Preface

Kiwi was a collaborative project between the University of Cambridge Computer Laboratory and Microsoft Research Limited, headed by David Greaves (UoCCL) and Satnam Singh (MRL). From 2013 onwards, the Kiwi compiler was further developed at the Computer Laboratory and using a logic synthesis library called HPR or HPR-L/S.

Kiwi is developing a methodology for algorithm acceleration using parallel programming and the C# language. Specifically, Kiwi consists of a run-time library for hardware FPGA execution of algorithms expressed within C# and a compiler that generates converts .NET bytecode into Verilog RTL for further compilation for FPGA execution. In the future, custom domain-specific front ends that generate .NET bytecode can be used.

Sadly, this manual is currently in a very-rough form. There is as much wrong and outdated information in here as there is correct!

Contents

1 Download and License 5
  1.1 Warranty 5
# Scientific Users’ Guide

## Kiwi Substrate

- 2.1 Console and LCD stdout I/O and LED GPIO
- 2.2 Exception Handler
- 2.3 File I/O
- 2.4 DRAM
- 2.5 Watchpoints and Start/Stop Control
- 2.6 Framestore
- 2.7 Profiling

## Installation and Easy Get Started

## KiwiC Synthesisable Language Subset Limitations and Style Guide

- 3.1 Data Structures with Kiwi
- 3.2 Testing whether I am running on the Workstation or the FPGA
- 3.3 Exception Handling
- 3.4 The ToString() Method
- 3.5 Accessing Simulation Time
- 3.6 Run-time Status Monitoring and Exceptions
- 3.7 Run-time Divide By Zero and similar Exceptions
- 3.8 Run-time Null-pointer Exceptions
- 3.9 Exiting Threads
- 3.10 Unwound loops
- 3.11 More-complex implied state machines
- 3.12 Inner loop unwound while outer loop not unwound
- 3.13 Entry Point With Parameters

## Generate Loop Unwinding: Code Articulation Point

## Libraries

- 5.1 Console.WriteLine and Console.Write
- 5.2 get.ManagedThreadId
- 5.3 System/String/ToCharArray
Introduction

Kiwi is a compiler and library and infrastructure for high-performance scientific computing targeting (amongst other things) FPGAs.

We aim to compile a fairly broad subset of the concurrent C# language subject to some restrictions:

- Works with the Linux/mono infrastructure but should also work on Windows.
- Program can freely instantiate classes but not at run time - a fixed number of instantiation operations must be detectable at compile time.
- Array sizes must all be statically determinable (i.e. at compile time).
- Program can use recursion but the maximum calling depth must be statically determined.
- Stack and heap must have same shape at each run-time iteration of non-unwound loops. In other words, every allocation made in the outer loop of your algorithm must be matched with an equivalent dispose or garbage generation event in the same loop.
- Program can freely create new threads but creation sites statically determined too.

1 Download and License

Kiwi has been given to several academic institutions for early experiments. A usable release is envisioned in March 2015 and this will be downloadable on completion of a web form. The web form will be on this url: http://koo.corpus.cam.ac.uk/kiwic-download.

1.1 Warranty

Neither the authors nor their employers warrant that the Kiwi system is correct, usable or noninfringing. It is an academic prototype. We accept no responsibility for direct or indirect loss or consequential loss to the maximum amount allowable in UK law.
Part I

Scientific Users’ Guide

2 Kiwi Substrate

We use the term substrate to refer to an FPGA board or boards that is loaded with various standard parts of the Kiwi system. The most important substrate facilities are access to DRAM memory, a disk filesystem and a console/debug channel. Basic run/stop/error status output to LEDs via GPIO is also provided.

The substrate is like an operating system on the FPGA. It supports connection to more than one application.

2.1 Console and LCD stdout I/O and LED GPIO

2.2 Exception Handler

Run-time exceptions include divide-by-zero, floating point overflow and null pointer dereference.
2.3 File I/O

The basic dotnet classes for streamreader, streamwriter, textreader and textwriter are provided. Random access using fseek is also supported.

2.4 DRAM

2.5 Watchpoints and Start/Stop Control

2.6 Framestore

Very high bandwidth writes to the framestore are an intrinsic feature of FPGA computing. The framestore can be used for high-performance visualisation or just for a progress indicator - e.g. percentage of the job processed.

2.7 Profiling

Certain basic block visit counts are collected and the results fed back to the performance counters...

Part II

Installation and Easy Get Started

Kiwi is currently not as easy to use as it could be. You can find an ‘addin’ for the monodevelop IDE on the following URL but it is currently not very useful. Currently it is best if you craft a makefile based on one of the examples.

Monodevelop addin: http://www.cl.cam.ac.uk/users/djg11/kiwi/kiwiaddins/KiwiScientificA

The makefile will compile your application and optionally run the application on your workstation under mono or the Windows equivalent.

The makefile will then invoke the Kiwi compiler to generate a Verilog RTL file and combine this with the provided substrate Verilog files for your FPGA target. Finally it will invoke the FPGA tool suite to give a bitstream file to be loaded to the FPGA.

The means for loading to the FPGA is currently highly-platform specific.

3 Get Started (Dotnet on Linux)

Kiwi is normally supplied as a zip file that contains folders called lib, bin, doc and so on. If you want the source for the compiler please ask.

Requirements: You need a working dotnet environment (mono or Windows) on your machine.

KiwiC/HPR is currently internally implemented in F# but you just need a C# compiler to use it.
If F# is not locally installed you will need to manually add at least the FSharpCore.dll added to the KiwiC/distro/lib folder. We do ship one you can move there. Otherwise you may get 'type load' and 'missing entry point' errors.

KiwiC uses the Mono.Cecil front end and hence the Mono.Cecil.dll is required, either installed on the machine or copied to the KiwiC/distro/lib folder.

Place the KiwiC distribution somewhere on your filesystem. Let us call that place PREFIX. To run KiwiC on Linux you must execute the KiwiC shell script

$ $(PREFIX)/bin/KiwiC ... args ...

The shellscript just contains mono $(PREFIX)/lib/KiwiC.exe

Windows users can invoke the KiwiC.exe executable directly.

The arguments to KiwiC should either be portable assembly files (suffix .dll or .exe) or option flags prefixed with a minus sign. Generally you will supply the current design and KiwiC will automatically load the Kiwi libraries it needs.

Two Kiwi libraries are commonly needed:

1. Kiwi.dll - This defines the Kiwi attributes and other material implemented in C# that should be supplied both to C# compilations and to the KiwiC compiler.

2. Kiwic.dll - This defines additional or replacement implementations of standard .NET library functions for use by the KiwiC compiler and must be supplied on the KiwiC command line. Generally, this is not needed for the first stage of a compilation when an application program in C# is converted to a .NET binary (.exe or .dll) where that binary is either going to be run on the workstation (mono/windows) or compiled further by KiwiC. It should be automatically found by KiwiC and so does not need to be named on any command line.

You must manually include the reference to Kiwi.dll in the C# compilation step. For the KiwiC compilation step, KiwiC will automatically search for the above libraries and include them in the compilation and this is equivalent to manually including them on the KiwiC command line.

To disable automatic search or redirect it to specific files, use the command-line flags -kiwic-dll and -kiwi-dll. Set these to the empty string to disable them or set them to a specific location, e.g. -kiwic-dll=/usr/lib/kiwic/mykiwic.dll.

Note that anything specified via the command line can also be specified in an XML recipe file, with the command line taking precedence when specified both ways. Kiwi comes with a standard recipe for accelerating scientific computing. You can modify this to get SystemC output or for privately developed flows based on Kiwi.

Two common flags, always needed in the past, were -root and -vnl. But now the latter is supplied in the default recipe and instead of using the -root command line flag, the Kiwi.HardwareEntryPoint attribute can be attached to one or more static methods in the input program. Both methods of root designation may be used at once: all marked classes and methods are converted to hardware, however specified.

To obtain Verilog RTL output, KiwiC requires a source file name and access to its libraries. So the most basic Makefile is something like
all:
gmcs /target:library tiny.cs /r:$(KLIB0)
$(KIWIC) tiny.exe

# Other useful options until recently: -vnl and -root:
$KIWIC tiny.exe -root "tiny;tiny.Main" -vnl tiny.v

Given that you have a file called tiny.exe to hand, this should result in a file called tiny.v in your current directory.

To generate tiny.exe one can do the following:

```csharp
$ cat > tiny.cs
using System;
using KiwiSystem;
class tiny {
    [Kiwi.HardwareEntryPoint()]
    public static int Main (string []argv) {
        Console.WriteLine("Hello World");
        return 1;
    }
}
$ gmcs tiny.cs # or use mcs the mono C# compiler.
```

Should you need it, KiwiC will write a disassembly of the PE file to obj/ast.cil in the current folder, enabled by recipe or command line flag.

If you do not have the Kiwi.dll library to hand (e.g. input from C++ instead of C#) or have other problems putting a HardwareEntryPoint attribute on a method then using the -root command line flag is a good idea.

If you do not have the Kiwi.dll library to hand (e.g. input from C++ instead of C#) or have other problems putting a HardwareEntryPoint attribute on a method then using the -root command line flag is an alternative.
Part III

KiwiC Synthesisable Language Subset
Limitations and Style Guide

Much work is needed in this section of the manual ...

All recursion must be to a compile-time determinable depth.
The heap and stack must have the same shape at each point of each iteration of every loop this is not unwound at compile time ...
The filesystem with TextReader ...
This chapter will explain the synthesisable subset of C# supported by KiwiC.
Dynamic storage allocation is supported in KiwiC, provided it is called only from constructors or once and for all on the main threads before they enter an infinite loop. If called inside a non-unwound loop, the heap must be the same shape at each point on each iteration. Dispose is not implemented yet.

3.1 Data Structures with Kiwi

To achieve high performance from any computer system the programmer must think about their data structures and have a basic knowledge of cache and DRAM behaviour. Otherwise they will hit memory bandwidth limitations with any algorithm that is not truly CPU bound.

As in most programming languages, C# variables and structures are static or dynamic. Dynamic variables are allocated on the heap or stack. All are converted to static form during compilation using the version 1 Kiwi compiler. Support for truly dynamic variables will be added in a future release.

It is helpful to define the following two terms for pointer variables. Pointers generally point to dynamic data but their pattern of use falls into two classes. We will call a static pointer one whose value is initially set but which is then not changed. A dynamic pointer is manipulated at run time. Some dynamic pointers range over the value null.

Every C# array and object is associated with at least one pointer because all arrays and objects are created using a call to ‘new’. Also, some valuetypes become associated with a pointer, either by being passed-by-reference or by application of the ampersand operator in unsafe code. The KiwiC compiler will ‘subsume’ nearly all static pointers in its front end constant propagation and any remaining static pointers will be trimmed by later stages in the KiwiC compiler or in the vendor-specific FPGA/ASIC tools applied to the output RTL.

KiwiC maps data structures to hardware resources in two stages. In the first stage, every C# form is converted to either scalars of some bit width or 1-D arrays (also known as vectors) of such scalars. In the second stage, physical placement and packing decisions are made as to which vectors and scalars to place in which physical storage resource (flip-flops, unregistered SRAM, registered SRAM, DP SRAM or off-chip in DRAM). The first stage behaviour is influenced mainly by C# programming style. Second stage behaviour is controlled by heuristic rules parameterised by command-line flags.
and recipe file values.

First Stage Processing:

Two-dimensional arrays are a good example to start with. Although there is syntactic sugar in C# for 2-D arrays, with current C# compilers this is just replaced with operations supplied by a library dll. The dotnet runtime and KiwiC support just 1-D arrays called vectors. There are two possible implementations of a 2-D array library: jagged and packed. The packed form subscript is computed using a multiply of the first co-ordinate with the arity of the second co-ordinate and then adding on the second co-ordinate. The jagged form uses a vector of static pointers to vectors that contain the data; the first co-ordinate is the subscript to the pointer vector and the second co-ordinate is the subscript to the selected vector. We use the term jagged to encompass their smooth form where all data vectors are the same length.

KiwiC inlines the subscript computation for a packed array as though the programmer had inlined such an expression in his C# code. Additionally, there is only one vector created. Therefore packed 2-D arrays first become 1-D vectors. However, such vectors are then subject to unpacking in first stage operation. For instance, if all subscripts are constant values, the array is replaced with a set of scalars. Of if the subscripts fall into clearly disjoint regions, the vector is split into multiple, separately-addressed regions. Or if all the subscripts have a common factor or common offset then these are divided and subtracted off respectively. This unpacking into multiple vectors removes structural hazards that would prevent parallelism.

For a jagged array, initially a number of separate vectors are created and a potentially large number of multiplexing expressions (that appear as the ?: construct in Verilog RTL) are created to direct reads to the correct vector. For writes, an equivalent demultiplexor is created to select the correct vector for writing. (The pointer vector is normally static and becomes subsumed, but we will later discuss what happens if the C# code writes to it, making it dynamic.)

Implementation note: if a jagged array is created by allocating a large 1-D array and storing references to offsets in that vector in the pointer array, it is possible to generate a structure that is identical to the packed array. KiwiC happens to detect this pattern and the behaviour would be as per the packed array: however this style of programming is not allowed in safe C#, but could be encountered in unsafe code or dotnet input form, say, C++.

If we create an array of objects do we expect the fields of the objects to be placed in vectors? Yes, certainly if the object pointers are subsumed.

If we take the parfir, there’s one initialise place where empty flags are written from a non-unwound loop and hence with dynamic subscript, but elsewhere they are updated only with constant subscripts and so should be simple scalar flags.

Kiwi on Loop Unwinding: Loop-carried dependencies in data or control form limit the amount of parallelism that can be achieved with unwinding.

The hard cbg algorithm unwinds all loops without event control. The soft algorithm allocates cycles based on greedy or searching strategies based on complexity and structural hazards. Consider 1: Hoisting of exit condition computation, or hoisting of data dependency computation: this should preferably be applied? So the post-dependent tail of each loop can be forked off...
3.2 Testing whether I am running on the Workstation or the FPGA

Call the function `Kiwi.inHardware()` for this purpose. Since this is a compile-time constant, it is useful for removing development and debugging code from the final implementation. KiwiC will ignore code that is inside `if (false) { }` constructs so write `if (!Kiwi.inHardware()) { ... test/... }

```csharp
[KiwiSystem.Kiwi.HprPrimitiveFunction()]
public static bool inHardware()
{
    return false; // This is the returned value when running on the workstation.
    // An alternative overriding implementation is hardcoded inside KiwiC and will re
}
```

3.3 Exception Handling

please explain ...

3.4 The ToString() Method

Kiwi implements a basic version of the ToString method. It will give output that is rather dependent on which version of the compiler is being used, but it is better than nothing.

3.5 Accessing Simulation Time

The Kiwi library declares a static variable called `tnow`. During compilation are replaced with references to the appropriate runtime mechanism for access to the current simulation time. For instance, the following line

```csharp
Console.WriteLine("Start compute CRC of result at {0}\n", Kiwi.tnow);
```

becomes

```text
$display("Start compute CRC of result at %t\n", $time);
```

when output as Verilog RTL.

3.6 Run-time Status Monitoring and Exceptions

The user requires an indication of whether an FPGA card is actively running an application. Nearly all FPGA cards have LED outputs controlled by GPIO pins that are useful for basic status monitoring. Some FPGAs have LCD or VGA framstore outputs that are also relatively easy to use for this purpose. When a remote console is connected, start, successful completion and exception reporting should be done to the remote console.

The sequencer index for each thread can be remotely tested over the console/debug connection.
3.7 Run-time Divide By Zero and similar Exceptions

Kiwi will log the thread identifier and sequencer index for threads that finish or abort.

3.8 Run-time Null-pointer Exceptions

It is possible to get a run-time null pointer exception.

3.9 Exiting Threads

When the main thread of Kiwi program exits the generated code may include a Verilog $finish statement if the flag “-finish=true” is supplied on the command line or in the recipe file, otherwise a new implicit state machine state is created with no successors and the thread sits in that state forever. Hanging forever is the default behaviour for forked threads.

3.10 Unwound loops

For a thread in hard-pause mode that executes loops with no Pause() calls in them will, KiwiC will attempt to unwind all of the work of that loop and perform it in a single run-time clock cycle. (There are some exceptions to this, such as when there are undecidable name aliases in array operations or structural hazards on RAMs but these are flagged as warnings at compile time and run time hardware monitors can also be generated that flag the error).

```csharp
main_unwound_leader()
{
    q = 100;
    for (int d=0; d<16; d++) Console.WriteLine("q={0}", q++);
    while (true) { Kiwi.Pause(); Console.WriteLine("q={0}", q++); }
}
```

The example main_unwound_leader will unwind the first loop at compile time and execute the first 16 print statements in the first clock tick and q will be loaded with 116 on the first clock tick.

3.11 More-complex implied state machines

```csharp
main_complex_state_mc()
{
    q = 1;
    while(true)
    {
        Kiwi.Pause(); q = 2;
        for (int v=0; v<din; v++) { Kiwi.Pause(); q *= v; }
        Kiwi.Pause(); q = 1;
    }
}
```

The example main_complex_state_mc has a loop with run-time iteration count that is not unwound because it contains a Pause call. This is accepted by KiwiC. However, it could not be compiled
without the Pause statement in the inner loop because this loop body is not idempotent. In soft-pause mode the pause call would be automatically added by KiwiC if missing.

### 3.12 Inner loop unwound while outer loop not unwound.

```c
main_inner_unwound()
{
    q = 1;
    while(true)
    {
        Kiwi.Pause(); q = 2;
        for (int v=0; v<10; v++) { q <<= 1; }
        Kiwi.Pause(); q = 1;
    }
}
```

In main_inner_unwound the inner loop will be unwound at compile time because it has constant bounds and no Pause call in its body. (This unwind will be performed in the bevelab recipe stage, not KiwiC front end.)

### 3.13 Entry Point With Parameters

An entry point with formal parameters, such as

```c
[Kiwi.HardwareEntryPoint()]
main_withparam(int x)
{
    ...
}
```

is not currently allowed, although in future it would be reasonable for these to be treated as additional inputs. Please see Kiwi.Remote() for details of how to do I/O to formal parameters of a method.

### 4 Generate Loop Unwinding: Code Articulation Point

The KiwiC front end unwinds certain loops such as those that perform storage allocation and fork threads. The main behavioural elaborate stage of the KiwiC flow also unwinds other loops. Because of the behaviour of the former, the latter operates on a finite-state system and it makes its decisions based on space and performance requirements typical in high-level synthesis flows. Therefore, the loop unwinding performed in the KiwiC front end can be restricted just to loops that perform structural elaboration. These are known as **generate loops** in Verilog and VHDL. It is a typical Kiwi programming style to spawn threads and allocate arrays and other objects in such loops. Such elaboration that allocates new heap items must be done in the KiwiC front end since the rest of the HPR recipe deals only with statically-allocated variables.

Since threads both describe compile-time and run-time behaviour a means is needed to distinguish the two forms of loop. The approach adopted is that every thread in the source code is treated as generally having a lasso shape, consisting of code that is executed exactly once before entering any non-unwound, potentially-infinite loop.
The front-end algorithm used selects an articulation point in the control graph of a thread where all loops before this point have been unwound and all code reachable after that point has its control graph preserved in the program output to the next stage. Figure 2 illustrates the general pattern. The articulation point selected is the first branch target that is the subject of a conditional branch during an interpreted run of the thread or the entry point to the last basic block encountered that does not contain a Kiwi.Pause() call.

The branch will be conditional either because it depends on some run-time input data or because it is after at least one Kiwi.Pause call. The semantics of Kiwi.Pause imply that all code executed after the call are in a new run-time clock cycle. Apparently-conditional branches may be unconditional because of constant propagation during the interpreted run. This is the basis of generate-style loop unwinding in the lasso stem.

Some programming styles require the heap changes shape at run time. A simple example occurs when an array or other object is allocated after the first call to Kiwi.Pause. We have found that programmers quite often write in this style, perhaps not always intentionally, so it is useful if KiwiC supports it.

```c
main_runtime_malloc()
{
   ...
   Kiwi.Pause();
   int [] a = new Int[10];
   for (int i=0; i<10; i++) a[i] = i;
   while (true) { ... }
}
```

Provided the heap allocator internal state is modelled in the same way as other variables, no further special attention is required. In this fragment the heap values are compile-time constants.
main_runtime_dyn_malloc()
{
    ...
Kiwi.Pause();
if (e)
{ int [] a = new Int[10];
    for (int i=0; i<10; i++) a[i] = i;
}
while (true) { ... }
}

If the value of ‘e’ in runtime_dyn_malloc is not a compile-time constant, KiwiC cannot compile this since there would be two possible shapes for the heap on the exit for the if statement. A solution is to call a.Dispose() before exit, but KiwiC currently does not support Dispose calls.

There’s also the matter of saved thread forks ....

Here the outer loop is non-unwound loop yet has a compile-time constant value on each read if the inner loop is unwound

while(true) // not unwound
{ 
    for (int i=0;i<3;i++) foo[i].bar(f);
    ...
}

5 Libraries

5.1 Console.WriteLine and Console.Write

The Write and WriteLine methods are the standard means for printing to the console in C# and Kiwi. They can also print to open file descriptors. They embody printf like functionality using numbered parameters in braces.

Overloads are provided for used with up to four arguments. Beyond this, the C# compiler allocates a heap array, fills this in and passes it to WriteLine, after which it requires garbage collection. This should provide no problem for Kiwi’s algorithm that converts such dynamic use to static use but if there is a problem then please split a large WriteLine into several smaller ones with fewer than five arguments (beyond the format string).

Kiwi will convert console writes to Verilog’s $display $write PLI calls with appropriate munging of the format strings.

One important choice is whether this console output is preserved for the FPGA implementation. By default it is, with the argument strings compiled to hardware and copied character by character over the console port.

Sometimes two other behaviours are selectively wanted:

- Additional (quick/debugging) console display that is only converted to Verilog PLI calls. This will display output during an RTL simulation of the FPGA (e.g. using Modelsim) but will be discarded by the vendor FPGA tools that convert KiwiC output to FPGA bit streams.
To disable all Console.Write and Console.WriteLine output by default from the FPGA console such that these calls behave just like item 1 above.

To achieve item 1, do not call Console.Write or Console.WriteLine. Instead call Kiwi.Write or Kiwi.WriteLine.

To achieve item 2, alter the recipe file or add the following command line argument to KiwiC

-kiwic-fpgaconsole-default=disable

5.2 get_ManagedThreadId

- returns an integer representing the current thread.

5.3 System/String/ToCharArray

- convert a string to an indexable array of 8-bit? chars?

5.4 System.IO.Path.Combine

- join a pair of file name paths - OS-specific. FileStream

5.5 TextWriter

5.6 TextReader

The TestReader ReadLine api is allowed to create garbage under Kiwi provided the outer loop frees or garbages the returned string on every iteration. It must not, for example, store a handle on the returned string in an array.

5.7 FileReader

5.8 FileWriter

5.9 Environment.Exit(1)

5.10 Threads

6 Attributes

The KiwiC compiler understands various .NET assembly language custom attributes that the user has added to the source code. In this section we present the attributes available. These control thinks such as I/O net widths and assertions and to mark up I/O nets and embed assertions that control unwinding.
C# definitions of the attributes can be taken from the file support/Kiwi.cs in the distribution. The Kiwi attributes can be used by referencing their dll during the C# compiler.

```
// C# code

gmcs /target:library mytest.dll /r:Kiwi.dll
```

Many attributes are copied into the resulting .dll file by the gmcs compiler. Other code from such libraries is not copied and must be supplied separately to KiwiC. To do this, list the libraries along with the main executable on the KiwiC command line.

**WARNING:** THE ATTRIBUTE LIST IS CURRENTLY NOT STABLE AND THIS LIST IS NOT COMPLETE. For the most up-to-date listing, see hprls/kiwi/Kiwi.cs.

The C# language provides a mechanism for defining declarative tags, called attributes, that the programmer may place on certain entities in the source code to specify additional information. An attribute is specified by placing the name of the attribute, enclosed in square brackets, in front of the declaration of the entity to which it applies. We present design decisions regarding attributes that allow a C# program to be marked up for synthesis to hardware using the KiwiC compiler that we are developing. This compiler accepts CIL (common intermediate language) output from either the .NET or Mono C# compilers and generates Verilog RTL.

### 6.1 Flag Unreachable Code

```
Kiwi.NeverReached("This code is not reached during KiwiC compilation.");
```

This call can be inserted in user code to create an error if elaborated by KiwiC. If a thread of control that is being expanded by KiwiC encounters this call, it is an error.

### 6.2 Hard and Soft Clock Control

Many net-level hardware protocols are intolerant to clock dilation. In other words, their semantics are defined in terms of the number of clock cycles for which a condition holds. A thread being compiled by KiwiC defaults to soft pause control (or other default set in the recipe or command line), meaning that KiwiC is free to stall the progress of a thread at any point, such as when it needs to use extra clock cycles to overcome structural hazards. These two approaches are incompatible. Therefore, for a region of code where clock cycle allocation is important, KiwiC must be instructed to use hard pause control.

The Kiwi.Pause() primitive may be called without an argument, when it will pause according to the current pause control mode of the calling thread. It may also be called with the explicit argument 'soft' or 'hard'.

The current pause control mode of the current thread can be updated by calling 'Kiwi.SetPauseControl'. When a thread calls Kiwi.SetPauseControl(hardPauseControl) its subsequent actions will not be split over runtime clock cycles except at places where that thread makes explicit calls to Kiwi.Pause() or makes a blocking primitive call.

The default scheduling mode for a thread can be restored by making the thread calls Kiwi.SetPauseControl(autoPauseControl).

A third mode for a thread, called soft, is under development. Kiwi.SetPauseControl(softPauseControl)
Finally, blockb pause control places a clock pause at every basic block and maximal pause control turns every statement into a separately-clocked operation \texttt{Kiwi.SetPauseControl(maximalPauseControl)}.

### 6.3 Loop NoUnroll Manual Control

Put a call to ‘\texttt{Kiwi.NoUnroll(loopvar)}’ in the body of a loop that is NOT to be unrolled by KiwiC. Pass in the loop control variable.

If there is a ‘\texttt{KiwiC.Pause()}’ in the loop, that’s the default anyway, so the addition of a NoUnroll makes no difference.

The number of unwinding steps attempted by the CIL front end can be set with the ‘‘-cil-uwind-budget N’’ command line flag. This is different from the ubudget command line flag used by the FSM/RTL generation phase.

Because a subsume attribute cannot be placed on a local variable in C#, an alternative syntax based on dummy calls to Unroll is provided.

```csharp
public static void Unroll(int a)
{
    // Use these unroll functions to instruct KiwiC to subsume a variable (or variables)
    // during compilation. It should typically be used with loop variables:
    //
    // for (int cpos = 0; cpos < height; cpos++)
    // { Kiwi.Unroll(cpos);
    //   ...
    // }
}
public static void Unroll(int a, int b)
{
    // To subsume annotate two variables at once.
}
public static void Unroll(int a, int b, int c)
{
    // To annotate three variables.
    // To request subsumation of more than three variables note that
    // calling Unroll(v1, v2) is the same as Unroll(v1 + v2). I.e. the
    // support of the expressions passed is flagged to be subsumed in total or
    // at least in the currently enclosing loop.
}
```

### 6.4 Elaborate/Subsume Manual Control

This manual control was used in early versions of KiwiC but has not been needed recently.

KiwiC implements an elaboration decision algorithm. It decides which variables to subsume at compile time and which to elaborate into concrete variables in the output RTL design.

The decisions it made can be examined by grepping for the word ‘decided’ in the obj/h1.log file.

The algorithm sometimes makes the wrong decision. This is being improved on in future releases.

For variables that can take attributes in C# (i.e. not all variables), it can be forced one way or the other by instantiating one of the pair of attributes, Elaborate or Subsume.

For example, to force a variable to be elaborated, use:

```csharp
[Kiwi.Elaborate()]
bool empty = true;
```
Examples of variables that cannot be attributed is the implied index variable used in a `foreach` loop, or the explicit local defined inside a `for` loop using the `for (int i=...;... ; ...) syntax.

The force of an elab can also be made using the `-fecontrol` command line option. For instance, one might put `-fecontrol 'elab=var1;elab=var2';`

### 6.5 Synchronous and/or Asynchronous RAM Mapping

Arrays allocated by the C# code are commonly converted directly into Verilog array definitions that compile to on-chip RAMs using today’s FPGA tools. There are a number of (adjustable) threshold values that select what sort of RAM to target. Very large RAMS are placed off chip by default.

The three thresholds set in the command line or recipe that distinguish between the four memory types are:

- “res2-regfile-threshold”, the threshold in terms of number of locations below which to not instantiate any sort of structural SRAM,
- “res2-combram-threshold”, the threshold in terms of number of locations at which to start instantiating synchronous, latency=1, structural SRAM
- “res2-offchip-threshold” the threshold in terms of number of locations at which to map to external RAM (e.g. ZBT or DRAM or cache).

In addition to comparing sizes against compilation thresholds, the user can add an attribute to instances to force a given technology choice on a per-RAM basis.

The `SynchSRAM(n)` attribute indicates that an array is to be mapped to an onchip RAM type that is not the default for its size. The argument is the number of clock cycles of latency for read.

**TODO:** describe PortsPerThread and so on... these control multi-port RAMS.

### 6.6 Offchip/Outboard Memory Array Mapping

The `OutboardArray` attribute indicates that an array is to be mapped to a region of external memory instead of being allocated a private array inside the current compilation. Very large RAMS are placed off chip in this way by default. It is up to the system architect what sort of memory to attach to the resulting port: it could range from simple large SRAM bank to multiple DRAM banks with caches.

The fullest version of this attribute takes two arguments: a bank name and an offset in that bank.

In general, arrays can be mapped to a specific bank by giving the bank name and leaving out the base address. KiwiC will then allocate the base addresses for each memory to avoid overlaps. If no bank name is given, then a default of `drambank0` is automatically supplied. Therefore, without using any attributes, all large arrays are mapped into consecutive locations of a memory space called `drambank0`.

Overall, these thresholds and attributes map each RAM instance to a specific level in a four-level memory technology hierarchy:

1. unstructured: no read or write busses are generated (the old default, sea-of-gates)
2. combinational read, synchronous write register file (address generated in same cycle as read data consumed)
3. latency of 1 SSRAM (address generated one clock cycle before read data used)
4. external memory interface for offchip ZBT, DRAM, or cached DRAM.

The FPGA tools will generally automatically choose whether to use a register file for a distributed RAM for forms 1 and 2. They will infer BRAM (or Altera equivalent) for 3.

The number of ports is not defined for type 1 (unstructured) and the FPGA tools will typically implement a register file if the number of operations per clock cycle is more than one. This depends on the number of subscription operators in the generated RTL, the number of different address expressions in use and whether the tools can infer disjointness in their use.

For types 2 through 4, the number of ports is decided by KiwiC and it generates that number of read, write and address busses. By default, KiwiC uses one port per clock domain, but this can be influenced in the future with PortsPerThread and ThreadsPerPort attributes.

6.7 DRAM Cache and Controller Interface

The following nets will require connection to the synthesis output when the DRAM is in use with the default, simple, 4P protocol. The KiwiC-generated code should be connected to an externally-provided memory controller that will typically also include some sort of cache. There are no byte write lanes here - the memory is word addressed always.

A suitable behavioural Verilog fragment to connect to them for simulation test purposes is ...

On the FPGA ...

```verilog
output reg drambank0_req,
input drambank0_ack,
output reg drambank0_rwbar,
output reg [31:0] drambank0_wdata,
output reg [31:0] drambank0_addr,
input [31:0] drambank0_rdata,
```

The restructure log file will explain, somewhat cryptically, how the DRAM is being used with a table for each bank. The lines in this report can be decoded with experience: D16 means sixteen bits wide. AX means an array. etc..

Offchip Memory Map

```
<table>
<thead>
<tr>
<th>Resource</th>
<th>Base</th>
<th>Width</th>
<th>Length</th>
<th>Portname</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8US_AX/CC/SOL</td>
<td>0x1312d02</td>
<td>32</td>
<td>0x989680</td>
<td>drambank0</td>
</tr>
<tr>
<td>D16SS_AX/CC/SOL</td>
<td>0x0</td>
<td>32</td>
<td>0x1312d02</td>
<td>drambank0</td>
</tr>
</tbody>
</table>
```

Performance generally needs to be enhanced above this baseline by packing data sensibly into DRAM words. Also, support of multiple in-flight requests is preferable for the highest performance.
6.8 Filesystem Interface

The following nets will require connection to the synthesis output when the Kiwi file system is in use.

For high performance computing applications the filesystem is part of the Kiwi Substrate (along side the DRAM).

output reg KiwiFiles_KiwiRemoteStreamServices_perform_op_req,
input KiwiFiles_KiwiRemoteStreamServices_perform_op_ack,
input [63:0] KiwiFiles_KiwiRemoteStreamServices_perform_op_return,
output reg [63:0] KiwiFiles_KiwiRemoteStreamServices_perform_op_a2,
output reg [31:0] KiwiFiles_KiwiRemoteStreamServices_perform_op_cmd,

A suitable behavioural Verilog fragment to connect to them for simulation test purposes is /kiwi/filesystem/kiwifs_bev.v that provides the basic console and file stat/exists/open/close/read/write calls required by the dotnet Stream and File.IO classes.

On the FPGA ...

6.9 Hardware Server

The $Server attribute indicates that a method and the methods it calls in turn are to be allocated to a separate RTL module that is instantiated once and shared over all calling threads.

6.10 Register Widths and Overflow Wrapping

Integer variables of width 1, 8, 16, 32 and 64 bits are native in C# and CIL but hardware designers frequently use other widths. We support declaration of registers with width up to 64 bits that are not a native width using an ‘HwWidth’ attribute. For example, a five-bit register is defined as follows.


When running the generated C# natively as a software program (as opposed to compiling to hardware), the width attribute is ignored and wrapping behaviour is governed by the underlying type, which in the example is a byte. We took this approach, rather than implementing a genuine implementation of specific-precision arithmetic by overloading every operator, as done in OSCI SystemC [?], because it results in much more efficient simulation, i.e. when the C# program is run natively.

Although differences between simulation and synthesis can arise, we expect static analysis in KiwiC to report the vast majority of differences likely to be encountered in practice. Current development of KiwiC is addressing finding the reachable state space, not only so that these warnings can be generated, but also so that efficient output RTL can be generated, such that tests that always hold (or always fail) in the reachable state space are eliminated from the code.

The following code produces a KiwiC compile-time error because the wrapping behaviour in hardware and software is different.
The cast of the rhs to a byte is needed by normal C# semantics.

Compiling this example gives an error:

```
KiwiC assign wrap error:
(widthclocks_fivebits(storage=8 )+1)&mask(7..0):
assign wrap condition test rw=8, lw=5, sw=8
```

### 6.11 Input and Output Ports

Input and Output Ports can arise and be defined in a number of ways.

Net-level I/O ports are inferred from static variables in top-most class being compiled. These are suitable for GPIO applications such as simple LED displays and push buttons etc.. The following two examples show input and output port declarations, where the input and output have their width specified by the underlying type and by attribute, respectively.

```
[Kiwi.InputPort("serin")]
static bool serialin;

[Kiwi.HwWidth(5)]
[Kiwi.OutputPort("data_out")]
static byte out5;
```

The contents of the string are a friendly name used in output files.

For designers used to the VDHL concept of a bit vector, we also allow arrays of bools to be designated as I/O ports. This can generate more efficient circuits when a lot of bitwise operations are performed on an I/O port.

```
[Kiwi.OutputWordPort(11, 0, "dvi_d")]
public static int[] dvi_d = new bool[12];

[Kiwi.OutputWordPort(11, 0, "dvi_i")]
public static int[] dvi_i = new int[12];
```

Although it makes sense to denote bitwise outputs using booleans, this may require castings, so ints are also allowed, but only the least significant bit will be an I/O port in Verilog output forms.

Currently we are extending the associated kiwi library so that abstract data types can be used as ports, containing a mixture of data and control wires of various directions. Rather than the final direction attribute being added to each individual net of the port, we expect to instantiate the same abstract datatype on both the master and slave sides of the interface and use a master attribute, such as ‘forwards’ or ‘reverse’, to determine the detailed signal directions for the complete instance.

The following examples work

```
// four bit input port
[Kiwi.HwWidth(4)]
[Kiwi.InputPort("")]
static byte din;

// six bit local var
[Kiwi.HwWidth(6)]
static int j = 0;
```
6.12 Clock Domains

You do not need to worry about clock domains for general scientific computing: they are only a concern for hardware interfacing to new devices. KiwiC generates synchronous logic. By default the output circuit has one clock domain and requires just one master clock and reset input. The allocation of work to clock cycles in the generated hardware depends on the current ‘pause mode’ and an unwind budget described in [?] and the user’s call to built-in functions such as ‘Kiwi.Pause’.

Terminal names clock and reset are automatically generated for the default clock domain. To change the default names, or when more than one clock domain is used, the ‘ClockDom’ attribute is used to mark up a method, giving the clock and reset nets to be used for activity generated by the process loop of that method.

A method with one clock domain annotation must not call directly, or indirectly, a method with a differing such annotation.

6.13 Remote

Object-oriented software sends threads between compilation units to perform actions. Synthesisable Verilog and VHDL do not allow threads to be passed between separately compiled circuits: instead, additional I/O ports must be added to each circuit and then wired together at the top level. Accordingly, we mark up methods that are to be called from separate compilations with a remote attribute.

When an implemented or up-called method is marked as ‘Remote’, a protocol is given (or implied) and KiwiC generates additional I/O terminals on the generated RTL that implement a stub for the call. The currently implemented protocol is synchronous (using the current clock domain - TODO explain how to wire up), using a four-phase handshake and a wide bus that carries all of the arguments in parallel. Another bus, of the reverse direction, conveys the result where non-void. Further protocols can be added to the compiler in future, but we would like to instead lift them so they can be specified with assertions in C# itself. The protocol argument can be omitted from the attribute.

A remote-marked method is either an entry point or a stub for the current compilation. This depends on whether it is called from other hardware entry points (roots).
If it is called, then it is treated as a stub and its body is ignored. Call sites will initiate communication on the external nets. The directions of the external nets is such as to send arguments and receive results (if any).

If it is not called from within the current compilation, then it is treated as a remote-callable entity. The directions of the external nets is such as to receive arguments and return results (if any).

### 6.14 Assertions - Pragmas

Sometimes it is convenient to generate compile time errors or warnings. Typically you might want to direct flow of control differently using the function `Kiwi.inHardware()` and to abort the compilation if it has gone wrong. Call the function `Kiwi.KPragma(true/false, 'my message')` to generate compile time messages. If the first arg holds, the compilation stops, otherwise this serves as a warning message.

You can make use of `System.Diagnostics.Debug.Assert` within Kiwi code.

You have to recode dynamic arrays as static size. The code below originally inspected the `fileStream.Length` attribute and created a dynamic array. But it had to be modified for Kiwi use as follows:

```csharp
int length = (int)fileStream.Length; // get file length - will be known at runtime
System.Console.WriteLine("DNA file length is {0} bytes.", length);
const int max_length = 1000 * 1000 * 10; // Arrays need to be constant length for
System.Diagnostics.Debug.Assert(length <= max_length, "DNA file length exceeds static
buffer = new byte[max_length]; // create buffer to read the file
int count; // actual number of bytes read
int sum = 0; // total number of bytes read
// read until Read method returns 0 (end of the stream has been reached)
while ((count = fileStream.Read(buffer, sum, length - sum)) > 0)
{
    sum += count; // sum is a buffer offset for next reading
}
System.Console.WriteLine("All read, length={0}", sum);
```

The C# compiler may/will ignore the Assert calls unless some flag is passed...

### 6.15 Assertions - Temporal Logic

Universal assertions about a design can be expressed with a combination of a predicate method (i.e. one that returns a `bool`) and a temporal logic quantifier embedded in an attribute. For instance, to assert that whenever the following method is called, it will return true, one can put

```csharp
[Kiwi.AssertCTL("AG", "pred1 failed")]
public bool pred1()
{
    return (...);
}
```

where the string `AG` is a computational tree logic (CTL) universal path quantifier and the second argument is a message that can be printed should the assertion be violated. Although the function
'pred1' is not called by any C# code. KiwiC generates an RTL monitor for the condition and Verilog $display statements are executed should the assertion be violated. In order to nest one CTL quantifier in another, the code of the former can simply call the latter’s method. Since this is rather cumbersome for the commonly used AX and EX quantifiers that denote behaviour in the next state, an alternative designation is provided by passing the predicate to a function called ‘Kiwi.next’. A second argument is an optional number of cycles to wait, defaulting to one if not given. Other temporal shorthands are provided by ‘Kiwi.rose’, ‘Kiwi.fell’, ‘Kiwi.prev’, ‘Kiwi.until’ and ‘Kiwi.wuntil’. These all have the same meaning as in PSL.

We are currently exploring the use of assertions to describe the complete protocol of an I/O port. Such a description, when compiled to a monitor, serves as an interface automaton. To automatically synthesise glue logic between I/O ports, the method of [?] can be used, which implements all non-blocking paths through the product of a pair of such interface automata.

Part IV
Expert and Hardware-level User Guide

7 Designing Hardware with Kiwi

The Kiwi system has a hard pause mode, clock domains and net-level I/O facilities for specifying cycle-accurate hardware. This is needed for connecting to existing hardware interfaces like AXI, I2C and LocalLink.

7.1 Input and Output Ports

Input and Output Ports can arise and be defined in a number of ways. Net-level I/O ports are inferred from static variables in top-most class being compiled. These are suitable for GPIO applications such as simple LED displays and push buttons etc. The following two examples show input and output port declarations, where the input and output have their width specified by the underlying type and by attribute, respectively.

```java
[Kiwi.InputPort("serin")]
static bool serialin;

[Kiwi.HwWidth(5)]
[Kiwi.OutputPort("data_out")]
static byte out5;
```

The contents of the string are a friendly name used in output files.

For designers used to the VDHL concept of a bit vector, we also allow arrays of bools to be designated as I/O ports. This can generate more efficient circuits when a lot of bitwise operations are performed on an I/O port.

```java
[Kiwi.OutputWordPort(11, 0, "dvi_d")]
public static int[12] dvi_d = new bool[12];

[Kiwi.OutputWordPort(11, 0, "dvi_i")]
public static int[12] dvi_i = new int[12];
```

Although it makes sense to denote bitwise outputs using bools, this may require castings, so ints...
7.2 Register Widths and Wrapping

Integer variables of width 1, 8, 16, 32 and 64 bits are native in C# and CIL but hardware designers frequently use other widths. We support declaration of registers with width up to 64 bits that are not a native width using an ‘HwWidth’ attribute. For example, a five-bit register is defined as follows.

```
[Kiwi.HwWidth(5)] static byte fivebits;
```

When running the generated C# natively as a software program (as opposed to compiling to hardware), the width attribute is ignored and wrapping behaviour is governed by the underlying type, which in the example is a byte. We took this approach, rather than implementing a genuine implementation of specific-precision arithmetic by overloading every operator, as done in OSCI SystemC [?], because it results in much more efficient simulation, i.e. when the C# program is run natively.

Although differences between simulation and synthesis can arise, we expect static analysis in KiwiC to report the vast majority of differences likely to be encountered in practice. Current development of KiwiC is addressing finding the reachable state space, not only so that these warnings can be generated, but also so that efficient output RTL can be generated, such that tests that always hold (or always fail) in the reachable state space are eliminated from the code.

The following code produces a KiwiC compile-time error because the wrapping behaviour in hardware and software is different.

```
[Kiwi.HwWidth(5)] byte fivebits;
void f()
{
  fivebits = (byte)(fivebits + 1);
}
```

The cast of the rhs to a byte is needed by normal C# semantics.

Compiling this example gives an error:

```
KiwiC assign wrap error:
(widthclocks_fivebits{storage=8 }+1)&mask(7..0):
assign wrap condition test rw=8, lw=5, sw=8
```

The following examples work

```
// four bit input port
[Kiwi.HwWidth(4)] [Kiwi.InputPort("" )] static byte din;
// six bit local var
[Kiwi.HwWidth(6)] static int j = 0;
```

A short-cut form for declaring input and output ports
7.3 How to write state machines...

Kiwi hardware coding styles: how to code combinational, Mealy and Moore systems in hard-pause mode.

7.3.1 Moore Machines

First compare the Moore machines defined by main_pre and main_post:

```java
[Kiwi.Input()] int din;
[Kiwi.Output()] int q;
main_pre()
{
  q = 100;
  while (true) { q -= din; Kiwi.Pause(); }
}
main_post()
{
  q = 100;
  while (true) { Kiwi.Pause(); q -= din; }
}
```

each has some initial reset behaviour followed by an indefinite looping behaviour. Their difference is the contents of q on the first tick: main_pre will subtract din on the first tick whereas main_post does not. In both cases, q is a Moore-style output (i.e. dependent on current state but not on current input).

The shortly-to-be-implemented optimisation in bevelab will make a further change: the run-time program counter will disappear entirely for main_post because the loading of q with its initial value will be done as part of the hardware reset. However, main_pre will still use a state machine to implement its different behaviour on the first clock tick.

7.3.2 Mealy and combinational logic:

Coding Mealy-style logic and purely combinational sub-circuits is not currently supported. Purely combinational logic could possibly be inferred from an unguarded infinite loop, such as main_comb

```java
main_comb() { while (true) q = (din) ? 42:200; }
```

However, main_comb is not a sanitary program to run under KiwiS since it will hog excessive CPU power.
Mealy-style coding could better be implemented with a new attribute as illustrated in main_mealy where the mel output is a function of both the current state q and current input din.

```cpp
[Kiwi.OutputMealy()] int mel;
main_mealy() { while (true) { q += 1; mel = q+din; Kiwi.Pause(); } }
```

Exploring this further would best be done in conjunction with further development of SystemCsharp to yield a nice overall semantic. TODO perhaps?

### 7.4 State Machines

Explicit state machines can be coded fairly naturally:

```cpp
main_explicit_state_mc()
{
    q = 1;
    while(true)
    {
        Kiwi.Pause();
        switch(q)
        {
            case 1: q = 2; break;
            case 2: q = 3; break;
            case 3: q = 1; break;
        }
    }
}
```

and the position of the single `Kiwi.Pause()` statement before or after the switch statement only alters the reset behaviour, as discussed above.

Implicit state machines can also be used:

```cpp
main_implicit_state_mc()
{
    q = 1;
    while(true)
    {
        Kiwi.Pause(); q = 2;
        Kiwi.Pause(); q = 3;
        Kiwi.Pause(); q = 1;
    }
}
```

Because `main_implicit_state_mc` is a relatively simple example, the KiwiC compiler can be expected to reuse the initial state as the state entered after the third Pause call, but in general the compiler may not always spot that states can be reused.

### 7.5 Clock Domains

A synchronous subsystem designed with kiwi requires a master clock and reset input. The allocation of work to clock cycles in the generated hardware is controlled by an unwind budget described in [?]
and the user’s call to built-in functions such as ‘Kiwi.Pause’. By default, one clock domain is used and default net names clock and reset are automatically generated. To change the default names, or when more than one clock domain is used, the ‘ClockDom’ attribute is used to mark up a method, giving the clock and reset nets to be used for activity generated by the process loop of that method.

```
[Kiwi.ClockDom("clknet1", "resetnet1")]
public static void Work1()
{ while(true) { ... } }
```

A method with one clock domain annotation must not call directly, or indirectly, a method with a differing such annotation.

## Part V

**SystemCsharp**

SystemCsharp follows the design of SystemC and currently there is a very initial version of it in existence. Please see the README.txt in its folder.

The KiwiC compiler can generate SystemCsharp output by using the -csharp-gen=enable command line flag. The default output name is the default name with the suffix .sysc.cs added. The -csharp=filename flag can be used to change the output filename.

## Part VI

**Kiwi Developers’ Guide and Compiler Internal Operation**

### 8 KiwiC Internal Operation

KiwiC is a compiler for the Kiwi project. It aims to produce an RTL design out of a named sub-program of a C# program.

KiwiC does not currently invoke the C# compiler: instead it reads a CIL portable assembly language file (.exe or .dll) generated by a Microsoft or Mono C# compiler.

Figure ?? shows the main flow through the tool as set up with the provided recipe file (KiwiC00.rcp). This is a six stage recipe and the obj folder created by running the tool contains the log files and intermediate forms for each stage. Other output flows and formats can be deployed by changing the recipe. The dotted line shows that using the simvnl command line option the internal simulator (Diosim) can be applied to the RTL after it has been round-tripped through Verilog. For debugging, Diosim can be applied to any HPR machine intermediate form, by varying the recipe. (There’s also a shortcut ‘-conerefine disable -repack disable -verilogen disable’ that will cause diosim to run the original VM generated by the KiwiC front end without conversion to hardware).
Figure 3: The main flow using the default recipe (KiwiC00.rcp) in the KiwiC tool.

---

"Kiwi Scientific Acceleration Manual
Rough Draft User Manual RD1.1"
Figure 4: The internal flow of the KiwiC front-end recipe stage.
The .NET executable is read using the Mono.Cecil front end and combined with some canned system libraries as an abstract syntax tree.

The KiwiC front end (IL elaborate stage) converts the .net AST to the internal form used by the core library, the HPR VM2 machine, which contains imperative code sections and assertions. Code sections can be in series or parallel with each other, using Occam-like SER and PAR blocks.

The front end-stage subsumes a number of variable present in the input source code, including all fixed object pointers.

The VM code emitted by KiwiC front end is a set of parallel ‘dic’ blocks. These are ‘directly indexed code’ arrays of imperative commands and there is one for each user thread. These are placed in parallel using the PAR construct. Each dic array is indexed by a program counter for that thread. There is no stack or dynamic storage allocation. The statements are: assign, conditional branch, exit and calls to certain built-in functions, including hpr_testandset, hpr_printf and hpr_barrier. The expressions occurring in branch conditions, rhs of assignment and function call arguments still use all of the arithmetic and logic operators found in the IL input form. In addition, limited string handling, including a string concat function are handled, so that console output from the IL input is preserved as console output in the generated forms (eg. $display in Verilog RTL).

The conversion to FSM is described in §?? and is common to the h2tool flow.. Actually, that is the old scheduler and a new one is now being used.

Its output is an HPR machine stylised in that there are no program counters and every statement operates in parallel.

The output forms available include Verilog RTL, which we have used for FPGA layout. The stylised output from the FSM generation stage is readily converted to a list of Verilog blocking assignments.

8.1 Background: HPR Library and Orangepath Tool

Orangepath is a refinement framework designed for synthesis of protocols and interfaces in hardware and software forms.

Orangepath H2 represents a system as an hierarchy of abstract machines. Each machine is a database of declarations, executable code and assertions/goals. The goals are assertions about the system behaviour, input directly, or generated from compilation of temporal logic and data conservation rules into automata. Executable code can pass through the system unchanged, but any undriven internal nodes are provided with driver code that ensures the system meets its goals.

As far as possible, all operations are ‘src-to-src’ like operations on HPR virtual machines and the operations are stored in a standard opath command format to be executed by an orangepath recipe (program of commands).

The library is structured as a number of components that operate on a VM to return another VM. The opath mini-language enables a ‘recipe’ to be run that invokes a sequence of library operations in turn. An opath recipe is held in an XML file and the default file is KiwiC00.rcp.

Loops in the recipe can be user to repeat a step until a property holds.

The opath core provides command line handling so that parameters from the receive and the command line are fed to the components. The opath core also processes a few ‘early args’ that must be at the start of the command line. These enable the recipe file to be specified and the logging level to be set.
The Orangepath library has a number of supported input and output formats.
In this manual, we concentrate almost entirely on the .NET CIL input format and the Verilog RTL output format.
Skip this section if you do not feel you need to know about the internal workings of KiwiC.
CIL code is the assembly language used by the mono and .NET projects. The HPR tool can read in CIL assembly code when invoked using the KiwiC command. This assembly code is generated by a large number of third party compilers from various input languages. OLD: NO: All dynamic storage allocation must be made by constructors before the entrypoint method is called. There are numerous other limitations on how the CIL code must be structured.
This stage reads CIL bytecode and generates VM code for the HPR virtual machine. The CIL bytecode is parsed to an AST by a bison parser.
A variable is a static or dynamic object field, a top-level method formal, a local variable, or a stack location. For each variable we decide whether to subsume it at the CIL processing stage. If not subsumed, it appears in the abstract VM code that is fed to the next stage (where it may then get subsumed for other reasons). Variables that are subsumed in this way tend to be object and array handles. Such variables must contain compile-time constant values throughout the execution of the output code.
We perform a symbolic execution of each thread at the CIL basic block level and emit VM code for each block. CIL label names that are branch destinations define the basic block boundaries and these appear verbatim in the emitted VM code.
Although CIL bytecode defines a stack machine, no stack operations are emitted from the CIL processing stage. Stack operations within a basic block are symbolically expanded and values on the stack at basic block boundaries are stored and loaded into stack variables on exit and entry to each basic block. The stack variables are frequently subsumed, but can appear in the VM code and hence, from time-to-time, in the output RTL.
A -root command line flag enables the user to select a number of methods or classes for compilation. The argument is a list of hierarchic names, separated by semicolons. Other items present in the CIL input code are ignored, unless called from the root items.
Where a class is selected as the root, its contents are converted to an RTL module with IO terminals consisting of various resets and clocks that are marked up in the CIL with custom attributes (see later, to be written). The constructors of the class are interpreted at compile time and all assignments made by these constructors are interpreted as initial values for the RTL variables. Where the values are not further changed at run time, the variables turn into compile-time constants and disappear from the object code.
Where a class is selected as a root, all of the methods in that class will be compiled as separate entry points and it is not normally appropriate for one to call another: calls should generally be to methods of other classes.
Where a method is given as a root component, its parameters are added to the formal parameter list of the RTL module created. Where the method code has a preamble before entering an infinite loop, the actions of the preamble are treated in the same way as constructors of a class, viz. interpreted at compile-time to give initial or reset values to variables. Where a method exits and returns a non-void value, an extra parameter is added to the RTL module formal parameter list.
The VM code can be processed by the HPR tool in many ways, but of interest here is the 'con-
Figure 5: The main flow implemented in the KiwiC tool (same as figure ??).
vert_to rtl’ operation that is activated by the ’-vnl’ command line option. (NB: This is now on by default in the KiwiC00 recipe, disable with -verilog-gen disable).

KiwiC TimesTable.exe -root ‘TimesTable;TimesTable.Main’ -vnl TimesTable.v

More than one portable assembly (CIL/PE) file can be given on the command line and KiwiC will aggregate them. The file name of the last file listed will be used to name the compilation outputs by default (in the absence of other command line flags).

(At some point, KiwiC might be extended to also invoke the C# compiler if given a C# file.)

Part VII

Miscellaneous

9 FAQ

Q. Can I generate a VCD using the built-in simulator, diosim.

Q. Yes, use the "-sim=nnnn" argument to set the number of cycles to simulate for and add "-diosim-vcd=myvcd.vcd" to set the output file name. The "-recipe=recipes/simkcode.rcp" command line flag is also useful for just running the KiwiC front end in a software-like simulation.

Q. Why is the reset input not used?

A. The reset net is disconnected unless you indeed add

   -vnl-resets=synchronous
   or
   -vnl-resets=asynchronous

   or change this xml line in the file /distro/lib/recipes/KiwiC00.rcp

   <defaultsetting> resets none </defaultsetting>

Q. Why does the type of the output result end up as: reg [31:0] FIFO2_result; instead of reg FIFO_result; ?

A. In Verilog, integers are signed and registers are not. You can alter this by adjusting the definition of result. Recent Verilog standards also allow signed registers to be defined.

Q. I thought I would have a go at synthesizing the ... However, the Verilog finish statement gets in the way. Should there really be a finish command in synthesizable Verilog?

A. If the main entry point to the C# program allows its thread to exit then a finish will be put in the
output code. This is indeed not synthesizable. Quite often one wants the program to exit when run
native but not when synthesised. The solution to this is to place the main body of the program in
a subroutine that is called from the Main method (ie the entry point). The same subroutine is also
called from a second method where it is enclosed in an infinite while loop. This second method can
then be named as the root to KiwiC and this will avoid a finish statement in the generated code.

Suppressing the default operation on main thread exit statement can be controlled with a command
line flag -finish set to true or false.

Another solution is to mark up the main body subroutine with the Kiwi.Remote attribute. This places
it in an infinite loop, and adds handshaking wires to start and stop its execution.

Q. Should I be worried about the checking failed comments?

+++ checking failed:
Factorial_fac[15:0]:OUTPUT::Unsigned{init=0, io_output=true, HwWidth=16, storage
=32} := Factorial_fac*FTFT4FactorialCircuit_V_0: assignment may wrap differently
: rhs/w=32, lhs/w=16, store/w=32

[Kiwi.OutputWordPort(15, 0)] static uint fac = 1;

A. These are warnings that the generated RTL will behave differently from the dot net versions if
overflow occurs in the custom bit width fields.

You defined the output port to be a sixteen bit register but used the 'uint' dot net valuetype to model
it in the dll. You are performing an operation on this field that is sensitive to its width. The warning
is that there might be a difference in behaviour if, e.g. you increment this value so that it goes above
56535.

Part VIII

Orangepath Synthesis Engines

The Orangepath project supports various internal synthesis engines. The aim is to include SSMG
but some more simple engines are also provided. The other engines include the FSM generator, the
PSL compiler and the restructurer.

Because all input is converted to the HPR machine and all output is from that internal form it is
sensible to use the HPR library for translation purposes without doing any actual synthesis.

A synthesis engine rewrites one HPR machine as another.

10 A* Live Path Interface Synthesiser

The H2 front end tool allows access to the live path interface synthesiser.

The A* version is described on this web page. http://www.cl.cam.ac.uk/djg11/wwwhpr/gpibpage.html

The follow-on to this work is being undertaken by MJ Nam.
11 Transactor Synthesiser

The transactor synthesiser is described on this link:
http://www.cl.cam.ac.uk/research/srg/hans/hprls/orangepath/transactors

12 Asynchronous Logic Synthesiser

The H1 tool implements an asynchronous logic synthesiser described on this link.

13 SAT-based Logic Synthesiser

The H1 tool implements a SAT-based logic synthesiser described on this link.
http://www.cl.cam.ac.uk/djg11/wwwhpr/dslogic.html
(This synthesiser is currently not part of the main HPR revision control branch.)

14 Bevelab: Synchronous FSM Synthesiser

Bevelab is an HPR plugin that converts HPR forms to RTL form. In the RTL form, assignments are made in parallel. Usually, the input is in DIC form where the DIC contains assignments, conditional gotos, fork/join and leaf calls to HPR library functions.

The resulting RTL is generally ‘synthesisable’ as defined by language standards for Verilog, VHDL and SystemC. Converting to one of those languages is by a subsequent plugin. The output of bevelab is normally passed to the verilog-gen plugin where it is converted to Verilog RTL syntax.

Bevelab take an additional input, from the command line, which is an unwind budget: a number of basic blocks to consider in any loop unwind operation. Where loops are nested or fork in flow of control, the budget is divided amongs the various ways.

The flag generate-nondet-monitors turns on and off the creation of embedded runtime monitors for non-deterministic updates.

The flag preserve-sequencer should be supplied to keep the per-thread vestigal sequencer in RTL output structures. This makes the output code more readable but can make it less compact for synthesis, depending on the capabilites of the FPGA tools to do their own minimisation.

The string -resets synchronous should be passed in to introduce synchronous resets to the generated sequencer logic. This is the default.

The string -resets asynchronous should be passed in to introduce assynchronous resets to the generated sequencer logic.

The string -resets none should be passed in to supress reset logic for FPGA targets. FPGA’s tend to have built-in, dedicated reset wiring.
Bevelab has a number of scheduling algorithms (selectable from recipe of commandline).

14.1 Bevelab: Internal Operation

The central data structure is the pending activation queue, where an activation consists of a program counter name, program counter value and environment mapping variables that have so far been changed to their new (symbolic) values.

The output is a list of finite-state-machine edges that are finally placed inside a single HPR parallel construct. The edges have to forms \((g, v, e)\) \((g, \text{fname}, \text{[args]})\) where the first form assigns \(e\) to \(v\) when \(g\) holds and the second calls function \(\text{fname}\) when \(g\) holds.

Both the pending activation queue and the output list have checkpoint annotations so that edges generated during a failed attempt at a loop unwind can be discarded.

The pending activation list is initialised with the entry points for each thread. Operation removes one activation and symbolically steps it through a basic block of the program code, at which time zero, one or two activations are returned. These are either added to the output list or to the pending activation list. An exit statement terminates the activation and a basic block terminating in a conditional branch returns two activations. A basic block is terminated with a single activation at a blocking native call, such as hpr\_pause. When returned from the symbolic simulator, the activation may be flagged as blocking, in which case it is fed to the output queue. Otherwise, if the unwind budget is not used up the resulting activations are added to the pending queue.
A third queue records successfully processed activations. Activations are discarded and not added to the pending queue if they have already been successfully processed. Checking this requires comparison of symbolic environments. These are kept in a "close to normal form" form so that syntactic equivalence can be used. This list is also subject to rollback.

Operation continues until the pending activation queue is empty. A powerful proof engine for comparing activations would enable this condition to be checked more fully and avoid untermination with a greater number of designs.

15 PSL Synthesiser

The PSL synthesiser converts PSL temporal assertions into FSM-based runtime monitors.

16 Statechart Synthesiser

The Sys-ML statechart synthesiser is built in to the front end of the H2 tool. It must be built in to other front ends that generate HPR VMs.

17 SSMG Synthesiser

SSMG is the main refinement component that converts assertions to executable logic using goal-directed search. The SSMG synthesiser is described in a separate document and is a complete sub-project with respect to HPR.

18 Restructure Synthesiser

The RTL-style machines can be restructured, so that different operations occur in different cycles, with automatic insertion of holding registers to maintain data values that would not be available when needed.

Restructuring is need to avoid structural hazards arising when an ALU or multiplier is not fully-pipeline or when a memory has insufficient ports for the level of concurrent access required.

Part IX

Output Formats

The HPR library contains a number of output code generators. All of these write out a representation of an internal HPR machine. Not all forms of HPR machine can be written out in all output forms, but, where this is not possible, a synthesis engine should be available that can be applied to the internal HPR machine to convert it.
Certaint output formats can encode both an RTL/hardware-style and a software/threaded style. For instance, a C-like input file can be rendered out again in threaded C style, or as a list of non-blocking assignments using the SystemC library.

The following output formats may be created:

1. **RTL Form**: The RTL output is written as a Verilog RTL. One module is created that either contains just the RTL portion of the design, or the RTL and instances of each MPU that is executing software parts of the design.

2. **Netlist Form**: The RTL output is compiled to a structural netlist in Verilog that contains nothing but gate and flip-flop instances.

3. **H2 IMP Form**: The HPR form is output to an IMP file. This has the same syntax as the imperative subset of H2.

4. **SMV form**: The HPR VM is output as an SMV code and the assertions that have not been compiled or refined are output as assertions for SMV to check.

5. **C Form**: The HPR VM is output as C code suitable for third-party compilers. RTL forms may also be output as synthesisable SystemC.

6. **UIA MPU Form**: The IMP imperative language is compiled to IMP assembly language and output as a `.s` file.

7. **IP XACT form**: The structural components are written out as IP XACT definitions and instances.

8. **S-expression form**: The HPR VM is dumped a lisp S-expression to a file.

9. **UIA Machine Code**: The IMP assembly is compiled to machine code for the UIA microcontroller. This is output as Intel Hex and also as a list of Verilog assignments for initialising a memory with this code.

The net-based output architecture is suitable for direct implementation as a custom SoC (system on chip). H2 defines its own microcontroller and we use the term MPU to denote an H2 microcontroller with an associated firmware ROM. The net-based architecture consists of RTL logic and some number of MPUs. However, by requesting that all output is as C code for a single MPU, the net-based output degenerates to a single file of portable C code.

Additional output files include log files and synthesisable and high-level models of the UISA microprocessor that executes IMP machine machine code.

**Part X**

**General Orangepath Facilities**

The Orangepath tool provides facilities for a number of experimental compilers. This part describes the core features, not all of which will be used in every flow.
19 FILES AND DIRECTORIES

When an Orangepath tool is run, it creates a directory in the current directory for temporary files. This is the obj directory. This obj directory contains temporary files used during compilation.

The .plt files are plot files that can be viewed using diogif, either on an X display or converted to .gif files.

The h2log file contains a log of the most recent compilation.

20 Espresso

Espresso is not currently needed for Fsharp implementation of HPR.

The Moscow ML implementation of the Orangepath tool requires espresso to be installed in /usr/local or else the ESPRESSO environment variable to point to the binary. If set to the ASCII string NULL then the optimiser is not used.

The -no-espresso flag can also be used to disable call outs to this optimiser. Internal code may be used instead.

21 Cone Refine

The cone refine optimiser deletes parts of the design that have no observable output. It can be disabled using the flag -cone-refine disable.

22 HPR Command Line Flags

The very first args to an HPR/Orangepath tool are the early args that enable the recipe file to be selected and the logging level to be set.

The first argument to an HPR/Orangepath tool, such as h2comp or kiwic, is a source file name. Everything else that follows is an option. Options are now described in turn.

The HPR/LS logger makes an object directory and writes log files to it.

Flag -verbose turns on a level of console reporting. Certain lines that are written to the obj/log files appear also on the console.

Flag -verbose2 turns on a further level of console reporting. Certain lines that are written to the obj/log files appear also on the console.

Flag -recipe fn.xml sets the file name for the recipe that will be followed.

Flag -loglevel n sets the logging level with 100 being the maximum n that results in the most output.

Flag -give-backtrace prevents interceptions of HPR backtraces and will therefore give a less processed, raw error output from mono.
Flag `-root rootname` specifies the root facet for the current run. A number of items can be listed, separated by semicolons. The ones before the last one are scanned for static and initialisation code whereas the last one is treated as an entry point.

In Kiwi, roots may instead or also be specified using the dot net attribute `Kiwi.Hardware`.

When you want only a single thread to be compiled to hardware, either add a `Kiwi.Hardware` attribute or use a root command line flag. If you have both the result is that two threads are started doing the same operations in parallel. The currently fairly-simplistic implementation of offchip has no locks and is not thread safe, so both threads may do operations on the offchip nets at once.

**NOTE:** Many of the command line flags listed here have a different command line syntax using the Fsharp version of Kiwi. To get their effect one must currently either make manual edits to the recipe xml file (e.g. `kiwici00.rcp`) or else simply list them on the command line using the form `-flagname value`

If the special name `-GLOBALS` is specified as a root, then the outermost scope of the assembly, covering items such as the globals found in the C language, is scanned for variable declarations.

Flag `-preserve-sequencer` structures output code with an explicit case or switch statement for each finite-state machine.

Synthcontrol sequencer=unpacked creates sequencer encodings where the PC ranges directly over the h2 line numbers: good for debugging. Otherwise it defaults to a packed binary coding.

Option `-array-scalarise all` converts all arrays to register files. Other forms allows names to be specifically listed.

See § 4.4.

Resets can be disabled using `-resets none` suitable for FPGA targets with built-in reset resources that do not need to be in the netlist.

Resets default to synchronous but can be made asynchronous with `-resets synchronous`.

See § 4.4.

The `-becontrol` command line option was used to pass additional args into the back end of an HPR run. It accepted a string whose individual items are separated with semicolons.

```
"-subexps=off"
```

The `subexps` flag turns off sub-expression commoning-up in the backend.

```
-rootmodname name
```

Use the `rootmodname` flag to set the output module name in Verilog RTL output files.

```
"-ifshare=on"
"-ifshare=none"
"-ifshare=simple"
```

The default `ifshare` operation is that guards are tally counted and the most frequently used guard expressions are placed outermost in a nested tree of if statements.
The _ifshare_ flag turns off if-block generation in output code. If set to 'none' then every statement has its own 'if' statement around it. If it is set to 'simple' then minimal processing is performed. The default setting is 'on'.

"-dpath=on"
"-dpath=none"
"-dpath=simple"

When dpath=on, with the preserve sequencer options for a thread, a separate 'datapath' engine is split out per threads and shared over all data operations by that thread.

Synthcontrol _cone-refine-keep=a,b,c_ accepts a comma-separated list of identifiers names as an argument and instructs the cone-refine optimiser/trimmer to retain logic that supports those nets.

-xtor mode specifies the generation of TLM transactors and bus monitors. The mode may be initiator, target or monitor.

-render-root rootname specifies the root facet for output from the the current run. If not specified, the root facet is used. This has effect for interface synthesis where the root module is not actually what is wanted as the output from the current run.

-ubudget n specifies a budget number of basic blocks to loop unwind when generating RTL style outputs.

The _finish={true false}_ flag controls what happens when the main thread exits. Supplying this flag causes generated output code to exit to the simulation environment rather than hanging forever. When running under a simulator such as Modelsim or when generating SystemC it is helpful to exit the simulation but certain design compiler and FPGA tools will not accept input code that finishes since there is no gate-level equivalent (no self-destruct gate).

The _re structure_ flag controls mechanisms for overcoming static hazards and moving on-chip RAMs to off chip. Currently the argument is the name of the protocol for off-chip RAMs, which may be BVC1 or HSIMPL3.

### 22.1 Other output formats

The _-sysc_ flag causes the tool to generate SystemC output files.

Header and code files are generated with suffix .cpp and .h. Additional header files are generated for shared interfaces and structures. Generally, to make a design consisting of a number of C++ classes, the tool is run a number of times with different root and sysc command line options.

The _-smv_ flag causes the tool to generate a nuSMV output file.

The _-ucode_ flag causes generation of UIA microprocessor code for the design.

- _-vnl fn.v_ specifies to generate a Verilog model and write it to file fn.v.

- _-gatelib NAME_ requests that the Verilog output is in gate netlist format instead of RTL. The identifier NAME specifies the cell library and is currently ignored: a default CAMHDL cell library is used.

- _-gatelib NAME_ requests that the Verilog output is in gate netlist format. This takes precedence over _-vnl_ that causes RTL output.
22.2 General Command Line Flags

The `-version` flag give tool version and help string.
The `-help` flag give tool version and help string.
The `-opentrace` flag sets the opentrace level: this alters the debugging output but most debugging is in the h2log file anyway.
The `-rwtrace` flag sets the rwtrace level, rather like the `-opentrace` option.

22.3 Simulation Control Command Line Flags

The Orangepath tool contains a built-in simulator called diosim. This can run on the input forms, the post-generation forms and post output generation forms. By default it runs on the latest form generated. Flags to alter this default will be removed.

Only the two Verilog output forms, RTL and gatelevel, support conversion back into HPR machine form for post generation simulation.

`-sim n` specifies to simulate the system using the builtin HPR event-driven simulator for n cycles. The output is written to t.plt for viewing. The `-traces` flag provides a list of net patterns to trace in the simulator.

The `-title title` flag names the diosim plot title.
The `-sim-rtl` flag causes diosim to simulate the results of the generator processor (e.g. compilation to FSM) rather than the input form.
The `-sim-gates` flag causes diosim to simulate the results of compilation to gates (-gatelib is used) rather than the input form.
The `-plot plotfile` flag causes plot file output of the diosim simulation to a named plot file.
The plot file can be viewed under x-windows and/or converted to a gif using the diogif program.

23 Diosim Simulator

The Orangepath system contains its own simulator called diosim. Since the target is output from the compiler as portable code to be fed into third-party C and Verilog compilers, it is not strictly necessary to use the Orangepath simulator. However, the simulator provides a self-contained means of evaluating a generated target without using external tools.

The simulator accepts an hierarchical H2 machine and simulates it.
The simulator will verify all safety assertion rules that contain no temporal logic operators. Other safety and all liveness assertions are ignored.

Non-deterministic choices are made on the basis of a PRBS that the user may seed.
The PRBS is also used for synthetic input generation from plant machines or external inputs. PRBS values used for external inputs are checked against plant safety assertions and rejected if they would violate.

Output is a log and plot file. The plot file is currently in diogif plot format, but a VCD format should
Detailed logging can be found in the obj/log files. If a program prints the string 'diosim:traceon' or 'diosim:traceoff' the level of logging is changed.

If a program prints 'diosim:exit' then diosim will exit a though builtin function hpr_exit() were called.

Old Get Started (MOSCOW ML)

**HPR was implemented in moscow ML but you should now be using the dotnet (fsharp) version. Please now ignore this section.**

These instructions apply to running on koo, but should be understandable enough for self-port to other machines.

The h2tool requires the toolmisc, and h2tool directories from the usr/groups/han CVS hprls tree. Examples and documentation are in the examples and doc directories.

The KiwiC tool also requires the mono directory.

Both the h2tool and the KiwiC tool have separate front-end parser binaries. These must be compiled using the command make in the relevant subdirectory: h2fe or cilfe.

Before use, please set the MOSML and HPRLS shell variables

At the computer lab use:

```bash
export MOSML=/usr/groups/theory/mosml2.01
```

HPRLS should be set to your copy of the HPR L/S library. DJG uses:

```bash
export HPRLS=$HOME/d320/hprls
```

ESPRESSO should point to the unix espresso binary or be set to NULL. Espresso is not currently needed for Fsharp implementation of HPR.

If you get this error:

```bash
Uncaught exception:
Fail: libmunix.so: cannot open shared object file: No such file or directory
while loading C library libmunix.so
```

then the setting of LD_LIBRARY_PATH is not working.

The following env vars should be sufficient for the KiwiC command to be run in any directory:

```bash
export HPRLS=/usr/local/hprls/hprls for current 'stable' release
```

or

```bash
export HPRLS=/home/djg11/d320/hprls for djg latest live version
```

also

```bash
export MOSML=/usr/local/mosml2.01
```
Basic invocation, in any directory where the source .il file resides

```bash
$(HPRLS)/mono/KiwiC_TimesTable.il -root 'TimesTable;TimesTable.Main' -vnl TimesTable.v
```

See also `$(HPRLS)/mono/README`

To compile a .il file from C# using the Kiwi attribute library, the following sequence of commands can be used:

```bash
KLIB=$(PREFIX)/support/Kiwi.dll
KIWIC=$(PREFIX)/bin/KiwiC $(KLIB)
ROOT=-root 'DVI4.DVI_Ports;DVI4.Generator.Generate_Synch_Only'

framestore.exe:frametore.cs
    gmcs frametore.cs -r:$(KLIB)

$(KIWIC) framestore.exe $(ROOT) -vnl verilogoutput.v
```

KiwiC using C++ instead of C#

Visual C++ and gcc4cil will generate dotnet portable assemblies from C++ code.

Using the gcc4cil compiler you should find a binary called "cil32-gcc" in the `<path_to_cross_compiler>/bin` directory. To create a CIL file use this compiler with the `-S` option.

Getting gcc4cil.

1. Get Gcc4Cil from the svn-repository that is mentioned on the Gcc4Cil website (http://www.mono-project.com/Gcc4cil)
   "svn co svn://gcc.gnu.org/svn/gcc/branches/st/cli"

2. As Gcc4Cil wants to compile files for the Mono-platform, you need the Mono-project installed on your system. The easiest way to install it is to use "Linux installer for x86" that can be found under [http://www.mono-project.com/Downloads](http://www.mono-project.com/Downloads). Installation instructions are available under [http://www.mono-project.com/InstallerInstructions](http://www.mono-project.com/InstallerInstructions).

3. It may be possible that you need to install the portable .NET project. During the manual compilation of gcc4cil I got errors, that made me install this project. However I could not find a line in the automatic generated Makefile that has a reference to the p.net path in my home-dir. If you get the impression that you need it, you can find it here: [http://www.gnu.org/software/dotgnu/pnet-install.html](http://www.gnu.org/software/dotgnu/pnet-install.html)

4. Because I did not know that there was a automatic script for this, I did a `<path_to_gcc4cil>/configure using the following options`:
   ```bash
   --prefix=<where it should be installed to>
   --with-mono=<install_dir_of_mono>
   ```
I then did a make bootstrap-lean and installed the following libraries because of compile errors:
- bison-2.3.tar.gz
- glib-2.12.9.tar.gz
- pkg-config-0.22.tar.gz

I think it is likely that you may want to skip this step, as this step DOES NOT generate a compiler for cil but for boring x86 code (what I learned after I did this). However I set up paths to the installed libraries in this step, so I mention it. I do not know for sure if all those paths are needed in the end. As it works for me now, I won't remove them:

```bash
setenv HOST_MONOLIB "/home/petero/mono-1.2.5.1/lib"
setenv HOST_MONOINC "/home/petero/mono-1.2.5.1/include/mono-1.0:/home/petero/"
setenv CIL_AS "/home/petero/p.net/lib:/home/petero/p.net/bin"
```

5. In the directory where you put the gcc4cil source code, you can find a shell script called "cil32-crosstool.sh". Execute this and the crosscompiler for C-to-CIL compilation hopefully now gets compiled.
Index