Kiwi Scientific Acceleration Manual
Rough Draft User Manual (KiwiC Version Alpha 2.08)

©2011-16 DJ Greaves + S Singh
April 14, 2016

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 system 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 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.

Contents

1 Download and License
   1.1 Open Source  7
   1.2 Warranty  7
I  Scientific Users’ Guide  

2  Kiwi Substrate  

2.1  Console and LCD stdout I/O and LED GPIO  

2.2  Exception Handler  

2.3  DRAM  

2.4  Watchpoints and Start/Stop Control  

2.5  Framestore  

2.6  Profiling  

II  Installation and Easy Get Started  

3  Get Started (Mono on Linux)  

III  Kiwi Supported Language Subset Limitations and Style Guide  

4  General CSharp Language Features and Kiwi Coding Style  

4.1  I/O with Kiwi  

4.2  Data Structures with Kiwi 1/2  

4.3  Data Structures with Kiwi 2/2 - more advanced and opaque temporary write up  

4.3.1  First Stage Processing (repack):  

4.4  Dynamic Storage Allocation  

4.5  Pointer Arithmetic  

4.6  Garbage Collection  

4.7  Testing whether I am running on the Workstation or the FPGA  

4.8  Debug.Assert  

4.9  Clone  

4.10  Exception Handling  

4.11  The ToString() Method  

4.12  Accessing Numerical Value of Pointer Variables  

4.13  Accessing Simulation Time  

4.14  Run-time Status Monitoring and Exception Logging  

4.15  Run-time Divide By Zero and similar Exceptions  

4.16  Run-time Null-pointer Exceptions  

4.17  Exiting Threads  

Kiwi Scientific Acceleration Manual  
Rough Draft User Manual (KiwiC Version Alpha 2.08)
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.18 Pause Modes</td>
<td>19</td>
</tr>
<tr>
<td>4.19 Unwound Loops</td>
<td>20</td>
</tr>
<tr>
<td>4.20 More-complex implied state machines</td>
<td>20</td>
</tr>
<tr>
<td>4.21 Inner loop unwound while outer loop not unwound</td>
<td>21</td>
</tr>
<tr>
<td>4.22 Entry Point With Parameters</td>
<td>21</td>
</tr>
<tr>
<td>5 Generate Loop Unwinding: Code Articulation Point</td>
<td>21</td>
</tr>
<tr>
<td>6 Supported Libraries Cross Reference</td>
<td>23</td>
</tr>
<tr>
<td>6.1 Console.WriteLine and Console.Write</td>
<td>23</td>
</tr>
<tr>
<td>6.2 getManagedThreadId</td>
<td>24</td>
</tr>
<tr>
<td>6.3 System/String/ToCharArray</td>
<td>24</td>
</tr>
<tr>
<td>6.4 System.IO.Path.Combine</td>
<td>24</td>
</tr>
<tr>
<td>6.5 TextWriter</td>
<td>24</td>
</tr>
<tr>
<td>6.6 TextReader</td>
<td>24</td>
</tr>
<tr>
<td>6.7 FileReade</td>
<td>25</td>
</tr>
<tr>
<td>6.8 FileWriter</td>
<td>25</td>
</tr>
<tr>
<td>6.9 Threading and Concurrency with Kiwi</td>
<td>25</td>
</tr>
<tr>
<td>6.9.1 Volatile Declarations</td>
<td>26</td>
</tr>
<tr>
<td>7 Kiwi C# Attributes Cross Reference</td>
<td>26</td>
</tr>
<tr>
<td>7.1 Kiwi.Remote() Attribute</td>
<td>27</td>
</tr>
<tr>
<td>7.2 Flag Unreachable Code</td>
<td>28</td>
</tr>
<tr>
<td>7.3 Hard and Soft Clock Control</td>
<td>28</td>
</tr>
<tr>
<td>7.4 End Of Static Elaboration Marker - EndOfElaborate</td>
<td>28</td>
</tr>
<tr>
<td>7.5 Loop NoUnroll Manual Control</td>
<td>29</td>
</tr>
<tr>
<td>7.6 Elaborate/Subsume Manual Control</td>
<td>29</td>
</tr>
<tr>
<td>7.7 Synchronous and/or Asynchronous RAM Mapping</td>
<td>30</td>
</tr>
<tr>
<td>7.8 Register Widths and Overflow Wrapping</td>
<td>30</td>
</tr>
<tr>
<td>7.9 Net-level Input and Output Ports</td>
<td>31</td>
</tr>
<tr>
<td>7.10 Clock Domains</td>
<td>31</td>
</tr>
<tr>
<td>7.11 Remote</td>
<td>32</td>
</tr>
<tr>
<td>7.12 Assertions - Pragmas</td>
<td>32</td>
</tr>
<tr>
<td>7.13 Assertions - Temporal Logic</td>
<td>33</td>
</tr>
<tr>
<td>8 Memories in Kiwi</td>
<td>34</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>28 Espresso</td>
<td>65</td>
</tr>
<tr>
<td>29 Cone Refine</td>
<td>65</td>
</tr>
<tr>
<td>30 HPR Command Line Flags</td>
<td>65</td>
</tr>
<tr>
<td>30.1 Other output formats</td>
<td>67</td>
</tr>
<tr>
<td>30.2 General Command Line Flags</td>
<td>68</td>
</tr>
<tr>
<td>30.3 Simulation Control Command Line Flags</td>
<td>68</td>
</tr>
<tr>
<td>31 Diosim Simulator</td>
<td>68</td>
</tr>
</tbody>
</table>
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 early 2016 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 Open Source ?

Is Kiwi open source? Currently the source code has been shared with a few partners. A nearly full source code release will be made once we have the stable version just mentioned. For one module, we may release the source code only for a low-performance implementation. The module in question would be the ‘restructure’ component of the recipe that trades off static and dynamic scheduling for high-performance memory interfacing and good floating-point ALU utilisation (§26). The source code cannot be exploited commercially without further license.

1.2 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 set of server blades that is/are 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 loaded in FPGA at once (cite farming paper).

2.1 Console and LCD stdout I/O and LED GPIO

2.2 Exception Handler

Run-time exceptions include integer divide-by-zero and null pointer de-reference. Floating point overflow is normally handled by returning IEEE Inf or NaN.
CIL bytecode has overflow trapping versions of the arithmetic operators that raise exceptions. We generate these from C# using checked keyword. In the future KiwiC can trap these overflows as run-time errors.

Convert exceptions for casting a value to an illegal value with respect to the target type range, as raised by the conv.ovf CLR instruction, ... please explain.

2.3 DRAM

DRAM and Caches are described in §8.2.

2.4 Watchpoints and Start/Stop Control

2.5 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.

2.6 Profiling

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

Tick counter ... for now.
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.


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 (Mono 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. An open source release is planned.

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 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.

The Kiwi.HardwareEntryPoint attribute can be attached to one or more static methods in the input program. The control-flow graph beneath such methods is converted to hardware. The command line -root flag is another way of specifying an entry point. KiwiC does not default to using a static Main method.

To obtain Verilog RTL output, KiwiC requires a source file name and access to its libraries. So the most basic Makefile is something like

```make
PREFIX=$(HPRLS)/kiwipro/kiwic/distro
KLIBC=$(PREFIX)/kiwipro/support/Kiwic.dll
KLIB0=$(PREFIX)/kiwipro/support/Kiwi.dll
KIWIC=$(PREFIX)/kiwipro/bin/kiwic

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:
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 `-kiwic-cil-dump=separately` or `-kiwic-cil-dump=combined`.

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

Kiwi Supported Language Subset

Limitations and Style Guide

Kiwi aims to support a very broad subset of the C# language and so be suitable for a wide variety of High-Performance Computing (HPC) applications. However, the user is expected to write in a parallel/concurrent style using threads to exploit the parallelism available in the FPGA hardware. However, conventional high-level synthesis (HLS) benefits should be realised even for a single-threaded program.

This chapter will explain the synthesisable subset of C# supported by KiwiC, but currently much work is needed in this section of the manual ...

In general, 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. In other words, dynamic storage allocation is supported in KiwiC, provided it is called only from constructors or once and for all on the main (lasso stems of) 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.

Dynamic storage regions cannot currently be shared between Kiwi threads. Currently, KiwiC implements different heap spaces for each thread ... really? If so this needs fixing ... TODO ... maybe they are only different AFTER a fork but resources allocated before Thread.Start are ok.

Floating point is being provided at last.

Atomic operations: Kiwi supports the CLR Enter, Exit and Wait calls by mapping them on to the hpr_testandset primitive supported by the rest of the toolchain. *Ed: The rest of this paragraph should be in the 'internal operation' section.* Although RTL target languages, such as Verilog, are highly-concurrent, they do not have native support for mutexes. The bevelab recipe stage correctly supports testandset calls implemented by its own threads, but KiwiC does not use these threads: instead it makes a different HPR virtual machine for each thread and these invoke bevelab once each instead of once and for all with bevelab threads within that invocation. Hence the the testandset primitives disappear inside bevelab. ... TODO explain further.

4 General CSharp Language Features and Kiwi Coding Style

4.1 I/O with Kiwi

Kiwi supports a number of forms of I/O:

- Net-level RTL-style I/O through peeking and poking of static variables that are shared with the outside world is the most basic form of I/O. Please see §7.2.

- Methods can also be designated as remotely-callable. Communication between separately-compiled hardware modules is then analogous to method calls between software components. This is explained in §7.1.
• Console debugging style output. Principally this involves calling `+verbConsole.WriteLine()` and running the KiwiC output on an RTL simulator. On the FPGA blades, output to a logging file is supported over the network. Also, certain real hardware devices on the substrate such as LEDs, LCD panels and framestores have also been run experimentally.

• Network and filesystem I/O. See §

### 4.2 Data Structures with Kiwi 1/2

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 perhaps be added in a future release.

Kiwi does not (currently) support taking the address of local variables or static variables in fields (except when pass by reference is being compiled). All pointers and object handles need to refer to heap-allocated items.

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. (As with all C# variables, such pointers can be declared as static or instance in C# program files — this is orthogonal to the current discussion.)

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 (known as repack §25, every C# form (that did not disappear entirely in the front end) is converted to either scalars of some bit width or 1-D arrays (also known as vectors) of such scalars. In the second stage (known as restructure §26), mapping to physical resource decisions are made as to which vectors and scalars to place in what type of component (flip-flops, unregistered SRAM, registered SRAM, DP SRAM or off-chip in DRAM) and which structural instance thereof to use. The first stage behaviour is influenced mainly by C# programming style. Second stage behaviour is controlled by heuristic rules parametrised by command-line flags and recipe file values.

### 4.3 Data Structures with Kiwi 2/2 - more advanced and opaque temporary write up...

#### 4.3.1 First Stage Processing (repack):

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 other 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 example, 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

4.4 Dynamic Storage Allocation

For statically-allocated object instances, KiwiC packs them into flip-flops, B-RAM or DRAM according to thresholds configured in the recipe or command line. This includes objects and structs allocated on the C# heap before the end of static elaboration.

For dynamically-allocated instances, KiwiC cannot easily tell how much memory may be needed and so defaults to DRAM channel 0 if present. But we can switch manually between B-RAM and
DRAM for dynamic storage allocation using C# attributes.

We make the following interesting observation: Once data structures are placed in DRAM there is no real need to have their number statically determined at compile time: instead they can be truly dynamically allocated at run time (DJ Greaves 2015). Indeed, if an application becomes overly dependent on DRAM data then the FPGA advantage will disappear and a Von Neumann (e.g. x86) implemention will have better performance. But, there remains some good FPGA mid ground where a lot of dynamic store is needed but where the access bandwidth required is not excessive.

Kiwi.HeapManager

Physical memories used for dynamic storage require a freespace manager. We can allocate a Heap-Manager for each physical memory and the user can direct requests to an appropriate instance. Typically there could be one for each separate DRAM bank and one for each separate on-chip B-RAM. Also, arrays with dynamic sizes ...

### 4.5 Pointer Arithmetic

```csharp
handleArith pointer arithmetic
Kiwi.ObjectHandler<T>
```

The object handler provides backdoors to certain unsafe code for pointer arithmetic that are banned even in unsafe C# code. Implementation in CIL assembler would be possible but having hardcoded support in the KiwiC compiler accessed via this object manager is easier.

### 4.6 Garbage Collection

With KiwiC V1, the 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.

KiwiC V2 is implementing a more easy to use, run-time storage allocator, but without garbage collection.

- arrays - Array sizes must all be statically determinable (i.e. at compile time).
- new
- linked list example ...
- field-arrays and spatial locality

### 4.7 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
}

4.8 Debug.Assert

Debug.Assert(bool cond) ....

4.9 Clone

Clone of arrays and objects ....

4.10 Exception Handling

please explain ...

4.11 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.

4.12 Accessing Numerical Value of Pointer Variables

Clearly the addresses used on the FPGA have little relationship when run on the mono VM, but it is possible to display class pointer value on the hardware platform. One method is to use the default ToString method on an object handle. This will generate a Kiwi-specific output.

For example

```
Console.WriteLine(" North test14w line0 : pointer={0}", ha.ToString());
Console.WriteLine(" North test14w line1 : left={0}", ha.left);
```

Might give:

```
North test14w line0 : pointer=Var(test14w/T401/Main/T401/Main/V_0%$star1$/test14w/
dc_cls%30008%, &(CTL_record(test14w/dc_cls,...)), ..., )
North test14w line1 : left=32
```

Ah - this has printed the variable not its value!
4.13 Accessing Simulation Time

The Kiwi library declares a static variable called \texttt{tnow}. During compilation reads of this are replaced with references to the appropriate runtime mechanism for access to the current simulation time. For instance, the following line

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

becomes

\begin{verbatim}
$display("Start compute CRC of result at \%t\n", $time);
\end{verbatim}

when output as Verilog RTL.

The substrate has a tick counter that is instantiated when tnow is used.

4.14 Run-time Status Monitoring and Exception Logging

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 framestore 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.

The KiwiStatusMonitors library provides

4.15 Run-time Divide By Zero and similar Exceptions

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

4.16 Run-time Null-pointer Exceptions

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

4.17 Exiting Threads

For simple simulations of the KiwiC-generated RTL, it is sometimes convenient to have the simulator automatically exit when the program has completed.

When the main thread of Kiwi program exits (return from Main), the generated code may include a Verilog $finish statement if the flag "-kiwic-finish=enable" is supplied on the command line or in the recipe file. The equivalent is generated for C++ output. 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.

For use with a standard execution substrate, having a $finish statement in the generated design makes no sense.
Pauses are inserted at

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Pauses are inserted at</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>auto</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>hard</td>
<td>exactly where pause statements are explicitly included</td>
</tr>
<tr>
<td>2</td>
<td>soft</td>
<td>where needed to meet soft-pause-threshold</td>
</tr>
<tr>
<td>3</td>
<td>maximal</td>
<td>inserted at every semicolon</td>
</tr>
<tr>
<td>4</td>
<td>block</td>
<td>every basic block boundary</td>
</tr>
</tbody>
</table>

Table 1: Kiwi Pause Modes

Environment.Exit(1) can also be invoked within C# to cause the same effect as main thread return.

## 4.18 Pause Modes

The mapping of logic operations to clock cycles is a critical aspect of high-level synthesis tools. KiwiC can take various approaches according to the pause mode selected. Pause modes are listed in Table 1. The number of ALUs and RAM ports available also makes a big difference owing to structural hazards. Fewer resources means more clock cycles needed.

The Kiwi.cs file defines an enumeration for locally changing the pause mode for the next part of a thread’s trajectory.

```csharp
enum PauseControl
{
    autoPauseEnable,
    hardPauseEnable,
    softPauseEnable,
    maximalPauseEnable,
    blockbPauseEnable
}
```

The idea is that you can change it locally within various parts of a thread’s control flow graph by calling `Kiwi.PauseControlSet(mode)` where the mode is a member of the PauseControl enumeration. Also, this can be passed as an argument to a Kiwi.Pause call to set the mode for just that pause. However, dynamic pause mode changing may not work at the moment ... owing to minor bugs.

For example, you can invoke `verb+Kiwi.PauseControlSet(Kiwi.PauseControl.softPauseEnable)+`.

The `bevelab-soft-pause-threshold` parameter is one of the main guiding metrics ...

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 bblock or soft pause control, 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 recipe file `kiwic00.rcp` sets the following as the default pause mode now

```xml
<option> bevelab-bevelab-default-pause-mode bblock </option>
```

This is not suitable for net-level interfaces but does lead to quick compile of scientific code which is what we are targeting at the moment.

For compiling net-level input and output, give `KiwiC -bevelab-bevelab-default-pause-mode=hard` as a command line option to override the recipe.
Maximal and blockb are considered just ‘debug’ modes where pauses are inserted at every semicolon and every basic block boundary respectively.

4.19 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).

TODO: describe the way KiwiC resolves structural hazards or variable-latency if the user has specified hard pause mode. Currently, KiwiC essentially tacitly takes and consumes any further clock cycles it needs to do the work.

```csharp
void 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.

4.20 More-complex implied state machines

```csharp
void 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.
4.21 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.)

4.22 Entry Point With Parameters

A top-level 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.

The `-root` command line flag is an alternative to the HardwareEntryPoint marker. Supplying this information on the command line is compatible with multiple compilation approaches where a given source file needs to be processed in different ways on different compilation runs.

5 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 KiwiPause() call.

The branch will be conditional either because it depends on some run-time input data or because it is after at least one KiwiPause call. The semantics of KiwiPause 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 KiwiPause. 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()
{
...
KiwiPause();
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.
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);
  ...
}
```

6  Supported Libraries Cross Reference

6.1  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).

Argument formats supported are

1. {n} — display arg n in base 10
2. {n:x} — display arg n in base 16

Kiwi will convert console writes to Verilog’s $display and $write PLI calls with appropriate munging of the format strings. These will come out during RTL simulation of the generated design. They can also be rendered on the substrate console during FPGA execution.

On 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
```

### 6.2 get.ManagedThreadId

- returns an integer representing the current thread identifier (tid).

```
int tid = Thread.CurrentThread.ManagedThreadId;
Console.WriteLine("Receiver process started. Tid={0}", tid);
```

// OLD
```
Console.WriteLine("Receiver process started. Tid={0}", System.Threading.ManagedThreadId);
```

### 6.3 System/String/ToCharArray

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

### 6.4 System.IO.Path.Combine

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

### 6.5 TextWriter

### 6.6 TextReader

The TextReader 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.
6.7 FileReader

6.8 FileWriter

6.9 Threading and Concurrency with Kiwi

One novel feature of Kiwi that sets it apart from other HLS systems is its support for concurrency. Threads can be spawned in the static lasso stem but Kiwi does not support thread creation at runtime.

To run a method of the current object on its own thread use code like this:

```csharp
public static void IProc()
{
    while (true) { ... }
}

... 

Thread IProcThread = new Thread(new ThreadStart(IProc));
IProcThread.Start();
```

Or use delegates to pass arguments to a spawned thread running a method of perhaps another object:

```csharp
Thread filterChannel = new Thread(delegate() { ZProc(1, 2, 3); });
filterChannel.Start();
```

Exiting threads can be joined with code like this:

```csharp
... missing ...
```

Mutual exclusion is provided with the lock primitive of C#. Its argument must be the object handle of any instance (not a static class).

The Monitor.Wait and Monitor.PulseAll are supported for interprocess events.

```csharp
lock (this)
{
    while (!emptyflag) { /* Kiwi.NoUnroll(); */ Monitor.Wait(this); }
    datum = v;
    emptyflag = false;
    Monitor.PulseAll(this);
}
```

The NoUnroll directive to KiwiC can decrease compilation time by avoiding unrolling exploration.
6.9.1 Volatile Declarations

Variables that are shared between threads may need to be marked as volatile. The normal semantics are that memory fences are inferred from lock block boundaries and other concurrency primitives such as PulseAll. However, if shared variables are used without such fences they should be declared as volatile. Otherwise a process spinning on a change written by another thread may never see it change.

The C# language does not support volatile declarations of some types. You may get an error such as

```csharp
//tinytest.cs(16,26): error CS0677: 'tinytest.shared': A volatile field cannot be of the type 'ulong'
```

To overcome this, you can try to use the Kiwi-provided custom volatile attribute instead for now. For instance:

```csharp
[Kiwi.Volatile()]
static ulong shared_var;
```

This technique will not stop the C# compiler from optimising away a spin on a shared variable, but the C# compiler may not do a lot of optimisation, based on the idea that backend (jitting) runtimes will implement all required optimisations. Ideally KiwiC works out which variables need to be volatile since all threads sharing a variable are compiled to FPGA at once.

7 Kiwi C# Attributes Cross Reference

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.

```
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 [2]. This compiler accepts CIL (common intermediate language) output from either the .NET or Mono C# compilers and generates Verilog RTL.
7.1 Kiwi.Remote() Attribute

RPC (Remote-Procedure Call) Interface Between Compilations.

Marking up given methods to be remotely callable:

Object-oriented software sends threads between compilation units to perform actions. Synthesizable 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.

```csharp
[Kiwi.Remote("parallel:four-phase")]
public return_type entry_point(int a1, bool a2, ...)
{
...
}
```

When an implemented or up-called method is marked as 'Remote', a protocol is given and kiwic generates additional I/O terminals on the generated RTL that implement a stub for the call. The currently implemented protocol is asynchronous, 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.

KiwiC will generate hardware both for the client and the server as separate RTL files. In more-realistic examples, there will be multiple files, with one being the top-level that contains client calls to some of the others which in turn make client calls to others, with the leaf modules in the design hierarchy being servers only.

One can also envision leaf modules in the design hierarchy making upcalls to parents, but this is not currently implemented in Kiwi.

```csharp
class test10
{
    static int limit = 10;
    static int jvar;

    [Kiwi.Remote("client1-port", "parallel: four-phase")]
    public static int bumper(int delta)
    {
        jvar += delta;
        return jvar;
    }

    [Kiwi.HardwareEntryPoint()]
    public static void Main()
    {
        Console.WriteLine("Test 10 Limit=", limit);
        for (jvar=1;jvar<limit;jvar+=2)
        {
            Console.Write(jvar + " ");
        }
    }
}
```

Kiwi Scientific Acceleration Manual
Rough Draft User Manual (KiwiC Version Alpha 2.08)
7.2 Flag Unreachable Code

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

This call can be inserted in user code to create a compile-time error if elaborated by KiwiC. If a thread of control that is being expanded by KiwiC encounters this call, it is a compile-time error. For flagging invalid run-time problems, please use System.Diagnostics.Debug.Assert within Kiwi code.

7.3 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 Kiwi.SetPauseControl(maximalPauseControl).

7.4 End Of Static Elaboration Marker - EndOfElaborate

public static void EndOfElaborate()
{
    // Every thread compiled by KiwiC has its control flow partitioned
    // between compile time and run time. The division is the end
    // of elaboration point.
    // Although KiwiC will spot the end of elaboration point for itself,
7.5 Loop NoUnroll Manual Control

Put a call to ‘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 ‘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 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.
}
```

7.6 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:
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';

7.7 Synchronous and/or Asynchronous RAM Mapping

See §8

7.8 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.

```
[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 [1], 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:assignment may wrap differently:
(widthclocks_fivebits{storage=8 }+1)&mask(7..0):
assign wrap condition test rw=8, lw=5, sw=8
```
7.9 Net-level 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 bools, 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;
```

A short-cut form for declaring input and output ports

```
[Kiwi.OutputIntPort("")]
public static int result;

[Kiwi.OutputWordPort(31, 0)]
public static int bitvec_result;
```

7.10 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 [2] 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.

```
[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.

### 7.11 Remote

Object-oriented software sends threads between compilation units to perform actions. Synthesizable 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.

```
[Kiwi.Remote("parallel:four-phase")]
public return_type entry_point(int a1, bool a2, ...)
{
    ...
}
```

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).

### 7.12 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 com-
pilation 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.

In KiwiC 1.0 you have to re-code dynamic arrays with static sizes and this is needed for all on-chip arrays in Kiwi 2.0. The code below originally inspected the fileStream Length attribute and created a dynamic array. But it had to be modified for Kiwi 1.0 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 ...

### 7.13 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 \cite{3} can be used, which implements all non-blocking paths through the product of a pair of such interface automata.

8 Memories in Kiwi

Arrays allocated by the C# code must be allocated hardware resources. Small arrays 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. Larger arrays are placed off-chip by default.

Sometimes there are multiple ports to a given memory space/bank for bandwidth reasons. For instance, on the Xilinx Zynq, it is common to use two high-performance AXI bus connections to the same DRAM bank. In addition, there can be multiple memory controllers each with its own channel. We prefer the term channel to the older term bank since bank now refers to an internal bank within a DRAM chip that can have up to one row open in each bank. Kiwi does not currently support multiple channels.

Terminology summary: we use the following hierarchy of terms to describe the off-chip memory architecture: bit, lane, word, row, col, bank, rank, channel.

Explanation: A word is addressed with a binary address. The row, col, bank and rank are all fields in the address. Ordering between col and bank may vary. Channels potentially have disjoint address spaces. Mapping the channel number into the address would eliminate spatial reuse and simply be an extension of the rank. Within the word there are multiple lanes that are separately writable and each lane has some number of bits. In today’s CPUs from Intel and ARM, the lane size is 8 (a byte lane) and the word size is also 8, making it a 64-bit word. In FPGAs, where clock frequencies are lower than DRAM speeds, word sizes of 512 can commonly be used with a correspondingly larger number of lanes.

In this documentation, we use the term ‘off-chip’ to denote resources that are not instantiated by KiwiC and which, instead, are provided by the substrate platform. In reality, the resources might physically be on the same silicon chip as the FPGA programmable logic.

Each array with off-chip status is allocated a base address in one of some number of off-chip memory channels and accessed via one or more off-chip load/store ports.

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, any number of concurrent reads and writes are possible without worry over structural hazard)
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 off-chip ZBT/QBI, DRAM, or cached DRAM.

The number of ports is unlimited for type 1 (register file) and the FPGA tools will typically implement such a register file if the number of operations per clock cycle is more than one. This depends
FPGAs tools support RAMs in four general ways. The four ways provide increasingly better FPGA area use, but become more complex to read and write.

1. **Flip-flop register file**: Each bit of RAM becomes a flip-flop. This does not limit the number of concurrent readers or writers.

2. **Distributed RAM**: The look-up table (LUT) of a typical FPGA is used normally for something like an arbitrary two-output function of five inputs. It is then actually a 32-word RAM of 2-bit words. The can be used as RAM by many FPGAs. It is called distributed or slice RAM.

3. **Block RAM**: As well as I/O, flip-flops and LUTs, all modern FPGAs also provide block RAMs as a first-class programmable resource. Typically these are dual ported and 18 kilobit in size.

4. **Off-chip RAM - SRAM or DRAM**: Rather than storing data on the FPGA, I/O pins are used to connect to external, standard RAM parts.

The FPGA tools will generally automatically choose which of the first three forms in the above list to infer for a given RTL array declaration. They take into account the size and use pattern. Important aspects of the use pattern are whether the output is used in the same clock cycle as the address is generated and how many different and concurrent address patterns are used. The fourth RAM form is not automatically generated by FPGA tools, but HLS tools such as KiwiC will deploy it and the FPGA tools will simply see logic that wiggles pins to operate the RAM or generate AXI transactions destined for a complex memory subsystem.

<table>
<thead>
<tr>
<th>Table 2: RAM forms supported by FPGAs.</th>
</tr>
</thead>
</table>

---

*[Kiwi Scientific Acceleration Manual](#)*  
*Rough Draft User Manual (KiwiC Version Alpha 2.08)*
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.

In the current version of Kiwi, the `res2-loadstore-port-count` (formerly called `res2-no-dram-ports`) recipe setting configures the number of load/store ports available per thread. Also, each thread that makes off-chip loads and stores must have its own port since KiwiC does not automatically instantiate the DRAM (HFAST) arbiters: instead the substrate top-level needs to instantiate the arbiters when KiwiC generates more DRAM ports than physically exist on the FPGA.

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

1. **res2-regfile-threshold**: the number of locations below which to not instantiate any sort of structural SRAM or register file: instead raw flip-flops are used.
2. **res2-combram-threshold**: the threshold in terms of number of locations at which to start instantiating synchronous, latency=1, structural SRAM,
3. **res2-offchip-threshold**: the threshold in terms of number of locations at which to map to an off-chip resource, such as TCM, ZBT or cached DRAM.

In addition to comparing sizes against compilation thresholds, the user can add CSharp attributes 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 on-chip 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 and how the number of external ports is configured.

Kiwi has a scheduler in its restructure phase that runs at compile time to sequence operations on scarce resources such as complex ALUs and memory resources. Kiwi supposedly implements run-time arbitration for resources that are contended between threads, but the reality is currently different. It follows three policies: 1. For ‘on-chip’ RAMs like FPGA B-RAM it allocates one port per thread so, with Xilinx and Altera that support up to two ports only two threads can access an ‘on-chip’ B-RAM. 2. For ALUs it does not share them between threads and starts the ALU budgeting freshly for each thread, just as though the threads had been separately compiled. 3. For ‘off-chip’ RAM like DRAM, it generates one (more are possible via the command line) HFAST port per thread. The user must currently manually instantiate arbiters that mux this collection of ports onto the DRAM banks that are available.

However, Kiwi does not care whether ‘off-chip’ resources are actually off-chip and instead one can use the off-chip technique to multiplex and arbitrate multiple threads onto on-chip resources, such as a large, manually instantiated B-RAM.

### 8.1 Forced Off-chip/Outboard Memory Array Mapping

The `OutboardArray` attribute forces that an array is to be mapped to a region of external memory instead of being allocated a private array inside the current compilation. Large arrays are placed off
chip in this way by default. It is up to the substrate 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.

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

OLD: Pre performance profiling: 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’.

8.2 Off-chip load/store ports

KiwiC generates load/store ports to access off-chip memory. (Off-chip means not instantiated by KiwiC, so the addressed resource can be on the same die in reality). With more load/store ports in use, greater memory access bandwidth is available AND greater opportunities for out-of-order memory service exist.

The off-chip port architecture is defined in recipe/command line settings. It is also written as a report file in every KiwiC run. The Off-chip Memory Physical Ports/Banks report looks something like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>No Words</th>
<th>Awidth</th>
<th>Dwidth</th>
<th>Lanes</th>
<th>LaneWidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadstor1</td>
<td>4194304</td>
<td>22</td>
<td>256</td>
<td>32</td>
<td>8</td>
</tr>
</tbody>
</table>

Total load/store port width = bits per lane * number of lanes.

Default -res2-loadstore-port-count=1
Number of LOADSTORE ports for automatic off-chipping of large RAMs.

res2-loadstore-port-lanes 32 LOADSTORE ports - number of write lanes.

res2-loadstore-lane-width 8 LOADSTORE lane width

When the number of lanes is 1 no lane write enables are used and the memory is word addressed always.

A suitable behavioural Verilog fragment to connect to them for simulation test purposes is available as part of the distro in the rams folder.

Typical DRAM controllers run much faster than the FPGA user logic and hence a wide word is presented to the KiwiC-generated code of 256 bits or so.

The user’s wanted data width is either rounded up to some integer multiple number of external words, or some fraction of a word where the fraction is rounded up to a bounding power of 2 number of lanes.
The restructure log file will explain, somewhat cryptically, how each DRAM bank is being used with a table that contains interleaved entries covering all the banks (portnames). The lines in this report can be decoded with experience: D16 means sixteen bits wide. AX means an array. etc.

Off-chip 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. The KiwiC-generated code should be connected to an externally-provided memory controller that will often also also include some sort of cache.

Three off-chip protocols are supported BVCI, HSIMPLE and HFAST. HFAST is most commonly used. BVCI allows multiple transactions to be in flight.

When we say ‘off-chip’ we simply mean outside the generated hardware circuit - the substrate configuration may put various items on the same Physical chip.

### 8.2.1 HSIMPLE Offchip Interface & Protocol

Low-performance HSIMPLE uses four-phase handshake and only transfers data once every four clock cycles. It is more suitable for connecting to simple peripherals than DRAM. The following nets will require connection to the synthesis output when the DRAM is in use with the default, simple, 4/P HSIMPLE protocol.

```vhdl
output reg hs_dram0bank_req,
input  hs_dram0bank_ack,
output reg hs_dram0bank_rwbar,
output reg [255:0] hs_dram0bank_wdata,
output reg [21:0] hs_dram0bank_addr,
input  [255:0] hs_dram0bank_rdata,
output reg [31:0] hs_dram0bank_lanes,
```

When the number of lanes is one, there are no lane outputs.

### 8.2.2 HFAST Offchip Interface & Protocol

HFAST1 offers one cycle read latency and back-to-back operations, achieving 100 percent throughput. It is ideal for front-side cache connections where prefetch is not used.

The signature for HFAST is typically as follows (the total width and number of lanes and address bus width are all parameterisable).
output reg hf1_dram0bank_opreq,
input hf1_dram0bank_oprdy,
input hf1_dram0bank_ack,
output reg hf1_dram0bank_rwlock,
output reg [255:0] hf1_dram0bank_wdata,
output reg [21:0] hf1_dram0bank_addr,
input [255:0] hf1_dram0bank_rdata,
output reg [31:0] hf1_dram0bank_lanes,

When the number of lanes is 1 no lane write enables are used and the memory is word addressed always.

A DDRAM2 controller is available in the file kiwi/rams/ddr2-models. This can be used for high-level simulations. It instantiates the DDR_DRAM_BANK underneath itself.

A behavioural model of a DDRAM2 is available in the file kiwi/rams/ddr2-models. It has signature:

// (C) 2010-14 DJ Greaves.
// Verilog RTL DDR2 behavioural model - fairly high level.
// The SIMM or DIMM (all the chips of the bank) is modelled with one RTL module.
module DDR_DRAM_BANK(
input clk, // DDR Clock - 800 MHz typically. We use one edge
input reset, // Active high synchronous reset
input ddr_ras, // Active low row address strobe
input ddr_cas, // Active low col address strobe
input [log2_internal_banks-1:0] ddr_ibank, // Internal bank select
input ddr_rwbar, // On CAS: 1=read, 0=write. On RAS l=precharge,
input [2*dwidth-1:0] ddr_wdata, // The wdata and rdata busses are here twice their
input [awidth-1:0] ddr_mux_addr, // Multiplexed address bus
input [2*dwidth/8-1:0] ddr_dm, // Lanes: Separate nets here for +ve and -ve edge
output reg [2*dwidth-1:0] ddr_rdata); // Read data bus.

parameter log2_dwidth = 5;
parameter dwidth = (1<<log2_dwidth); // Word width in bits - we actually have twice
// FOR DRAM style
// E.g. MT41K256M32-125 DDR3 @ 800 MHz/1.25ns RCD-RP-CL=11-11-11 Arch=32M x 32 bits x 8
parameter LOG2_ROW_SIZE = 15; // Log_2 number of words per RAS
parameter LOG2_COL_SIZE = 10; // Log_2 number of words per CAS
parameter PRECHARGE_LATENCY = 11;
parameter ACTIVATE_LATENCY = 11;
parameter CAS_LATENCY = 11;
parameter log2_internal_banks = 3;
parameter awidth = LOG2_ROW_SIZE; // Address width in bits - word addressed.
// DRAM burst size - can be dynamically encoded in high-order CAS address. Currently fixed g for DDR) this requires 4 clocks to transfer the burst.
parameter burstSize = 4;

HFAST2 is the same as HFAST1 but uses a two-cycle, fully-pipelined read latency.

A simple cache is provided. Its signature is:

module cache256_hf1
(input clk,
input reset, // synchronous, active high.
The cache must be manually instantiated by the substrate designer.

HFAST arbiters can be instantiated on the front or back side of the cache, so that multiple synthesised load/store ports can share one cache or multiple caches can share one DRAM bank. Sharing would be inconsistent.

The default substrate runs the DRAM and DRAM controller at 800 MHz and the Cache and KiwiC generated code at 133 MHz which is 1/6th of this.

8.2.3 BVCI Offchip Interface & Protocol

Text missing.

8.3 AXI and HFAS-TO-AXI mapping

AXI has become the most prevalent SoC and FPGA bus interface standard. AXI supports burst transactions and out-of-order service. Such AXI service discipline is well-suited to a high-performance DRAM bank controller. (Such a bank controller typically has 8 internal banks, all of which can be concurrently open on a DRAM row.)

Today’s CPUs use multiple load/store stations per core that are pari passu with that core’s ALUs. KiwiC-generated hardware is no different. Each load/store station is busy with at most one scalar load/store request and this can only be served in order.

As with CPUs, there are two techniques that adapt between single-issue load/store stations: multiplexing and caching:

- Multiplexing multiple single-issue, in-order clients onto a single bus readily generates a traffic load that can be served out of order. In addition, there may be spatial locality between requests that can be aggregated into a burst.
• The front-side of a cache is optimised for random-access, low-latency operations. Since each is served (nominally) instantly, there is no scope for out-of-order discipline. On the other hand, the back side of a cache creates line fills and writebacks that are burst operations.

KiwiC load/store stations are served with HF AST interfaces. In the fullness of time, KiwiC will provide automated support for HF AST to AXI adaptation but currently a substrate that manually matches the number of load/store ports is required.

The substrate typically converts the KiwiC-generated HF AST interfaces to AXI or other off-chip protocols not currently supported by KiwiC. The substrate provider writes RTL transactors to convert protocols.

### 8.4 Off-chip address size

KiwiC assumes it can use address zero upwards in the off-chip space. The substrate must offset the address bus to address available SoC regions if this is not the case.

KiwiC accepts a recipe parameter to bound the amount of off-chip memory it can use in its one channel. Where a design attempts to use more memory, a compile-time error is raised.

`res2-loadstore-lane-addr-size` gives the off-chip address bus width in bits. In other words, this is the log2 no of words of memory available in each address space. Providing different limits for different off-chip spaces will be enabled in future. The word size and lane structure is defined with `res2-loadstore-port-lanes` and `res2-loadstore-lane-width` where the first of these is typically 4, 8, 16 or 32 and the second nearly always 8 (ie byte-sized lanes).

### 8.5 B-RAM Inference

B-RAM instantiation is normally automatic in FPGA tools. B-RAMs with an access latency of one clock cycle are normally used although KiwiC can support zero and two cycle reads (but how to access them is not described here! TODO).

A B-RAM is inferred from a structure following one of several paradigms based on all addresses passing through a single register or all read data being passed through a single register. These can be mapped onto the same underlying technology by posting the writes as necessary but the effects of read while writing to the same location differ.

KiwiC generates on-chip RAMs as explicit instances in the generated RTL. It uses 'read before' coding style. The FPGA Vendor 'read after' forms, where newly written data is read out are not explicitly found in the generated RTL: KiwiC will forward the data for itself when needed, either at compile or run time.

```vhdl
// (C) Xilinx 2009. Single-Port B-RAM with Byte-wide Write Enable: Read-First mode
// File: HDL_Coding_Techniques/rams/bytewrite_ram_1b.v
//
module v_bytewrite_ram_1b #(
    parameter SIZE = 1024,
    parameter ADDR_WIDTH = 10,
    parameter COL_WIDTH = 9,
    parameter NB_COL = 4)
(
    input clk,
    input [NB_COL-1:0] we,
    input [ADDR_WIDTH-1:0] addr,
    //
```
module SINGLE_PORT_BLOCK_RAM
(input clk,
 input wen,
 input [6:0] a,
 input [15:0] di,
 output reg [15:0] do);

reg [15:0] ram [0:127];
reg [15:0] do0;

always @(posedge clk) begin
 if (wen) ram[a] <= di;
 do0 <= ram[a];
 do <= do0;
 end
endmodule

Style 1:

always @(posedge clk) begin
 addr_reg <= addr ... ;
 if (wen ...) data[addr_reg] <= (wdata ...);
 rdata = data[addr_reg]; // Note blocking assign used or
 else the rhs freely used elsewhere.
 end

Style 2:

always @(posedge clk) begin
if (wen ...) data[addr] <= (wdata ...);
rdata_reg <= data[addr]; // No other reads elsewhere
end

There is also the dual-ported equivalents of these styles, supported by both Xilinx and Altera.

9 Filesystem Interface

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

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 (alongside 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.

The remainder of this part of the user manual is missing, but please check the Bowtie Genome Sequencer demo for an example of file system use.

9.1 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.
Part IV

Kiwi Performance Predictor

In 2015 a performance predictor was added to Kiwi so that estimates of run-time performance can be rapidly provided without having to do an FPGA place-and-route or even a complete pre-FPGA RTL simulation. The performance predictor is based on basic block visit ratios stored in a database that is updated with the results from short runs. When the application is edited and recompiled with KiwiC, a new prediction is generated, straightaway, based on the contents of the database generated by previous versions. Short profile runs of the new design can then be run to improve prediction accuracy. Every prediction is reported with confidence limits. The reported confidence is reduced (wider error bars) both by certain design edits and by extrapolating to runs that are much longer than those used for profiling.

Performance prediction is based on accurate knowledge of control flow branching ratios: the percentage of time a conditional branch is taken or not taken. This enables execution counts for each basic block to be estimated. Profile information from previous runs is the default basis for this knowledge. To ensure the information stored in the profile database is robust against program edits, it cannot be indexed by fragile tags such as a basic block number in global syntax-directed enumeration. Instead, performance prediction uses the method names occurring naturally in the application program as timing markers. Every method has a clear entry point as well as potentially several exit points (return statements are numbered in their textual order in the CIL byte code... branches to the exit). With loops that contain no method calls in their bodies, the user must add a method call to a dummy method (null body) and that method should be (preferably?) annotated with a KppMarker attribute. Conditional branches and basic block names are then taken in a syntax-directed way from the code between the named control-flow points and discrepancies in the control flow graph between named points is used to flag warnings and discard profile information no longer usable.

All call strings for a method can either be considered separately or in common. The call string is the concatenation of the call site textual names from the thread or program entry point. If the call strings are considered in common, they are being disregarded and the average over all callstrings is used.

These attributes also enable the user to control the way the performance estimation report is presented. They also enable the user to provide a substitute loop or visit count that overrides the stored profile. This provides the basis for extrapolating the run time from a small test or profiling data set to the envisioned real date size that will be processed on the FPGA.

Where the performance predictor cannot find profile information for a branch it assumes a 50/50 division and the number of such assumptions and their effect on the confidence in the result is included in the report.

9.2 Phase Changes, Way Points and Loop Markers

Hardware itself does not have a start and end time. Instead, performance metrics are always quoted between a START/FINISH pair of named events. A typical program is structured with a time-domain series of internal phases, such as ‘startup’, ‘load’, ‘compute’ and ‘report’. The performance predictor makes separate predictions for each phase and sums them. The confidence for different phases may be different, typically according to which part of the program was most recently edited. A marker between phases is called a way point. Kiwi.KppMark dummy calls and/or Kiwi.KppMarker attributes are used to define waypoints. Each way point has a name and all but the last start a phase that optionally also has a name. The entry and exit waypoints should be called START and FINISH respectively. The program’s control flow cannot loop around a way point. If a KppMarker is found in a loop body, or a method body where that method is called more than once, the provided labels are code point markers (explained below).

Kiwi.KppMark("START", "subsequent-phase-name1");
A waypoint is a special form of code point marker. The use of code point markers adds robustness to the information stored in the profile database against program edits, allowing it to be safely applied to edited programs. The markers provide index points that can be associated with loop heads and other control-flow points, to assist in robustness of the profile for complex method bodies. Basic block names are then named in a syntax-directed way with respect to, and as textual extensions of, the previous and next labelled control point.

KppMark has no innate multi-threaded capabilities and so should generally be set by an application’s master-controlling thread, assuming it has one.

9.3 Growth Parameter Assertions/Denotations

C# attributes also enable the user to provide a substitute loop or visit count that overrides the stored profile. This provides the basis for extrapolating the run time from a small test or profiling data set to the envisioned real data set size that will be processed on the FPGA. Also, hardware itself does not have a start and end time - it is static/eternal. Instead, performance metrics are always quoted between a start/end pair of named code labels, again specified with C# attributes. Times for various phases within a program, such as ‘load’, ‘process’ and ‘write out’, can also be predicted by inserting appropriate further control-graph delineations with an attribute that denotes a way point.

10 Generated RTL

Kiwi generates Verilog RTL for synthesis to FPGA by vendor tools. It can also generate SystemC but we do not use that here.

KiwiC will assume the presence of various IP blocks in Verilog. These include RAMs and fixed and floating point ALUs. It will instantiate instances of them.

The library blocks are generally provided in the following source files:

\[
\begin{align*}
\text{CV_FP_ARITH_LIB} &= \$(\text{HPRLS})/\text{hpr/cv_fparith.v} \\
\text{CV_INT_ARITH_LIB} &= \$(\text{HPRLS})/\text{hpr/cvgates.v} \\
\text{CVLIBS} &= \$(\text{CV_INT_ARITH_LIB}) \$(\text{CV_FP_ARITH_LIB})
\end{align*}
\]

10.1 RAM Library Blocks

Fixed-latency RAMs are provided in the cvgates.v. They have names such as CV_SP_SSRAM_FL1 which denotes a synchronous RAM with fixed read latency of one clock cycle (FL1) and one port (SP). The cvgates implementations are intended to be synthesisable by FPGA tools.

Parameter overrides set the address range and word and lane width.
10.2 ALU Library Blocks

These blocks are found in cv_fparith.v

Example: CV_FP_FL5_DP_ADDER - floating point, fixed latency of 5 clock cycles, double precision
CV_FP_FL_SP_MULTIPLIER

Key: FLASH=combinational. FLn = fixed latency of $n$ clock cycles, VL variable latency with

11 Examples

11.1 Mini 32-bit get started counter

Here’s how to make a simple counter that produces a 32-bit net-level output.

using KiwiSystem;
{
  [Kiwi.OutputWordPort("counter")]
  static int counter;

  [Kiwi.HardwareEntryPoint()]
  static int Main2()
  {
    while(true)
    {
      Kiwi.Pause();
      counter = counter + 1;
    }
  }
}
Part V
Expert and Hardware-level User Guide

12 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.

12.1 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.

```csharp
[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.

```csharp
[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.

12.2 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.

```csharp
[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 [1], 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.

```c
[Kivi.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=6, sw=8
```

The following examples work

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

// six bit local var
[Kivi.HwWidth(6)] static int j = 0;
```

A short-cut form for declaring input and output ports

```c
[Kivi.OutputIntPort("")]
public static int result;

[Kivi.OutputWordPort(31, 0)]
public static int bitvec_result;
```

12.3 How to write state machines...

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

12.3.1 Moore Machines

First compare the Moore machines define by main_pre and main_post:

```c
[Kivi.Input("")]
int din;
[Kivi.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.

12.3.2 Mealy and combinational logic:

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

```c
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.

```c
[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?

12.4 State Machines

Explicit state machines can be coded fairly naturally:

```c
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:

```c
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.

12.5 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 [2] 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.

```c
[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.

13 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

14 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 3 shows key components of the main flow through the tool as set up with the provided recipe file (KiwiC00.rcp). The full recipe contains ten or so stages 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
Figure 4: The internal flow of the KiwiC front-end recipe stage.

©2011-16 DJ Greaves + S Singh
The .NET executable bytecode is read using the Mono.Cecil front end. Any needed libraries, including Kiwi.dll and Kiwic.dll are also read in. These are combined with some canned (hardwired in the front end) system libraries. The result is a large CIL abstract syntax tree. This can be output for tracing/debugging if desired.

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 for each user thread. They 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, r.h.s. 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 CIL input is preserved as console output in the generated forms (eg. $display in Verilog RTL).

The conversion to FSM is described in §20 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.

14.1 Background: HPR Library and Orangepath Tool

HPR/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 receipe 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 'convert_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.)
Figure 5: The main flow implemented in the KiwiC tool (same as figure ??).
Part VII

Miscellaneous

15 FAQ and Bugs

Note: Do not use Console.Writeline or Write with 4 or more arguments since MCS converts these calls to a different style not supported by KiwiC.

Q. Why do I get KiwiC: do not update your formal parameters for now.
A. The message you have now encountered is a result of storing or modifying a formal parameter to a function which is functionality that is sadly currently missing. Just copy your formal into a local variable at the start of the function body for now.

```c
void myfun (int fp)
{
    int copied_fp = fp;
    copied_fp += 1; // Do not directly modify your call by value formals for the moment. (P)
}
```

Q. What does this mean: System.Exception: CV_INT_FL2_MULTIPLIER_S unrecognised gate for presim:
A. This is from the built-in simulator, diosim. The design has used a fixed-latency of 2 multiplier (from cvgates.v or elsewhere) but the simulator does not know how to simulate it. Restructure its own simulation model for each component it deploys, but one fix is to not apply diosim to (miss off the -sim=nnnn flag) since the generated RTL should be ok.

Q. How can I get meaningful line numbers in my error messages from KiwiC?
A. Line numbers are hard to track through the C# front end, but errors should be reported on a method name basis. There is a fairly-detailed log file written to the obj/h02_kiwic folder but it is hard to understand. Increasingly you can get a finer cross reference with the source code by embedding waypoints in your source file.

Q. Why are bools using 32 bits, even in arrays?
A. A C# compiler may compiles them this way - CIL has no run-time bool class. It may be best to instantiate your own bit-packed array class with suitable overloads if you want to exploit bit-level storage.

Q. Can I generate a VCD using the built-in simulator, diosim.
A. 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] FIFO1_FIFO2_result; instead of reg FIFO1_FIFO2_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 have lots of X uncertain values in my simulation

A. Is the source of X from flip-flops that are not cleared at reset or is it floating inputs? Did you put -vnl-resets=synchronous? You do not need this on all FPGA simulations since FPGA flops are self resetting, but with the associated simulator you may need this.

It is good to trace the pc10nz program counter (or similar name) generated by KiwiC for each thread. This normally starts at zero. You can cross check that with the dot graphviz output or the tables appended to the back of the .v file (also present in the obj/h08_restructure/000... file).

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 \{tt -kiwic-finish\}. \index{-finish} \index{-kiwic-finish}

\begin{verbatim}
-kiwic-finish= [ enable | disable ]
\end{verbatim}

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.

Another solution is to put an infinite loop in the main entry point (perhaps including a Kiwi.Pause() statement in the loop if there is other complexity to ensure KiwiC spends less time working out that it is infinite).

Q. I get the following strange error message even when I am sure my program is not allocating fresh memory inside the thread lasso loop: Bad form heap pointer for obj_alloc (already allocated a variable sized object?).

A. Check whether you are allocating local arrays on the stack: if these are just constant lookup tables makes sure you put the keyword const in front to make them statically-allocated.

Q. Can I use in Kiwi the data type struct?
A. Kiwi aims to support static and dynamic classes well. Structs in C# are slightly odd things and Kiwi has a little support form them that is not well tested. Normally you should use classes but it you have a good reason to use structs we can see how well it is currently working.

Q. Does Kiwi supports the keyword ‘break’?
A. Yes, all control flow constructs like for/while/continue/break are handled by the C# compiler and just appear as goto’s in the CIL dot net code input to KiwiC.

Q. If I instantiate: static ulong[] buffer = new ulong[10], KiwiC will generate registers. In the simulation I noticed that I got, not 10 regs, but 18
I tried also with static ulong[] buffer = new ulong[5] and got 8 regs.

A. A short array of 10 entries is most likely to be mapped to 10 separate registers, especially if you only use constant subscripts. If your subscripts can be determined not to use the whole range or only use multiples of a some constant or fall in disjoint regions you will get other patterns. Quite how it gets allocated depends on the pattern of subscriptions you use. The figure 18 you quote is presumably inflation on top of that from other aspects of the design? Kiwi does not replicate and mirror storage at the moment although this could possibly be useful under some circumstances. Ditto 5 to 8. Also, it depends on how many time you assign to buffer and how many different calls to new you make. I assume you have just one assign outside of any loop or re-ntrant code.

Q. I try to instantiate 2 ulong[256] arrays. In the RTL there are two memories, one A[64 US][255:0][63:0] and one A[64 US][2047][63:0]. I checked also the verilog file and I noticed that the address of the second array, whenever there is an operation, is multiplied by 8. Is it because of some optimization?
A. The byte address of a u64 array will be a factor of 8 different from the word address. Also If you only used every 8th location in an array, the repack recipe stage might notice this and divide each address by 8 to save space. The addresses on the input to the repack recipe stage are byte addresses. The addresses afterwards should be efficiently packed addresses, which would be /64 if you used only every 8th word owing to both effects acting.

Q. KiwiC seems to be deleting most of my design. Is this correct?
A. The processing stage called conerefine deletes unused parts of the design. A part of the design is unused if it generates no output. Outputs include PLI calls like Console.WriteLine or net-level outputs flagged with Kiwi.OutputWordPort or similar. Adding -conerefine=disable to the command line suppresses the associated trimming, resulting in a larger RTL or other output file, although occasionally this may lead to elements being present at the code generation stage that cannot be sensibly rendered in the output language.

Q. KiwiC is taking a very long time to compile and then fails. It says it has run out of unwind steps. Why is this?
A. If you are in a soft pause mode, KiwiC will infer Kiwi.Pause() statements where it feels necessary to allocate work to clock cycles. In hard pause mode KiwiC is not free to insert such pauses. If you have an infinite loop without a pause in it, KiwiC will fail to unwind the loop. Check that all control paths (PC trajectories) inside infinite loops have at least one Kiwi.Pause() inside them.

+++ 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.

For the output port to be a sixteen bit register but used the 'uint' dot net vartype 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.

16 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/dig11/wwwhpr/gpibpage.html

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

17 Transactor Synthesiser

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

18 Asynchronous Logic Synthesiser

The H1 tool implements an asynchronous logic synthesiser described on this link.
19 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.)

20 Bevelab: Synchronous FSM Synthesiser

Bevelab is an HPR plugin that converts HPR threaded forms to RTL form. Both the input and outputs to this stage typically have the concept of a program counter per thread, but the number of program counter states is greatly reduced. In the output form, many assignments and array writes are made in parallel. A custom data path is generated for each thread and the program counter becomes the internal state of a micro-sequencer that controls that data path. The emitted program counter does not need to be treated differently, then on, from any other scalar register, although the distinction is preserved in the output form for readability, debugging (and perhaps to assist proof tools), and for the Kiwi Performance Predictor that needs to track the control flow graph through the complete toolchain.

(An alternative to bevelab is the VSFG stage (21) that can achieve greater throughput with heavily-pipelined components in the presence of complex control flow.)

Usually, the input is in DIC form where the DIC contains assignments, conditional gotos, fork/join and leaf calls to HPR library functions. More-advanced imperative control flow constructs, such as while, for, continue, break, call and return need to have been already removed.

The resulting RTL is generally ‘synthesisable’ as defined by language standards for Verilog, VHDL and SystemC. the resulting RTL is generally ‘synthesisable’ as defined by language standards for Verilog, VHDL and SystemC. Although it uses common subexpression sharing, it is hopelessly inefficient since a naive compilation to hardware would instantiate a fresh, flash arithmetic operator at every textual site where an operator occurs. In addition, it will typically be full of structural hazards where RAMs are addressed at multiple locations in one clock cycle, whereas in reality they are limited in number of simultaneous operations by their number of ports. Finally, the RAMs and ALUs are assumed to be combinatorial by this RTL, whereas in realitythey are pipelined or variable latency.

Converting to one of the output languages, such as SystemC, is by a subsequent plugin. But the output of bevelab is normally first passed via restructure (that overcomes structural hazards and performs load balancing) to the verilog-gen plugin where it is converted to Verilog RTL syntax.

Both bevelab and restructure can trade execution time against number of resources in parallel use: the time/space fold/unfold. Bevelab is the core component of any ‘C-to-gates’ compiler. It packs a sequential imperative program into a hardware circuit. As well as packing multiple writes into one cycle, it can unwind loops any bounded number of times. Loops that read and write arrays can generate very large multiplexor trees if the array subscripts are incomparable at unwind time, since there are very many possible data bypasses and forwardings needed. Therefore, a packing that minimises the number of multiplexors is normally chosen. A simple greedy algorithm is used by default: as much logic as possible is packed into the first state, defined by the entry point to the thread, subject to four limits:

1. a multiplexing logic depth heuristic limit being reached,
2. a name alias (undetermined array address comparison) being needed,
3. a user-annotated loop unwind limit being reached, and
4. containing an intrinsically pausing operation.
Once the first state is generated, which may contain multiple input conditional branches that become predication within that state, successive micro-sequencer states are generated until closure.

Certain operations are already known to be pausing. One is a user-level explicit pause where the source code contains a call to ‘Kiwi.Pause()’. This is needed for net-level protocols, such as parallel to serial conversion in a UART. Others, such as trying to use results from integer divide, any floating point arithmetic, non-fully-pipelined multiply and reads from RAMs that are known to be registered also generate pauses when their source operands are also generated in the current micro-sequencer state.

Bevelab operates using the heuristics given in Table 3 takes 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 over 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 capabilities 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.

---

**Table 3: Bevelab Heuristic Table.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Style</th>
<th>Default</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of name aliases array read</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of multiplexors in logic path</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum default number of iterations to unwind</td>
<td>loops</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
The string \texttt{-resets as synchronous} should be passed in to introduce asynchronous resets to the generated sequencer logic.

The string \texttt{-resets none} should be passed in to suppress reset logic for FPGA targets. FPGA's tend to have built-in, dedicated reset wiring.

\texttt{-synthcontrol 'preserve-sequencer;resets:none;sequencer:packed'}

Bevelab has a number of scheduling algorithms (selectable from recipe of commandline).

\section*{20.1 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 \text{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.

\section*{21 VSFG - Value State Flow Graph}

VSFG is an alternative to the bevelab plugin - it uses distributed dataflow instead of having a centralised micro-sequencer per thread. It is based on the paper ‘A New Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware’ \cite{4}. It can achieve greater throughput with heavily pipelined components in the presence of complex control flow compared with traditional loop unwinding and static scheduling.

Its implementation within Kiwi is currently experimental (January 2015).

\section*{22 PSL Synthesiser}

The PSL synthesiser converts PSL temporal assertions into FSM-based runtime monitors.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Style</th>
<th>Default Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max no of integer adders and subtractors per thread</td>
<td>flash</td>
<td>unlimited</td>
</tr>
<tr>
<td>Max no of integer multipliers per thread</td>
<td>one-cycle</td>
<td>5000 bit products</td>
</tr>
<tr>
<td>Max no of integer dividers per thread</td>
<td>vari-latency</td>
<td>5</td>
</tr>
<tr>
<td>Max no of F/P ALUs per thread</td>
<td>fixed latency of 5</td>
<td>5</td>
</tr>
<tr>
<td>Max size register file (bits)</td>
<td></td>
<td>512</td>
</tr>
<tr>
<td>Max size single-port block RAM per thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max no of single-port block RAMs per thread</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Max no dual-port block RAMs shared over threads</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Max size dual-port block RAMs shared over threads</td>
<td>bits</td>
<td></td>
</tr>
<tr>
<td>No of DRAM front-side cache ports</td>
<td></td>
<td>unlimited</td>
</tr>
<tr>
<td>No of DRAM banks</td>
<td></td>
<td>platform-specific</td>
</tr>
</tbody>
</table>

Table 4: An Example Structural Resource Guide Table.

23 **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.

24 **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.

25 **Repack Processing Step**

The repack function is essentially KiwiC-specific. It is therefore described in the KiwiC chapters of this manual ($\S$ 4.3.1).

26 **Restructure Processing Step**

Restructuring is need to overcome structural hazards arising when there are insufficient resources for all the required operations to take place in parallel and to generally sequence operations in the time domain. Resources are mainly ALUs and memory ports. Table 4 shows the main parameters that control time/space trade off while restructuring a design. Further parameters relate to the cache size and architecture, DRAM clock speed. The repack phase ($\S$ 25) generated as many memories as possible. These must now be allocated to the allowed hardware resources, which may mean combining memories to reduce their total number, but taking into account a good balance for port bandwidth. Hardware platforms vary in the number of DRAM banks provided. The number of block RAMs inside an individual FPGA, like the number of ALUs to use, can be varied between one compilation and another.

The restructure phase bounds the number of each type of structural resource generated for each thread. It then generates a static schedule for that thread. Certain subsystems can have variable latency, in which case the static schedule is based on the average execution time, with stalls and holding registers being generated for cases that
run respectively slower or faster than nominal. The schedule may also get stalled at execution time owing to
dynamic events that cannot be predicted in advance. Typical dynamic events are cache misses, contention for
shared resources from other threads and blocking message passing between threads.

The scheduler statically maps memory operations to ports on multi-ported memories. It overcomes all static
hazards, ensuring that no attempt to use a resource more than once at a time occurs. It therefore ensures that
different operations occur in different cycles, with automatic insertion of holding registers to maintain data
values that would not be available when needed.

The five-stage pipeline for FPUs consists of, for an add, the following fully-pipelined steps: 1. unpack bit fields
and compare mantissas, 2. shift smaller mantissa, 3. add mantissas, 4. normalise, 5. round and repack.

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.

Certain 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 synthesizable 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.

27 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.

28 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.

29 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.

30 HPR Command Line Flags

The very first args to an HPR/Orangepath tool are the early args that enable the receive 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.

The developer mode flag, `-devx`, enables internal messages from the toolchain that are for the benefit of developers of the tool. Setting the environment variable `HPRLS_DEVX=1` performs the same action.

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 KiwiC. 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-sequence` 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 § ??.

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 § ??.

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.

`-vnl-rootmodname name`

Use the `-vnl-rootmodname` flag to set the output module name in Verilog RTL output files.
-vnl-roundtrip name= [ enable | disable ]

Converts generated Verilog back to internal VM form for further processing.
When enabled, generated RTL will be converted back again before (for example) being simulated with diosim. When disabled, the input to the verilog generate (vnl) recipe stage will be passed on unchanged and a typical recipe will then simulate that directly.

"-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 ever 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.

-restructure 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 BVCI or HSIMPLE.

30.1 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.

### 30.2 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.

### 30.3 Simulation Control Command Line Flags

The HPR L/S library provides 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 `-diosim-techno=enable` flag causes print statements from the simulator to include ANSI colour escape codes for various highlighting options.

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.

### 31 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 digif plot format, but a VCD format should be added.

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 cifie.

Before use, please set the MOSML and HPRLS shell variables

At the computer lab use:

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

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

```
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 implemenation of HPR.

If you get this error:

```
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:

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

or

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

also
export MOSML=/usr/local/mosml2.01

Basic invocation, in any directory where the source .il file resides

$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:

KLIB=$(PREFIX)/support/Kiwi.dll
KIWIC=$(PREFIX)/bin/KiwiC $(KLIB)

ROOT=-root 'DVI4.DVI_Ports;DVI4.Generator.GeneraTe_Synch_Only'

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

framestore: framestore.exe
  $(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/cil"

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. Installation instructions are available under 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

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
   --prefix=<where it should be installed to>
   --with-mono=<install_dir_of_mono>
   --with-gmp=<install_dir_of_glib>

   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:

```
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/mono-1.2.5.1/include"
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.

### References


Index

- bevelab-default-pause-mode, 28
- conerefine=disable, 53, 58
- diosim-techno, 68
- kiwic-cil-dump, 12
- no-cone-refine, 65
- repack=disable, 53
- sim-gates, 68
- sim-rtl, 68

A* Live Path Interface Synthesiser, 59
Assert, 17
assertions, 33
Asyncronous Logic Synthesiser, 59
Auto pause control, 28
AXI, 40, 47

Becontrol, 66
Bitvector I/O, 31, 47
Bitwise I/O, 31, 47
Blockb pause control, 28

cache, 37, 39
CAMHDL, 68
CIL, 54
Cil-wind-budget, 29
ClockDom (attribute), 31, 50
Clone, 17
Conditional compilation, 16
Cone refine, 65

Default pause mode, 28
diogif, 68
diosim:traceon, 69
divide by zero exception, 18
Dpath, 63
DRAM, 30, 47

Early arg -devx, 65
Early arg -give-backtrace, 65
Early arg -loglevel, 65
Early arg -recipe, 65
Early arg -verbose, 65
Early args, 53
Elaborate, 29
EndOfElaborate(), 28
Environment.Exit(1), 18
Espresso, 65
Exceptions, 17
exceptions, 18

exit, 18
Fecontrol (elab), 30
Filesstem, 43
finish, 67
Framestore, 19
Fsmsgen (-resets asynchronous), 66
Fsmsgen (-resets none), 66

gatelib, 68
generate loops, 21
Generate-nondet-monitors, 63
GLOBALS, 66
GPIO, 47

Hard pause control, 28
hard pause mode, 47
Hardware Server, 43
help, 68
hpr_exit, 69
HwWidth (attribute), 50, 47

I2C, 47
Ifshare, 67
inHardware, 16
Initiator, 67
Intel Hex, 64
IntPtr, 17

Kiwi, 54
Kiwi.Elaborate, 29
Kiwi.HardwareEntryPoint attribute, 11
Kiwi.KppMarker, 45
Kiwi.NoUnroll, 29
Kiwi.Subsume, 29
Kiwi.Unroll, 29
KiwiC, 54
KppMarker, 44, 45

load/store port, 37
LocalLink, 17
logging, 65
Loop unwind, 29
loop unwind, 67

makefile, 10
ManagedThreadld, 24
Maximal pause control, 28
memory channels, 40
Monitor, 67
monodevelop, 10

NeverReached assertion, 28
new, 16
Nosubexps, 66
NoUnroll, 29
null pointer exception, 18
nuSMV, 67

Off-chip, 30, 36
On-chip, 30
Onehot, 61
opentrace, 68
Outboard, 66
Output Formats, 64

Pause Mode, 19
PauseControlSet, 19
Performance Predictor, 44
plot, 68
Pointer Arithmetic, 16
Preserve-sequencer (synthcontrol), 61
profiling, 9
PSL, 62

Re-structure, 63
recipe, 11
Remote (attribute), 32
remote console, 8, 9, 18
Remote() Attribute, 27
render-root, 67
Repacker, 63
Resets:asynchronous (synthcontrol), 61
Resets:none (synthcontrol), 61
restructure, 63
Restructurer, 63
root, 66
rootmodname, 66
roundtrip, 66
run-time exceptions, 18
rwtrace, 68

SAT-based Logic Synthesiser, 64
Server (attribute), 43
SetPauseControl, 28
sim, 68
Simulation time, 18
SMV, 67
Soft pause control, 28
SSMG, 63

Statecharts, 63
Structural hazard, 63
Subsume, 29
SynchSRAM, 30
Synchronous FSM Synthesiser, 60
Synthcontrol, 61
Synthcontrol (-array-scalarise all), 66
Synthcontrol (cone-refine-keep), 67
Synthcontrol (preserve-sequencer), 66
Synthcontrol (sequencer=unpacked), 66
Synthesis Engines, 59

SysC, 67
SystemC, 64, 67

Target, 67
tick counter, 18
tid, 24
time, 18
tnow, 18
ToString, 17
traces, 68
Transactor Synthesiser, 59

Ubudget, 29
ubudget, 67
ucode, 68
Unroll, 29
unsafe CIL code, 16

Verbose, 65
Verbose2, 65
version, 68
val, 68
val-rootmodname, 66
val-roundtrip, 66
volatile, 65

way point, 45
Width (register), 30, 47
Wrapping, 40, 47
Write, 23

Xtor, 67