800 words)
     ^ the recommended length of foo in records (and in words)
```

9.4 Stripmining

Strip-mining involves processing a large initial input stream in smaller batches so that the intermediate streams produced while processing a batch will all fit in the SRF. Since most stream programs operate on inputs that are larger than the SRF, this optimization is essential for good performance. A typical stream program consists of a series of stream operations that process an initial input to produce a final output. Each stream operation in the series writes one or more outputs that are read as inputs by the next stream operation. Strip-mining applies the series of stream operations to a small portion of the initial input to produce a small portion of the final output, such that the output of every stream operation fits in the SRF. It then applies the series to another small portion of the initial input to produce another small portion of the final output, and so on until all of the initial input has been processed. The size of this portion of the initial input is called the strip size. IStream includes semi-automated stripmining support which, given a stripmined loop, recommends a strip size close to the maximum that will fit in the SRF. The profile information file contains strip mining recommendations unless the -nstrip option is used. IStream presents a set of recommended sizes for all loops in the program, and all streams within each loop. The profile information file presents the stripmining recommendations in the following form:

```
Stripmining spill: 0%
                   ^ percentage of streams that overflow under below
                     recommendations (see below)
LOOP 0
```

```
      ^ the index of the loop, numbered sequentially as ordered within
        the profile

    foo: 80  (1600 words), currently 40 (800 words)
         ^ the recommended length of foo in records (and in words)
    bar1: 160 (6400 words), currently 80 (3200 words)
                          ^ the current length of bar1 in records (and
                            in words)
    bar2: 160 (6400 words), currently 80 (3200 words)
    baz: 40  (200 words), currently 20 (100 words)
    xyz: 800 (16000 words), currently 400 (8000 words)
  ----------------------------------------
```

IStream computes these approximate values by scaling the streams within the loop proportionally to the largest size possible without inducing additional memory traffic. However, not all streams scale proportionately to the strip size, and some are too large to fit in the SRF for any reasonable strip size. To make the strip size estimate more accurate there are two constants which can be binary OR'ed into the data-dependence parameter of a stream definition to control the assumptions made about the scaling of that stream:

- im_strip_none - the size of the stream does not vary with input size
- im_strip_ignore - the stream is very large and should spill for any reasonable strip size, do not attempt to scale it and disregard any spilling

For instance, these constants could be used as follows:

```
im_stream<im_int> myContants = newStreamData(100, im_strip_none);
im_stream<Foo> myMassiveOutput = newStreamData(100000, im_countup |
   im_strip_ignore);
```

In the case of variable length, variable size, or variable position streams, stripmining scales the stream based on its maximum length during the run used to generate the profile. If the stream application detects overflow cases in which such streams exceed the allocated size and handles them explicitly, it may be desirable to generate a larger than normal strip size such that a small percentage of the streams will overflow. Strip size recommendations such that n% of the streams overflow can be generated by running with the option -stripsp <n>. IStream prints this percentage, which defaults to 0, in the profile information file.

For example, consider the following modified version of the example program used in the exercise. It performs a slightly more complicated series of three kernel calls, in a stripmined fashion. It contains a stripmined loop that calls the three kernels on a strip of the input stream to produce a strip of the output stream each iteration. The strip size is given by the constant stripSize, and is initially set to 4800 based on a very rough guess by the developer.

```
#include "idb_streamc.hpp"
#include "test_kc.hpp"

STREAMPROG(testProgram);

const int dataSize = 480000;
```

```
const int stripSize = 4800;

void testProgram(STREAM_SCHEDULER, String args)
{
  im_stream<im_int> NAMED(s1) =
    newStreamData<im_int>(dataSize, im_strip_ignore);
  im_stream<im_int> NAMED(s2) =
    newStreamData<im_int>(dataSize / 2, im_strip_ignore);

  int data[dataSize];
  for (int i = 0; i < dataSize; i++) {
      data[i] = i;
  }
  streamLoadBin((uint32*)data, dataSize, s1);

  im_uc<im_int> uc_sum1 = 0;
  im_uc<im_int> uc_sum2 = 0;
  im_uc<im_int> uc_sum3 = 0;

  profile("test/testProgram") {

    im_stream<im_int> NAMED(s1i) =
    newStreamData<im_int>(stripSize);
    im_stream<im_int> NAMED(temp1) =
      newStreamData<im_int>(stripSize);
    im_stream<im_int> NAMED(temp2) =
      newStreamData<im_int>(stripSize);
    im_stream<im_int> NAMED(s2i) =
      newStreamData<im_int>(stripSize / 2);

    int i = 0;
    int strip;

    // loop over strips of input producing strips of output
    while_VARIABLE(i < dataSize / stripSize) {
      loopIter();

      // copy a strip of input into a fixed size stream, s1i
      strip = i;
      streamCopy(s1(stripSize * strip, stripSize * (strip + 1),
        im_var_incr),
        s1i);

      // add s1i to itself to get temp1
      addAndSum(s1i, s1i,   temp1,   uc_sum1);

      // add temp1 to itself to get temp2
      addAndSum(temp1, temp1, temp2,   uc_sum2);

      // add the even records of temp2 to the odd records of temp2
      // to get a strip of output, s2i
      addAndSum(
        temp2(1, stripSize, im_fixed, im_acc_stride, 2),
        temp2(0, stripSize, im_fixed, im_acc_stride, 2),
        s2i, uc_sum3);

      // copy the strip of output into s2
      streamCopy(s2i,
        s2(stripSize / 2 * strip, stripSize / 2 * (strip + 1),
        im_var_incr));
```

```
      i++;
    }
  }

  streamSaveBin((uint32*)data, s2);
  cout << endl;
  cout << "sum = " <<
    (ucRead(uc_sum1) + ucRead(uc_sum2) + ucRead(uc_sum3)) << endl;
}
```

The profile information file generated by running this application contains the following stripmining suggestions:

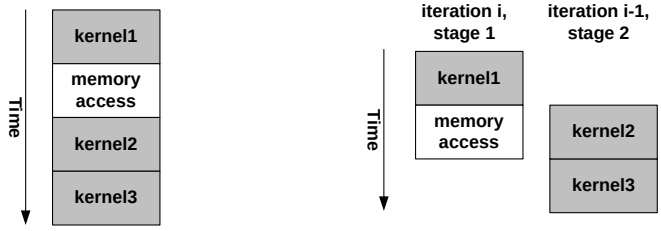
```
*****************************************
*                                       *
  STRIPMINING:
Stripmining loops may improve performance.
Stream name (possibly one of many used to access stream data):
  suggested stream data size in records
-----------------------------------------
Stripmining spill: 0%
LOOP 0
  s1i: 15976 (15976 words), currently 4800 (4800 words)
  temp1: 15976 (15976 words), currently 4800 (4800 words)
  temp2: 15976 (15976 words), currently 4800 (4800 words)
  s2i: 7984 (7984 words), currently 2400 (2400 words)
*                                       *
*****************************************
```

Based on these suggestions, the strip size should be changed to a value near, but not greater than, 15976. These suggestions require interpretation in the context of the specific application. For instance, a strip size of 15000 would be a good choice because it goes evenly into the input size.

9.5 Software pipelining

Software-pipelining involves dividing a loop into stages and overlapping execution of one stage of one iteration with execution of another stage of another iteration. Software-pipelining can be used to hide the memory access time of a sequential memory access, a memory access that must occur between a pair of sequential kernels. A sequential memory access occurs when the result of one kernel is stored to memory and then reloaded using a different access pattern as an input to the next kernel, or when the result of a kernel is used as an index stream for an indexed stream loaded as an input to the next kernel. In either of these cases, the second kernel cannot start immediately after the first kernel, it must wait for the intervening memory load to complete. Software pipelining

can hide the latency for this memory access by overlapping execution of a kernel from another stage with the sequential memory access.



Loop with sequential memory access **Software-pipelined loop**

IStream includes semi-automated software pipelining support which always recommends a software-pipelined order for the stream operations within any loop that contains a sequential memory access. It can also take a source file with chunks of specially tagged code that contains a loop and reorder those chunks of code to produce a new source file containing a software-pipelined version of the loop with a prologue and epilogue. The remainder of this section describes how to use this capability.

The source file must be divided into several tagged sections of code. All tagged sections of code must begin with "SWP_BEGIN(*tagIdentifier*);" and end with "SWP_END(*tagIdentifier*);". Except for the special tags that begin with "*@*", *tagIdentifier* can be any valid identifier so long as the start and end tags match for each section and a unique *tagIdentifier* is used for each section. If and only if a special tag's SWP_BEGIN or SWP_END appears outside of a function, it must be within a comment.

The format of the file must be as follows:

```
// SWP_BEGIN(@headers@);
// all header files
...
// SWP_END(@headers@);

#ifndef SWP

// SWP_BEGIN(@prologue@);
// all code until start of stripmined loop
...
      SWP_END(@prologue@);

      SWP_BEGIN(@loopstart@);
      while_VARIABLE(/* loop condition */) {
        loopIter();
        SWP_END(@loopstart@);

        SWP_BEGIN(tag1);
        // first stream operation (or group of stream ops)
        ...
```

```
        SWP_END(tag1);

        ...

        SWP_BEGIN(tagN);
        // Nth stream operation (or group of stream ops)
        ...
        SWP_END(tagN);

        SWP_BEGIN(@loopincr@);
        // increment loop condition
        ...
        SWP_END(@loopincr@);

        SWP_BEGIN(@loopend@);
        // stream operations that happen at the end of each iteration
        ...
      }
      SWP_END(@loopend@);

    SWP_BEGIN(@epilogue@);
// all code after loop
...
// SWP_END(@epilogue@);
#endif /* SWP */
```

In general, each stream operation within the loop should be placed in its own tag to allow maximum software pipelining flexibility and improve resulting performance.

Software pipelining rearranges tagged sections of code, which can cause variables to become unexpectedly loop carried when reads are moved before assignments, as in:

```
SWP_BEGIN(tag1);
int x = ...;
SWP_END(tag1);
...
SWP_BEGIN(tag9);
... = x + ...;
SWP_BEGIN(tag9);
```

becomes:

```
// stage2
... = x + ...;

// stage1
int x = ...;
```

For this reason, all variables that are not confined to a single tagged section should be declared outside of the loop. Further, it does not consider data dependencies except for stream data, and so a write in one stage of the software pipeline can occur before a read in a later stage. The worst case is:

```
// stage1
x = ...;
```

```
// stage2
... = x + ...;
```

This order will cause the read of *x* in stage 2 to see the value of *x* for the next iteration. To avoid this problem, all variables that are not confined to a single tagged section need to be declared as an array of size “numStages.” Each element in the array in used to track the value in a different stage. All uses of the variable within the loop must be of the form *variable*[stage]. For each such variable, a SWP_ROTATE(*variable*); macro must be inserted directly after the loopIter() statement to rotate the values between stages. An example of such a variable is shown by the following:

```
SWP_BEGIN(@loopstart@);
int myVariable[numStages];
while_VARIABLE(/* loop condition */) {
  loopIter();
  SWP_ROTATE(myVariable);
  SWP_END(@loopstart@);
  ...
  SWP_BEGIN(someTag);
  myVariable[stage]++;
  SWP_END(someTag);
  ...
  SWP_BEGIN(someLaterTag);
  cout << myVariable[stage];
  SWP_END(someLaterTag);
  ...
```

Complex data structures or loop-carried variables may not be used in this way and must be confined to a single tagged section. If, for some reason, this restriction needs to be violated, such as when tagged sections are located in different functions, the two tags can be forced to occur in the same stage by appending a “+” to the start of each tag after the first.

Any nested data-dependent control-flow block (if-statement or loop) within the strip-mined loop must be confined to a single tagged section, with one exception. A special version of if_VARIABLE, if_CONTINUE, can be used to split a single if statement across multiple tags by enclosing each portion of the if statement after the first in a separate if_CONTINUE control flow block as follows:

```
SWP_BEGIN(someTag);
if_VARIABLE(condition) {
  // start doing somthing if condition is true
}
SWP_END(someTag);

SWP_BEGIN(theNextTag);
if_CONTINUE(condition) {
  // do more
}
SWP_END(someTag);

...
```

Successive if_CONTINUE blocks assume that the prior if_CONTINUE blocks have also been executed. This assumption improves performance greatly because it allows the profile compiler to assume that inputs used in later if_CONTINUE blocks came from if_CONTINUE blocks in the same iteration. Otherwise, if multiple if_VARIABLE blocks were used, it would conclude that the inputs could come from a previous iteration and try to keep such streams in the SRF for the entire loop. This conclusion results in increased SRF usage and usually much more memory traffic due to spilled streams.

To generate a new source file from the tagged source file, execute it with the -arp and -pg*N* *<tagged source file 1> ... <tagged sourcefile N>* options, where the first tagged source file is the file containing the loop to be stripmined and source files two through N contain functions called by that source file which can also contain tagged sections of code.

The resulting source file containing a pipelined loop is named by appending “pipe_” to the start of the name of the old source file. Both the original source file and the pipelined source file can be included in the same VC++ project, but only one source file can be used at a time. If the project includes “#define SWP” in a header file common to both source files (or in the global definitions), then the pipelined file is used. If it does not, then the original file is used.

Returning to the example used for stripmining, consider software pipelining the loop to hide the memory access time involved in storing the stream temp2 to memory in order to load its even and odd elements. The following shows the program with software pipelining tags added, and variable *strip* modified for software pipelining:

```
// SWP_BEGIN(@headers@);

#include "idb_streamc.hpp"
#include "test_kc.hpp"

// SWP_END(@headers@);

#ifndef SWP

// SWP_BEGIN(@prologue@);

STREAMPROG(testProgram);

const int dataSize = 100000;
const int stripSize = 4800;

void testProgram(STREAM_SCHEDULER, String args)
{
  im_stream<im_int> NAMED(s1) =
    newStreamData<im_int>(dataSize, im_strip_ignore);
  im_stream<im_int> NAMED(s2) =
    newStreamData<im_int>(dataSize / 2, im_strip_ignore);

  int data[dataSize];
  for (int i = 0; i < dataSize; i++) {
      data[i] = i;
  }
  streamLoadBin((uint32*)data, dataSize, s1);
```

```
im_uc<im_int> uc_sum1 = 0;
im_uc<im_int> uc_sum2 = 0;
im_uc<im_int> uc_sum3 = 0;

profile("test/testProgram") {

  im_stream<im_int> NAMED(s1i) =
  newStreamData<im_int>(stripSize);
  im_stream<im_int> NAMED(temp1) =
    newStreamData<im_int>(stripSize);
  im_stream<im_int> NAMED(temp2) =
    newStreamData<im_int>(stripSize);
  im_stream<im_int> NAMED(s2i) =
    newStreamData<im_int>(stripSize / 2);

  int i = 0;
  int strip[numStages];
  SWP_END(@prologue@);

  SWP_BEGIN(@loopstart@);
  // loop over strips of input producing strips of output
  while_VARIABLE(i < dataSize / stripSize) {
    loopIter();
    SWP_ROTATE(strip);
    SWP_END(@loopstart@);

    SWP_BEGIN(tag1);
    // copy a strip of input into a fixed size stream, s1i
    strip[stage] = i;
    streamCopy(s1(stripSize * strip[stage], stripSize *
      (strip[stage] + 1), im_var_incr),
      s1i);
    SWP_END(tag1);

    SWP_BEGIN(tag2);
    // add s1i to itself to get temp1
    addAndSum(s1i, s1i,   temp1,   uc_sum1);
    SWP_END(tag2);

    SWP_BEGIN(tag3);
    // add temp1 to itself to get temp2
    addAndSum(temp1, temp1, temp2,   uc_sum2);
    SWP_END(tag3);

    SWP_BEGIN(tag4);
    // add the even records of temp2 to the odd records of temp2
    // to get a strip of output, s2i
    addAndSum(
      temp2(1, stripSize, im_fixed, im_acc_stride, 2),
      temp2(0, stripSize, im_fixed, im_acc_stride, 2),
      s2i, uc_sum3);
    SWP_END(tag4);

    SWP_BEGIN(tag5);
    // copy the strip of output into s2
    streamCopy(s2i,
      s2(stripSize / 2 * strip[stage], stripSize / 2(strip[stage] +
      1), im_var_incr));
    SWP_END(tag5);
```

```
      SWP_BEGIN(@loopincr@);
      i++;
      SWP_END(@loopincr@);
      SWP_BEGIN(@loopend@);
    }
    SWP_END(@loopend@);

    SWP_BEGIN(@epilogue@);
  }

  streamSaveBin((uint32*)data, s2);
  cout << endl;
  cout << "sum = " <<
    (ucRead(uc_sum1) + ucRead(uc_sum2) + ucRead(uc_sum3)) << endl;
}

// SWP_END(@epilogue@);

#endif
```

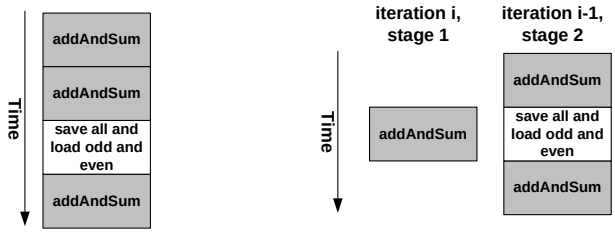
Generating a profile for this program using the -pg option results in the following source file, which contains a software pipelined version of the loop. Note that because the order of kernels changes, the best strip size also changes. For this reason, it is recommended that loops be software-pipelined then have their strip sizes adjusted based on the strip-mining recommendations for the new software-pipelined loop.

```
#include "idb_streamc.hpp"
#include "test_kc.hpp"

//
#ifdef SWP
#undef stage
#define stage stage
#undef numStages
#define numStages 2

STREAMPROG(testProgram);

const int dataSize = 100000;
const int stripSize = 10432;

void testProgram(STREAM_SCHEDULER, String args)
{
  im_stream<im_int> NAMED(s1) =
    newStreamData<im_int>(dataSize, im_strip_ignore);
  im_stream<im_int> NAMED(s2) =
    newStreamData<im_int>(dataSize / 2, im_strip_ignore);

  int data[dataSize];
  for (int i = 0; i < dataSize; i++) {
      data[i] = i;
  }
  streamLoadBin((uint32*)data, dataSize, s1);

  im_uc<im_int> uc_sum1 = 0;
  im_uc<im_int> uc_sum2 = 0;
  im_uc<im_int> uc_sum3 = 0;
```

```
  profile("test/testProgram") {

    im_stream<im_int> NAMED(s1i) =
    newStreamData<im_int>(stripSize);
    im_stream<im_int> NAMED(temp1) =
      newStreamData<im_int>(stripSize);
    im_stream<im_int> NAMED(temp2) =
      newStreamData<im_int>(stripSize);
    im_stream<im_int> NAMED(s2i) =
      newStreamData<im_int>(stripSize / 2);

    int i = 0;
    int strip[numStages];

#define PIPELINE
#ifndef PIPELINE
    // loop over strips of input producing strips of output
    while_VARIABLE(i < dataSize / stripSize) {
      loopIter();
      SWP_ROTATE(strip);

#endif
  setSWP();
//-------------------------------------------------------------
// SWP LOOP PRIME
//-------------------------------------------------------------
  int stage = 0;
      // copy a strip of input into a fixed size stream, s1i
      strip[stage] = i;
      streamCopy(s1(stripSize * strip[stage], stripSize *
(strip[stage] + 1), im_var_incr),
        s1i);

      // add s1i to itself to get temp1
      addAndSum(s1i, s1i,   temp1,   uc_sum1);

#ifdef PIPELINE
      i++;

//-------------------------------------------------------------
// SWP LOOP BODY
//-------------------------------------------------------------
    // loop over strips of input producing strips of output
    while_VARIABLE(i < dataSize / stripSize) {
      loopIter();
      SWP_ROTATE(strip);

    stage = 1;
      // add temp1 to itself to get temp2
      addAndSum(temp1, temp1, temp2,   uc_sum2);

    stage = 0;
      // copy a strip of input into a fixed size stream, s1i
      strip[stage] = i;
      streamCopy(s1(stripSize * strip[stage], stripSize *
        (strip[stage] + 1), im_var_incr),
        s1i);

    stage = 0;
      // add s1i to itself to get temp1
```

```
      addAndSum(s1i, s1i,   temp1,   uc_sum1);

    stage = 1;
      // add the even records of temp2 to the odd records of temp2
      // to get a strip of output, s2i
      addAndSum(
        temp2(1, stripSize, im_fixed, im_acc_stride, 2),
        temp2(0, stripSize, im_fixed, im_acc_stride, 2),
        s2i, uc_sum3);

    stage = 1;
      // copy the strip of output into s2
      streamCopy(s2i,
        s2(stripSize / 2 * strip[stage], stripSize / 2 * (strip[stage]
        + 1), im_var_incr));

    stage = 1;
      i++;

    }

#endif
//-------------------------------------------------------------
// SWP LOOP DRAIN
//-------------------------------------------------------------
  stage = 0;
      // add temp1 to itself to get temp2
      addAndSum(temp1, temp1, temp2,   uc_sum2);

      // add the even records of temp2 to the odd records of temp2
      // to get a strip of output, s2i
      addAndSum(
        temp2(1, stripSize, im_fixed, im_acc_stride, 2),
        temp2(0, stripSize, im_fixed, im_acc_stride, 2),
        s2i, uc_sum3);

      // copy the strip of output into s2
      streamCopy(s2i,
        s2(stripSize / 2 * strip[stage], stripSize / 2 * (strip[stage]
           + 1), im_var_incr));

#ifdef PIPELINE
  if (true) {
#endif
      i++;

    }

  }

  streamSaveBin((uint32*)data, s2);
  cout << endl;
  cout << "sum = " <<
    (ucRead(uc_sum1) + ucRead(uc_sum2) + ucRead(uc_sum3)) << endl;
}

//
#endif
```

This tranformation is summarized by the following figure:



Loop with sequential memory access **Software-pipelined loop**

9.6 Combining stripmining and software-pipelining

An application can contain a loop that is both stripmined and software-pipelined. To generate such a loop, the following procedure should be used:

1. Make sure SWP is not #defined.
2. Run entire program in idebug with a small strip size (one that should have no problem fitting in the SRF) and -pg to generate pipelined source file as described in Section 9.5.
3. Add pipelined source file to the project
4. #define SWP and rebuild
5. Run the program again in idebug.
6. Adjust stream sizes based on stripmining recommendations as described in Section 9.4.
7. Run the program again in isim for cycle-accurate performance numbers.

9.7 Exercise, part 8

1. Generate a profile for the test application by compiling and executing it under IDebug with the command line:

   test/Release/test.exe -m gold8.md -s test/test.sim -idb -arp

2. Execute it under ISim using that profile with the command line:

   test/Release/test.exe -m gold8.md -s test/test.sim -fht

   Do not quit the run, it will be used in the next part of this exercise.

10.0 SchedViz, the Interactive Visualizer

SchedViz is an interactive tool for visualizing schedules of various types. At present, SchedViz is used for visualizing three kinds of schedules: a kernel schedule produced by IScd, a resource (e.g. SRF) allocation produced by IStream, and a high-level trace of an application produced by ISim.

10.1 Basic Usage

SchedViz has a multiple document interface (MDI), the basic functionality of which should be familiar to anyone who has used a Microsoft office application. A schedule visualizer file (.viz) can be opened using the File Menu, or by executing SchedViz.exe from the command line with the file name as the only argument.

Each schedule is displayed in its own document window within the SchedViz container window. A document window contains (top to bottom) a large white picture box showing all or some of the schedule, a narrow text box that displays information about whatever the mouse is currently being held over in the schedule, and set of “tape-player” controls and scroll bar which can be used to replay the scheduling process as described in Section 10.7.

A schedule has one or more resources displayed along the top, horizontal axis, and a time displayed along the left, vertical axis. A rectangle within the axes indicates using a resource over a period of time. Initially, all of a schedule is shown. To zoom in on part of a schedule, position the mouse over a corner of the region of interest, hold down the right mouse button, and drag the mouse to define a rectangular region to zoom in on, then release the right button. To zoom out and see the whole schedule again, right-click on the schedule.

10.2 IScd Kernel Schedules

10.2.1 Operations

When a kernel is scheduled, IScd also generates a SchedViz file with a .viz extension as described in Section 5.5. When viewed using SchedViz, the schedule is shown with the name of each functional unit in an Imagine cluster across the top, and the cycles in the kernel down the side. Small rectangles aligned beneath each functional unit indicate an operation on that functional unit, with latency reflected by the vertical size of the rectangle. An operation rectangle can be white, indicating that it was scheduled as early as possible, light yellow indicating that it was delayed due to functional unit availability, or light blue indicating that it was delayed due to interconnect resource availability.

Holding the mouse over an operation displays information about the operation in the text box below the schedule, with the following format:

```
OP: ( bfly_pair [ RF6[3]  RF7[14] ] ) =
      ^ result  ^ stored in registers
SELECT   (new_stage, hw_const#0, tmp#11 )
^ opcode  ^ operands
b=5,           i=53/53,                s=0/0,
^ basic block  ^ instruction start/end ^ SWP stage start/end
u=0,                 t=272,              l=103
^ func. unit index   ^ scheduling order  ^ KernelC line number
```

10.2.2 Basic Blocks

Basic blocks are separated by horizontal black lines that span the width of the schedule, with downward or upward curling tips representing the start or end of a loop, respectively. A single basic block can be shown and all other basic blocks hidden by holding the mouse over the basic block and pressing Ctrl-V. Pressing Ctrl-V again shows all basic blocks.

If a schedule contains a software pipelined basic block, the operations in each stage of that basic block can be shown by positioning the mouse over the block and pressing Ctrl-X. Each stage is separated by a dashed horizontal line. Operations from one stage are shown within other stages as gray outlines.

10.2.3 Dependencies

A kernel schedule also contains information about the dependencies between operations. Initially, all dependencies are hidden. To show (or hide, if already visible) the dependencies involving an operation, left-click on that operation or hold down the left mouse button and select a region containing the operation. To show (or hide) all dependencies, press Ctrl-D.

A schedule contains three types of “dependencies:”

1. normal dependencies such as read-after-write dependencies, which are shown as solid colored lines with different colors indicating:
   - red = read-after-write or read-forward-write for scratchpad
   - purple = write-after-read or write-forward-read for scratchpad
   - blue = read-after-read
   - green = write-after-write
   - yellow = permanent-state-change-in-same-or-later-stage-as-loop-condition-check-operation
2. special software pipelining dependencies that show which operation must occur within one iteration interval of one another, which are shown as dotted colored lines
3. communications between operations, which are shown as gray lines

By default, only the normal dependencies are visible. To select another type of dependency, press the corresponding number key (1-3) or use the View menu.

Holding the mouse over a dependency displays information about the dependency with the following format:

```
DEP: type=normal,          dist=1,
     ^ type of dependency  ^ min. difference between cycle ops issued
     sa=1, sb=2,                      cp=1
     ^ used for scheduler debugging   ^ cp=1 if on critical path
```

The critical path for a kernel can be displayed by pressing Ctrl-C.

10.3 IStream Resource Allocations

When a profile is compiled, IStream produces a SchedViz file for each type of resource it allocates: the SRF, memory, and various types of control registers (e.g. SDRs). When viewed using SchedViz, a resource allocation has the address space of the resource across the top, and “cycles” down the left side with a stream operations (e.g. kernels) occurring every three “cycles.”

10.3.1 Stream Operations

Stream operations are indicated by dark grey horizontal bars. Holding the mouse above a stream operation displays information about the operation in the following format:

```
OP: 24 xyrast (         IN: 'rast_tri_str'  OUT: 'pixel_rast_str'  )
    ^ index of operation ^input streams            ^outputs streams
b=0, i=73/73, u=0, f=0/1, t=0
^ for internal purposes, unimportant
```

10.3.2 Allocated Resources

An allocated resource is indicated by a dark gray rectangle spanning the allocated portion of the address space horizontally and the duration for which the resource is allocated vertically. Holding the mouse above an allocated resource displays information about the resource in the following format:

```
OP: pos: 820                    size: 198
    ^ location in address space ^size of allocation
b=0, i=69/89, u=0, f=0.800781/0.994141, t=51
^ for internal purposes, unimportant
```

10.3.3 Reads and Writes

Reads from the inputs to a stream operation are indicated by light colored bars just above the intersection of a that stream operation and the portion of the resource that is read. Writes to the outputs of a stream operation are indicated by dark colored bars just below the intersection of that stream operation and the portion of the resource that is written. These bars are of one of three colors:

- green, indicating a normal read or write
- blue, indicating a read or write that requires a memory access (or update to a control register by the host)
- red, indicating a read or write that requires a double buffered memory access.

Holding the mouse above a read or write displays information about the read or write in the following format:

```
OP: pixel_rast_str    ( 0, 6336, LS, stride, 3, 3 )
    ^ name of stream    ^ derivation of stream (coordinates in words)
b=0, i=74/74, u=0, f=0.800781/0.994141, t=0
^ for internal purposes, unimportant
```

For any resource allocation, it is possible to highlight all reads and writes that require memory accesses caused by spilled data by using the Find dialog box described in Section 10.6 and searching for the word “spill.”

10.4 ISim Application Traces

ISim can be used to produce an application trace showing how high-level Imagine resources (such as the clusters) were used during the course of the application. To dump such an application trace, first change the prefix at the ISim command line to the top level of the Imagine chip you want to dump the trace for -- see Section 8.6 for an example of how to do this. Then, the following command will dump a trace:

```
d ./sc “viz [options]”
```

This is assuming that the stream controller is named *sc* -- this name is determined by the machine description file (see ips_developer.pdf for details). If the first argument in quotes is *viz*, a trace is dumped. Several optional arguments can follow the *viz* keyword, and these are listed in Table 10. The default options are ‘`-vfile test.viz +cl +msdat`.’ (Please see isa.pdf for detailed information on the architectural modules and stream operations mentioned in the table below).

TABLE 10. *viz* command options for dumping application traces

Option	Description
-vfile *filename*	dump the trace to a file named *filename* -- relative to path from which ISim was started
-from *cycle*	only dump operations that started after cycle number *cycle*
-to *cycle*	only dump operations that started before cycle number *cycle*
+/-cl	display or don’t display kernel operations
+/-ms	display or don’t display memory operations
+/-sc	display or don’t display stream controller operations
+/-ni	display or don’t display network interface operations
+/-clstr	display or don’t display cluster stream transfers
+/-msdat	display or don’t display memory system data stream transfers
+/-msidx	display or don’t display memory system index stream transfers
+/-mcstr	display or don’t display microntroller stream transfers for loading microcode instructions

TABLE 10. *viz* command options for dumping application traces

Option	Description
+/-nistr	display or don't display netowrk stream transfers
+/-histr	display or don't display host data strean transfers
+all	display all streams

10.4.1 Used Resources

The application trace shows resources across the top and cycles down the side. A rectangle under a particular resource indicates how that resource is used for the span of cycles covered by the height of the rectangle. Holding the mouse above a resource usage rectangle displays information about it in the following format:

```
OP: 132 Run 'assemble_poly_clip' at MPC 830 with SDRs( 4 6 20 21 22 )
        ^ description of the Imagine operation that used the resource
 (s: 7)  b=108,    i=7281836/7282236,    u=0, t=92989
         ^ ignore  ^ cycle range of use  ^ ignore
```

10.4.2 Other

Barriers are shown by horizontal black lines (barriers are instructions that require all previous stream operations to have completed before any subsequent operation can be issued).

10.5 Menu Reference

The following gives brief explanation of what each menu items does. Most menu items have associated hot keys shown in parentheses.

10.5.1 File

- Open (Ctrl-O) — Opens new schedule window
- Reopen (Ctrl-R) — Updates current schedule window
- Print — Prints schedule _as shown_ to printer
- Print to metafile — Prints schedule _as shown_ to .wmf file
- Close — Closes current schedule window
- Exit — Quits SchedViz

10.5.2 View

- Visible dep. types (1...) — Controls which dependency types are visible
- Click toggles only up deps. — Left-click toggles only dependencies to operation
- Click toggles only down deps. — Left-click toggles only dependencies from operation
- Visible blocks (Ctrl-V) — Controls which blocks are visible

- Expanded blocks (Ctrl-X) — Controls which software pipelined blocks are expanded
- Black and white — Displays everything in black and white when selected, cannot be reversed except by reloading
- No node text — Hides text in schedule

10.5.3 Tools

- Find... — Displays find dialog box (see X below) to find and mark operations and/or dependencies
- Find passes — Finds and marks all pass operations
- Find critical path — Finds and marks all critical path dependencies
- Unmark all — Unmarks all pass operations and dependencies
- Show .viz file (F2) — Displays schedule window
- Show .i file (F3) — Displays microassembly file
- Show .uc file (F4) — Displays microcode file

10.5.4 Window

- (displays a list of all windows, select one to view it)

10.5.5 Help

- Help (F1) — Displays help file
- About — Displays version information

10.6 Find Dialog

The find dialog can be used to search for specific text associated with an object in a schedule (the text associated with an object is the text shown when the mouse is held over that object). To use the find dialog, enter the string to search for, select a color to highlight with, select operations and/or dependencies to be highlighted. Find supports limited regular expressions in which "*" matches any series of characters, "?" matches any one character, and “[<*start char*>-<*end char*>]”, e.g. “[A-Z]”, matches a range of characters.

The Find dialog remembers previous finds, which can be selected using the drop-down combo box. In combination with the "Find and Unmark" button this may be used to selectively unmark things.

10.7 Scheduler Replay

The "tape-recorder" buttons and scrollbar at the bottom of the schedule window can be used to play back the scheduling process slowly. Press ">" to start the schedule replay, "||" to pause the schedule replay, and ">|" to show the entire schedule. The scrollbar thumb can also be dragged to show a specific point in the scheduling process.

10.8 Text Editor

The text editor that is to display .i and .uc files is a simple "Notepad"-like text editor. Files are initially displayed in read-only mode, but can be toggled read-write using check-box in lower-left corner.

10.9 Exercise, part 9

1. Save a trace of the application executed in the previous part of the exercise by typing:

   d ./sc “viz -vfile test/test.viz +ms”

2. Use SchedViz to examine the following files:
   test/test_kc.viz, the kernel schedule
   test/test_srf.viz, the SRF allocation for the application
   test/test.viz, the application trace

11.0 Advanced Topics

This section describes how to make a kernel with high register pressure pass register allocation, how to use StreamC with verilog, and other advanced topics.

11.1 Making a Kernel Passing Register Allocation

Imagine has a large number of registers and most kernels should pass register allocation. If a kernel fails register allocation, try scheduling it with the -rf 2 -r 10 command line options. If it still fails, try to determine which of the following is the problem:

11.1.1 Too many temporary variables:

Affects: very large, very parallel kernels

Symptom: large number of registers occupied in multiple arithmetic units AND/OR large number of cc registers occupied

Cause: many temporary values are computed early in the loop and used late in the loop

Solution: divide the loop into multiple parts with barrier operations. This solution will not work for software pipelined loops.

Example: render_sc/renderspan/sort32frag_kc

11.1.2 Loop carried state used in multiple locations:

Affects: kernels with a lot of loop carried state that is used in multiple locations within the loop (often due to "expands")

Symptom: large number of registers occupied in arithmetic units

Cause: loop carried variables are replicated in multiple register files

Solutions:

1. add forced copies (assignments using the %= operator) before multiple uses in loop to make sure that variable is loop carried in one register file only. For example:

```
loop {
...
x %= x;// add this
... = x + ...;
... = x + ...;
...
x %= x;// add this
... = x + ...;
... = x + ...;
...
x = x + 1;
}
```

2. replace "expands" with arrays

Example: mpeg_sc/blocksearch_kc

11.1.3 Loop carried variables concentrated on one unit:

Affects: kernels that use select operations to update a large amount of loop carried state

Symptom: large number of registers occupied in divider

Cause: all select operations on loop carried variables are concentrated on divider since it is often not used for any other purpose

Solution: use the -rf 3 option

Example: render_sc/renderspan/spansprep_kc

11.1.4 Loop-carried variables in a specific register file:

Affects: kernels with lots of loop carried variables that are used by a unique functional unit for non-commutative operations

Symptom: a single register file for a unique functional unit overflows

Solution: added forced copy operations to enable scheduler to store variables in other register files (see example above)

Example: mpeg_sc/rle_kc

11.1.5 Other problems

If none of the above seem to apply, try:

1. try many random seeds w/ the -rf 2 option
2. try using the -rf3 option
3. try 1-2 in combination with using the -cons scheduling option
4. add forced copies for any loop carried variables that are used in multiple places
5. increase the initial iteration interval for a software pipelined loop above the minimum at which it schedules using the -b command line option (this change gradually unpipelines the loop).
6. remove all software pipelining
7. group uses of the same variables together, then insert barriers every few lines
8. store infrequently used variables in arrays
9. split the kernel into multiple kernels

11.2 Using Regression

The Imagine regression suite is intended to provide representative set of applications that test as much of the Imagine toolset as possible. It verifies the functionality of IScd, IStream, IDebug, ISim, and the constituent applications. Tests included in it are gener-

ally targeted tests or representative versions of longer applications (i.e., a full application with a smaller dataset).

The regression suite is intended to be run before any checkin to the SourceSafe repository. This ensures that any changes made do not affect the functionality of other tools and/or applications. It is the responsibility of the person performing the checkin to remove all errors in the regression suite before committing the checkins. Thus, it behooves application developers to add a representative test for their application to the regression suite. This ensures that any tool change made later on will be guaranteed to work with their application.

The regression suite can be run by executing the *regress.bat* command located in the *im_apps* directory. The usage for this command is:

```
regress [1] [8] [opts] [sim [opts]] [scd [opts]]

Options:
   1    - run 1 cluster regress tests
   8    - run 8 cluster regress tests
   8sc  - run 8 cluster streamc regress tests
   8sca - also run 8 cluster streamc regress tests
   opts - run the simulator and scheduler with these options
   sim  - run the simulator with opts as options
   scd  - run the scheduler with opts as options

The default is to run all regress tests. If a number of clusters are
specified, then only those tests are performed. If sim or scd is spec-
ified, then only that test will be performed unless the other is also
specified.
```

This command will run the IScd first on all relevant files, and then run the simulator on the relevant applications. The applications may be run more than once with different command-line options to test different modes of execution in the siimulator. If the regression fails, the batch will pause and wait until the user presses a key. If you want to add a test to the regression suite, please contact *abhishek@cva.stanford.edu*.

11.2.1 StreamC Regression

StreamC regression is executed using the following variation on the regress command line:

```
regress 8sc <options>
regress 8sca <options>
```

Where the “8sc” version runs just StreamC regression and the “8sca” version also runs standard regression. All standard regression options are supported.

Adding a StreamC application to regression is similar to adding a normal application, and consists of the following steps:

1. Add the application the scd.bat so that it can be scheduled with the -a option as described above.
2. Open gold-scd.bat and search for “REM ADD STREAMC HERE”. This line occurs in two places. The first occurrence marks the end of a list of the applications to be

scheduled. Each entry in the list defines the scheduler options for an application, and has the form:

```
set gold8sc_<idx> = -a <application name>
```

Add a new line above the REM comment of this form, replacing <idx> with a unique integer index, and <application name> with the name used by the -a option for the application.

The second occurrence of “REM ADD STREAMC HERE” is at the end of a list of calls to schedule the applications. Each entry calls scd.bat with the options specified above for the application, and takes the form:

```
call ./scd -g8 %scdparams% %gold8sc_<idx>%
```

Add a new line above the REM comment of this form, replacing <idx> with the unique integer index used above.

3. Open gold-sim.bat and search for “REM ADD STREAMC HERE”. This line occurs in one place, at the end of a list of calls to regress_sc_sim, each of the form:

```
call ./regress_sc_sim <app dir> <app exe root> <sim root> <sls root>
  %simparams%
```

Add a new line above the REM comment of this form, replacing <app dir> with the relative path in im_apps of the application directory, <app exe root> with the name of .exe file, minus the .exe extension, <sim root> with the name of standard sim file, minus the .sim extension, and <sls root> with the name of an equivalent sim file that uses the -sls option described in Section 8.0.

11.3 Using Verilog

StreamC works correctly with the verilog version of the simulator, however data can only be loaded and saved for the verilog version in the sim file. To enable this to work, use the -sls command line option, and, for every streamLoadFile, streamSaveFile, and streamCompare file in the application, add a corresponding a read, dump, or vcmp to the sim file that duplicates the functionality in the same order. Reads should read the data to some non-overlapping location with a start address as low as possible, but not lower than 0x100000. Dumps and vcmps should specify an address of 0, which will be over-ridden at run time.

For instance, if the stream program contains the following:

```
streamLoadFile("streamctest/a.txt", "txt", "", a);
streamLoadFile("streamctest/b.txt", "txt", "", b);
streamLoadFile("streamctest/index.txt", "txt", "", idx);
...
streamCompareFile("streamctest/output_correct.txt", c, 0, "a");
```

Then add the following to the sim file:

```
(add this)
```

```
read txt ./ms/data "streamctest/a.txt" 0x100000
read txt ./ms/data "streamctest/b.txt" 0x110000
read txt ./ms/data "streamctest/index.txt" 0x120000

(already in the sim file)
run streamctest ../hp " "
go

(add this)
vcmp ./ms/data "streamctest/output_correct.txt" 0 0.0 "a"
```

11.4 Using special hardware units

The run time tools also support the use of special hardware units. These hardware units are exposed to a stream application as special *hardware kernels*. Hardware kernels are called just like normal kernels with stream and microcontroller variable arguments but can contain arbitrary functionality. Implementing a new hardware kernel involves two parts: defining the kernel and modifying IStream and ISim for cycle-accurate simulation of the new hardware.

11.4.1 Defining a hardware kernel

The kernel is defined just like any other kernel, with two major exceptions. First, the KERNELDEF statement in the _kc.cpp does not contain a .uc file name, since the kernel is implemented directly in hardware. Instead, information about the hardware kernel is specified directly in a string passed in place of the file name. This string takes the form of:

“hw <argument 1 type> <argument 1 resource index> ... <argument n type> <argument n resource index> hwargs <special arguments>”

Where the argument types and resource indices correspond to the arguments to the hardware kernel (in order). Argument type can be “i” for and input stream, “o” for an output stream, or “u”, “r”, or “w”, for a microcontroller variable that is written and read in the kernel, only read, or only written, respectively. Argument index is the physical stream buffer or internal register of the special hardware unit the argument maps to.

The remainder of the string after “hwargs” is stored in the kernel definition data structure (see developer documentation). The string is arbitrary, it can be used to convey whatever information the hardware kernel implementer desires.

Second, the contents of the kernel can be arbitrary code to model the functionality of the hardware; it does not have to be KernelC. Persistent state can be supported through the use of static variables.

The following is an example of a hardware kernel that implements a special hardware unit that generates streams of constants:

```
#include "idb_kernelc.hpp"
#include "test_kc.hpp"
#include "idb_kernelc2.hpp"
```

```
// no .uc file name, direct specification of arguments
KERNELDEF(constGen, “hw r 0 r 1 o 0 hwargs none”)

kernel constGen(im_uc<im_int> uc_val,
                im_uc<im_int> uc_len,
                im_ostream<im_int> out)
{
  // could have arbitrary C++ here
  loop_count(uc_len) {
    out << uc_val;
  }
}
```

For use in implementing special memory units, streams also support the im_cacheable flag, which allows for special memory handling.

11.4.2 Modifying IStream

IStream converts a call to the hardware kernel into the same Imagine operations as a normal kernel (except that it never loads microcode), but provides a simple mechanism for directly modifying those operations. The function StreamProfile::modifyOpForHardwareKernel is called for each such operation to make any needed modifications. Similiarly, setting the im_cacheable flag for a stream causes the StreamProfile::modifyOpForCacheableStream function to be called for all MAR writes and memory loads and stores for that stream. Both of these functions are located in sc_extensions.cpp and documented in the source code.

11.4.3 Modifying ISim

ISim must be explicitly modified to handle the operations produced by StreamProfile::modifyOpForHardwareKernel. Such modifications are covered in the ISim developer documentation. (JDO/UJK?)