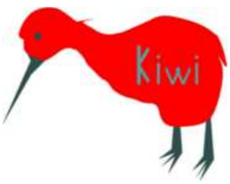
Kiwi Scientific Acceleration Manual
Rough Draft User Manual (KiwiC Version Alpha 0.3.2e) (Sept 2017)

November 14, 2019

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-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, KiwiC, that converts .NETbytecode into Verilog RTL for further compilation for FPGAexecution. In the future, custom domain-specific front ends that generate .NETbytecode can be used.

The Kiwi technology has many potential uses, but some of note are:

1. Kiwi-HPC: High-performance computing or scientific acceleration.
2. ASIC hard-core generation for standard algorithms that are to be implemented in silicon, such as MPEG compression.
3. Routing logic for software-defined networking.
4. Rapid transaction processing and hardware implementation of automated trading algorithms.

Compared with existing high-level synthesis tools, KiwiC supports a wider subset of standard programming language features. In particular, it supports multi-dimensional arrays, threading, file-server I/O, object management and limited recursion. Release 1 of KiwiC supports static heap management, where all memory structures are allocated at compile-time and permanently allocated to on-FPGA RAM or external DRAM. Release 2 of KiwiC, which has had some successful tests already, supports arbitrary heap-allocation at run time but does not implement garbage collection.

The Kiwi performance predictor is an important design space exploration tool. It enables HPC users to explore the expected speed up of their application as the modify it, without having to wait for multi-hour FPGA compilations in each development iteration.

The Kiwi compiler, KiwiC, itself consists of about 22 klocs (thousand lines of code) of F# (FSharp) code that is a front end to the HPR L/S logic synthesis library that is composed of another 60 or so klocs of F#. The code density for F#, like other dialects of ML, is perhaps (conservatively perhaps) 3 times higher than for common imperative languages like C++, Java and C#, so it is a significant project.

Note that the PDF version of this document tends to be more up-to-date than the HTML version.
`http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/manual/kiwic.pdf`

Contents

Introduction

Kiwi is a compiler and library and infrastructure for hardware accelerator synthesis and general support for high-performance scientific computing. The output is intended for execution of on FPGA or in custom silicon on ASIC.

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

For Kiwi 1, the current version, we have the following aims:

- 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 and heap structure sizes must all be statically determinable (i.e. at compile time).
- Program can use recursion but the maximum calling depth must be statically determined in Kiwi 1.
- 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, manifestly-implicit garbage generation event or explicit `obj.Dispose()` or `Kiwi.Dispose(Object obj)` in the same loop.
- Program can freely create new threads but creation sites statically determined too.

In Kiwi 2 we will relax the static restrictions and allow the size of data structures in DRAM to be determined at runtime. See
http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/kiwic-demos/linkedlists.html

Kiwi 2, planned to be available in the middle of 2017, supports three major compilation modes. These can be mixed in a single design, at a subsystem granularity, with the new incremental compilation support based on IP-XACT.

1. The Sequencer major mode is 'classical HLS'. It will generate a custom datapath made up of RAMs, ALUs and external DRAM connections and folds the program onto this structure using some small number of clock cycles for each iteration of the inner loops.
2. The Pipelined Accelerator major mode (§15) generates hardware with a low initiation interval (II). When the II is unity we say the component is 'fully-pipelined'. A fully-pipelined component will execute the complete computation every clock tick, accepting new argument data every clock cycle, allbeit with some number of clock cycles latency between a particular input appearing at the output.
3. The SoC Render major mode provides C# access to an IP-XACT-driven wiring generator with support for automatic glue logic insertion. The invoked subsystem is called HPR System Integrator (§39). This can target multi-FPGA designs and provides a clean mechanism to wrap up third-party IP blocks, such as CAMs. (This is now not a separate major mode — instead it is a separate application program invoked from a Makefile or command line.)

0.1 Asymptotic Background Motivation for FPGA Computing

The Von Neumann computer has hit a wall in terms of increasing clock frequency. It is widely accepted that Parallel Computing is the most energy-efficient way forward. The FPGA is intrinsically massively-parallel and can exploit the abundant transistor count of contemporary VLSI. Andre DeHon points out that the Von Neumann architecture no longer addresses the correct problem: he writes "Stored-program processors are about compactness, fitting the computation into the minimum area possible".

> 'Stored-program processors are about compactness, fitting the computation into the minimum area possible.' — *'Fundamental Underpinnings of Reconfigurable Computing Architectures'* by Andre DeHon.

Why is computing on an FPGA becoming a good idea ? Spatio-Parallel processing uses less energy than equivalent temporal processing (ie at higher clock rates) for various reasons. David Greaves gives nine:

1. Pollack's rule states that energy use in a Von Neumann CPU grows with square of its IPC. But the FPGA with a static schedule moves the out-of-order overheads to compile time.
2. To clock CMOS at a higher frequency needs a higher voltage, so energy use has quadratic growth with frequency.
3. Von Neumann SIMD extensions greatly amortise fetch and decode energy, but FPGA does better, supporting precise custom word widths, so no waste at all.
4. FPGA can implement massively-fused accumulate rather than re-normalising after each summation.
5. Memory bandwidth: FPGA has always had superb on-chip memory bandwidth but latest generation FPGA exceeds CPU on DRAM bandwidth too.
6. FPGA using combinational logic uses zero energy re-computing sub-expressions whose support has not changed. And it has no overhead determining whether it has changed.
7. FPGA has zero conventional instruction fetch and decode energy and its controlling microsequencer or predication energy can be close to zero.
8. Data locality can easily be exploited on FPGA — operands are held closer to ALUs, giving near-data-processing (but the FPGA overall size is x10 times larger (x100 area) owing to overhead of making it reconfigurable).
9. The massively-parallel premise of the FPGA is the correct way forward, as indicated by asymptotic limit studies [DeHon].

1 Download and License

Kiwi has been open source since early 2017 and is downloadable (perhaps on completion of a web form). The download page is `http://koo.corpus.cam.ac.uk/kiwic-download`.

1.1 Warranty

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

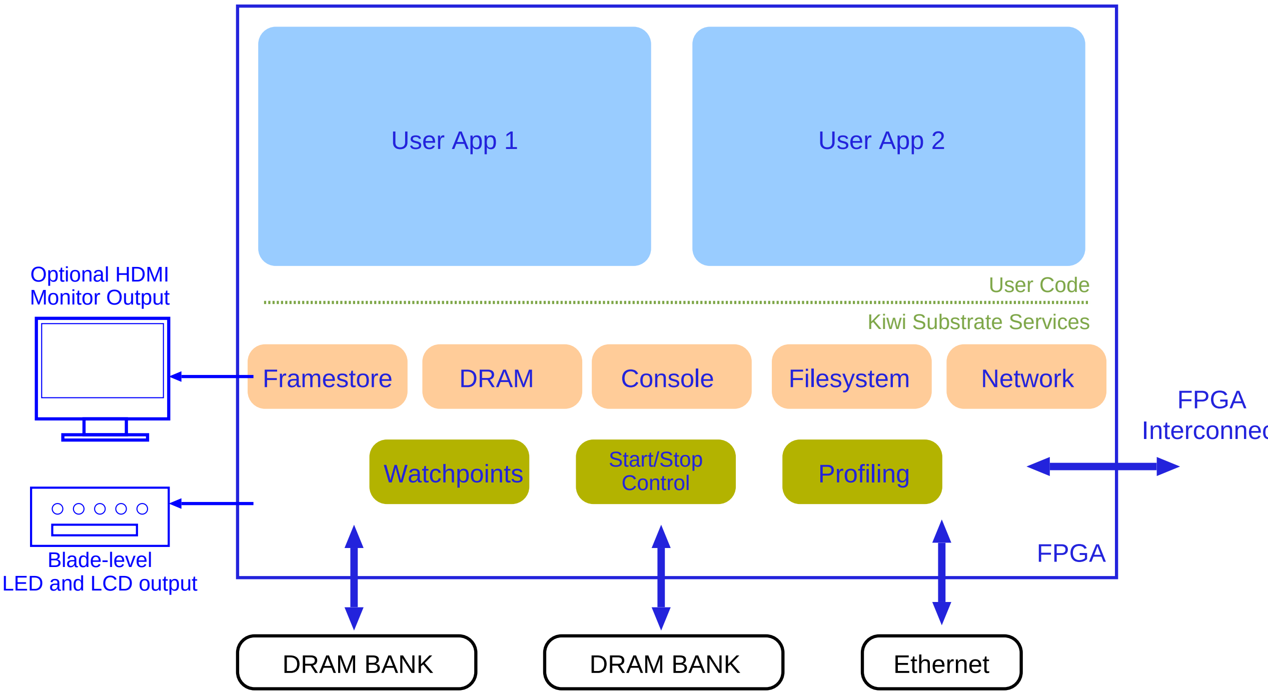


Figure 1: Kiwi Substrate: Typical Structure of the Kiwi FPGA.

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 FGPA at once (cite farming paper).

There is some basic information on the Zynq substrate here:

`http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/kiwic-demos/zynq-pio-dma`

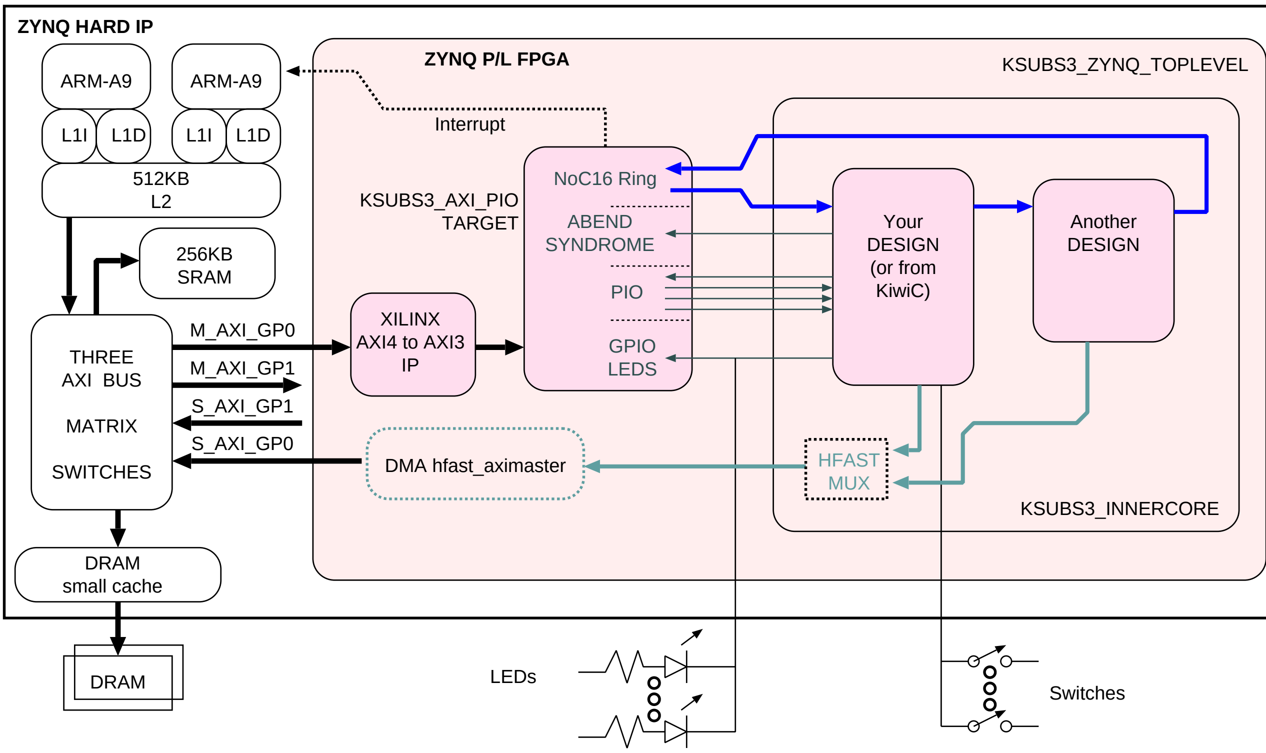


Figure 2: Kiwi Substrate: Structure of the Kiwi Ksubs3 Zynq Substrate.

2.1 Console and LCD stdout I/O and LED GPIO

2.2 Run-time Exception Handler

Run-time exceptions include integer divide-by-zero and null pointer de-reference, array bounds fail and runtime fail of `Debug.Assert()`. 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. Numeric casts can also be out of range, as in `(ushort)0x10000` (a CIL `conv_ovf.u2` assembly instruction is used.) In the future KiwiC can trap these overflows as run-time errors.

CIL bytecode has overflow trapping versions of the arithmetic operators that raise exceptions. We generate these from C# using `checked` keyword. Numeric casts can also be out of range, as in `(ushort)0x10000` (a CIL `conv_ovf.u2` assembly instruction is used.) 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.

Array bounds checking can also give a run-time error.

TODO: explain here about a per-clock domain error net generated by KiwiC as part of control wires.

The C# `Try` construct is partially implemented - it does not do anything - no C# exception handling is supported at the moment.

2.3 DRAM

DRAM and Caches are described in §8.4.

2.4 Watchpoints and Start/Stop Control

2.5 Framestore

Having very high bandwidth for writes to the framestore is an intrinsic feature of FPGA computing. The framestore can be part of the compute engine and used for high-performance visualisation. Or it might just be used for a progress indicator - e.g. percentage of the job processed and final output.

2.6 Profiling

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

Tick counter ... for tnow. `Kiwi.tnow`.

There is a simple version of `System.Diagnostics.Stopwatch` that is built trivially on top of the `Kiwi.tnow` mechanisms. It has the methods: `Reset`, `Start` and `Stop` . The current reading is via a getter for `Stopwatch.ElapsedMilliseconds`.

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 and since it is really focussed on Kiwi performance prediction which is immature. Currently it is best if you craft a **Makefile** based on one of the examples.

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

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

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

The means for loading to the FPGA is currently highly-platform specific. Each substrate should have its own user guide.

3 Get Started (Mono on Linux)

Kiwi is available in source and precompiled binary form.

Requirements:. You need a working dotnet environment (mono or Windows) on your machine including a C# compiler. It is also handy to have Modelsim or Icarus Verilog and Verilator and SystemC.

Do not do this for Windows, but for linux set your shell environment MONO variable

```
$ setenv MONO=mono
$
```

KiwiC/HPR is currently internally implemented in F# but you just need a C# compiler to use the precompiled distribution.

Kiwi binary form 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 it is now public.

You will need the F# compiler to compile HPR and KiwiC from source. The source build should be configured by editing the paths in `hpr/Makefile.inc`. Set the HPRLS shell variable to where the source code sits and run 'make' in `kiwipro/kiwic/src` to build the compiler.

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.

FSharp can be simply obtained with `apt-get install fsharp` on some machines.

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.

Note: on some versions of linux, applying the shell to the .exe file without invoking mono invokes wine. The wine Windows emulator munges the dot net O/S interface, telling KiwiC that it is running on Windows NT and interchanging slashes and backslashes. This is liable not to work very smoothly (although more robust programming inside KiwiC would help in this respect).

On a Windows box, to get started running KiwiC from the Windows command line, create a folder in your K-Distro folder, cwd to it and copy in a simple target, like `tiny.exe` described in §3.2. Then, in that folder run

```
..\lib\kiwic.exe tiny.exe -vnl tiny.v
```

3.1 Getting A K-Distro Binary Distribution

When the Kiwi system is itself compiled, it generates a folder called K-Distro. A user can download this folder or can compile it themselves from the Kiwi source distribution.

The important components of K-Distro are a lib folder containing all of the compiler dlls, a recipes folder containing the recipe XML file and a support folder containing `Kiwi.dll` and `Kiwic.dll` and other Kiwi-specific versions of the runtime system (such as System.Random, Math.Sqrt ...).

3.2 Using A K-Distro Binary Distribution

The Kiwi compiler itself is invoked via a shellscript called kiwic in the bin folder of the K-Distro. It is usual to put that folder on your PATH. The shellscript does little other than apply mono to `../lib/kiwic.exe`.

Place the Kiwi distribution somewhere on your filesystem. Let us call that place PREFIX. For source build this will be `$HPRLS/kiwipro/kiwic/distro/lib`. 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 for both FP and WD.
2. `Kiwic.dll` - This defines additional or replacement implementations of standard .NETlibrary functions for use by the KiwiC compiler and must nominally 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 .NETbinary (`.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 actually named on any command line.

To enable the same RTL file to be synthesised for FPGA by vendor tools and simulated [RTL_SIM] but to have slightly different behaviour (e.g. w.r.t. BIST self test) it is handy to define an external input to the Kiwi code that you tie low in the RTL_SIM testbench but strap high in the FPGA substrate pad ring.

```
[Kiwi.InputBitPort("FPGA")] static bool FPGA;
```

If you have these libraries in .cs form only, you will need to compile them to .dll form using mcs or similar. You will get some warnings about the 'unsafe' code they contain.

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.

Kiwi defines the terms WD, RTL_SIM and FP to define three, so-called, *execution environments*.

1. WD — Rapid development of applications on the workstation with performance prediction.
2. RTL_SIM — Verilog simulation (verilator is fastest) in case of KiwiC bugs and for performance calibration when interacting with RTL models of other system components.
3. FPGA — high-performance execution on the FPGA.

CIL assemblies have the option for an EntryPoint method to be designated. Having one of these is a main difference between .exe and .dll files.

I can add an option to recognise the entrypoint as a root, or make this default failing all else, but, for most cases, a different entry point is preferable for the different execution envs and we'd want to reserve entrypoint for WD. This needs to be looked at especially for multi-FPGA designs.

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

The `HardwareEntryPoint` attribute can take a pause mode as an argument. This will, in future, set the starting pause mode for that entrypoint, and moreover, be used to set pipelined accelerator mode.

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

It might be helpful to pass constant values as arguments to the HardwareEntryPoint but this is not supported. Instead, write a C# shim that takes no arguments and passes constants to a putative entry point. (But see also §7.17.)

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

all:
gmcs /target:library tiny.cs /r:$(KLIBO)
$(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:

```
$ cat > tiny.cs
using System;
using KiwiSystem;

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

Should you need it, KiwiC will write a disassembly of the PE file to `obj/ast.cil` in the current folder, enabled by recipe or command line flag '-kiwic-cil-dump=separately' or '-kiwic-cil-dumpl=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.

Also, you can externally disassemble a .net CIL file using `ikdasm` (which works better than the older `monodis`) shell command. The command pedump may also be useful.

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, for Kiwi 1, 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.

KiwiC implements a form of garbage collection called 'autodispose'. This can currently (October 2016) be enabled with `-autodispose=enable`. It will be turned on by default in the near future when further escape analysis is completed. Currently it disposes of a little too much and when that memory is reused we have a nasty aliasing problem since that store was still live with other data. This will crop up with linked-list and tree examples or where the address of a field in a heap object is taken.

When autodispose fails to free something (or is turned off) you can explicitly free such store with a call to `obj.Dispose()` or `Kiwi.Dispose(Object obj)`.

WRONG: 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 for the standard 32 and 64-bit IEEE precisions, but FPGAs really shine with custom precision floating point so we will add support for that while maintaining bit-accurate compatiblity between the execution environments.

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 invokation. Hence the the testandset primitives dissappear inside bevelab. ... TODO explain further.

4 General CSharp Language Features and Kiwi Coding Style

4.1 Supported Types

Kiwi supports custom integer widths for hardware applications alongside the standard integer widths of dotnet 8, 16, 32 and 64.

Char is a tagged form of the 16-bit signed integer form.

Single and double-precision floating point are supported.

Enumerations are supported with custom code points. MSDN says the approved underyling types for an enum are byte, sbyte, short, ushort, int, uint, long, or ulong, but Kiwi uses a suitable custom width of any number of bits.

One-dimension arrays are supported natively by Kiwi since they are part of the dot net virtual machine. The `Length` attribute does not always work at the moment since its implementation is fully at compile time and it fails where this varies at a given call site at run time. This can be fixed by the user using a wrapper class as per the higher-dimensional arrays.

Higher-dimensional arrays, including jagged arrays, are implemented in the `Kiwic.cs` file as wrappers around the native one-dimensional array. This is the same as for other dot net uses of higher-dimensional arrays. In theory, the standard dot net version of these wrappers should work well with Kiwi but we have not tried it. The Kiwi-supplied wrappers have various and properties and methods missing that should be available. Feel free to add them or paste the code from the standard implementations.

Classes and structs are supported. These are different from each other in C# (unlike C++). Although having much in common, C# treats structs and classess differently. C# passes structs by value to a method, meaning local modifications to contents do not commit to original instance. C# assigns structs by value, so all fields in the destination are updated by the assigment, rather than the handle just being redirected. Support for C# structs is being added.

Static and dynamic instances of classes and structs work. There is also some support for static arrays, as used in the C++ gcc4cil front end, but arrays are normally dynamically-allocated in C#. Certain restrictions regarding dynamic storage allocation and automatic garbage collection apply (§III).

4.2 Supported Constants and Variables

Kiwi supports static, instance, local and formal parameter variables.

Variables may be classes or built-in primitive types and arrays of such variables. An array may contain a class and a class may contain an array, to any nesting depth. Multi-dimensional arrays (as opposed to jagged arrays) are supported with a little syntactic sugar in the C# compiler but mostly via library class code provided in Kiwic.dll.

Signed and unsigned integer and floating point primitive variables are fully supported.

4.3 String Handling

Verilog and SystemC have 8-bit chars but C# and dotnet have 16-bit chars. KiwiC maps all 16-bit chars to the closest 8-bit char. UTF-8 escapes could easily be supported in this process but are missing at the moment.

Strings are supported a little, but there is currently no run-time concatenation or creation of new strings, so all such string creation operations must be elaborated at KiwiC compile time and hence be applied to constant strings.

4.4 Supported Operators

All standard arithmetic and logical operators are supported. Some operators, especially floating-point converts and floating-point arithmetic result in components being instantiated from the cv-gates.v library. Integer mod, divide and larger multiplies also result in ALU instantiation, unless arguments are constant identity values or powers of two that are easily converted to shifts. Divide and multiply by a constant may result in adders being generated.

4.5 Supported Class Features

Classes can be statically and dynamically allocated. Constructor code is executed.

Static classes have their constructor code called at compile time (although it is perhaps possible for the lasoo stem to end partly through the last one one of them.) The same goes for dynamic classes that are converted to static within the lasoo stem.

Class and array instance handles can be manipulated at run time. KiwiC (repack stage) will allocate a small integer for each one in each equivalence group where handles are interchanged or shared. KiwiC checks whether the null value requires a code point in each group. Run-time null dereference errors will be reported in the abend code register at some point soon.

Many class and array handles are never changed (the group has just one member) and hence are merely an artefact of the C# language. Such handles are optimised away inside KiwiC and have no run-time overhead. Class and array instance handles can be manipulated at run time. KiwiC (repack stage) will allocate a small integer for each one in each equivalence class where handles are interchanged. KiwiC checks whether the null value requires a code point. Run-time null de-reference errors will be reported in the abend code register at some point soon.

4.6 Supported 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 §7.10.
- 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.

- Local console debugging style output. Principly 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.
- Remote Console, Network and Filesystem I/O via the substrate gateway. See §9.2

4.7 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 §29), 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 §30), 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.

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

4.8.1 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

4.9 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 truely dynamically allocated at run time* (DJ Greaves 2015). Indeed, if an application becomes overly dependant on DRAM data then the FPGA advantage will disappear and a Von Neumann (e.g. x86) implemenation may likely 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.

The `Kiwi.MemSpace("spacename")` attribute can be added to a struct or class definition to take manual control of the DRAM bank its instances are placed in... Spacename is either a user name or a logical name for a memory bank. ... examples needed ...

Kiwi.MemSpace

`Kiwi.HeapManager`

Physical memories used for dynamic storage require a freespace manager. We can allocate a HeapManager 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 ...

4.10 Pointer Arithmetic

`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.

4.11 Garbage Collection

With Kiwi 1, 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 the compiled program must be matched with an equivalent dispose or garbage generation event in the same loop.

Where a heap object is allocated inside a loop that is not compile-time, it will potentially consume fresh memory on each iteration. There are two basic senarios associated with such a condition: either the fresh memory is useful, such as when adding items to a linked-list datastructure, or else it is not needed because the previous allocation is no longer live and the same heap space could be simply reused. This second case is fully served by converting to static allocation at compile time.

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

KiwiC V1 does not support genuine dynamic storage allocation inside an execution-time loop. Bit it provides two mechanisms to support dynamic to static reduction where dynamic store is not really needed. The first uses an explicit dispose and the second uses an implicit dispose. Either way, when the loop iterates, the active heap has shrunk and KiwiC makes sure to reuse the previously allocated heap record at the allocation site (call to C# new).

See the linked list example ... http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/kiwic-demos/linkedlists.html

KiwiC V1 arrays - Array sizes must all be statically determinable (i.e. at compile time).

System.BitConverter provides a typical use case that involves a lot of temporary byte arrays. The F# compiler also uses a lot of temporary structures and the KiwiC has a chance of compiling F# bytecode by exploiting the implicit disposal approach.

Arrays in .NETdo not have a `Dispose()` method. Instead an array can be disposed of with `Kiwi.Dispose<T>(`
This is a nop when running on mono/dotnet.

System.BitConverter returns char arrays when destructing native types and the arrays returned by BitConverter should be explicitly disposed of inside a non-unwound loop if KiwiC is failing to spot an implicit manifest garbage creating event, as reported with the an error like:

System.BitConverter returns char arrays when destructing native types. The arrays returned by BitConverter should be explicitly disposed of inside a non-unwound loop if KiwiC is failing to spot an implicit manifest garbage removal opportunity, as reported with the an error like

```
KiwiC +++ Error exit: BitConverterTest.exe: constant_fold_meets
 entry_point=5:: Bad form heap pointer for obj_alloc of type
 CT_arr(CTL_net(false, 32, Signed, native), 8) post end of elaboration
 point (or have already allocated a runtime variable sized object ?).
 Unless you are geninuely making a dynamic linked list or tree, this
 can generally be fixed using a manual call to Kiwi.Dispose() in your
 source code at the point where your allocation could be safely
 garbage collected.
```

Unless you are geninuely making a dynamic linked list or tree, the failed implicit garbage collector can generally be worked around using a manual call to `Kiwi.Dispose()` in your source code at the point where your allocation could be safely garbage collected.

`new`

For making trees and lists, see the linked list example ... http://www.cl.cam.ac.uk/research/srg/han/hprls/orang
demos/linkedlists.html

... field-arrays and spatial locality

4.12 Testing Execution Env: Whether I am running on the Workstation, RTL_SIM or the FPGA blades.

We need sometimes to achieve different behaviour, for debugging and scaling reasons, in the three execution environments.

1. For Workstation Development - WD - we can invoke
`Kiwi.inHardware()` and find that it returns false. The `Kiwi.dll` file returns false when run as a normal dotnet program, but KiwiC has a hardcoded bypass that makes it hold (return true).

2. For RTL_SIM check that inHardware returns false and that the `Kiwi.InputBitPort("FPGA")] static bool FPGA;` returns false. You should tie this net low in your simulator top-level instantiation.

3. Otherwise we are in FPGA. The Kiwi substrate for a hardware PCB should tie this net high in the pad ring.

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/debug code ... }`.

```
[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
  //return 'true' for FPGA and RTL simulation.
}
```

4.13 Clone

Clone of arrays and objects ....

4.14 Varargs

not there yet ... The varargs support is also pretty trivial to implement inside KiwiC under the current technique of fully inlining method calls during KiwiC compilation - it's just a matter of a few lines of simple interpretative code in the elaborator...

4.15 Delegates and Dynamic Free Variables

Kiwi Dynamic Method Dispatch

Dynamic method dispatch is where which function body that gets called from a callsite is potentially data-dependent. Computed function calls occur with action and function delegates and dynamic object polymorphism.

In C++ there are restrictions that higher-order programming is only possible within a class hierarchy. This arises from the C compatibility issues where the higher-order function passing does not have to manage an object pointer. These issues are neatly wrapped up in C# using delegates. An action delegate has void return type whereas a function delegate returns a value.

Kiwi supports the function and action delegates of C#.

KiwiC partitions dynamically-callable method bodies into equivalence classes and gives each body within a class an integer. These classes typically contain only a very few members each. It then uses constant folding on the entire system control-flow graph as a general optimisation. This may often turn a dynamic dispatch into a static dispatch, hence these integers will not appear in the output hardware unless truly dynamic dispatch is being used, such as in

```
Action<int, string> boz_green = delegate(int var1, string var2)
  { Console.WriteLine("  {1} {0} boz green", var1, var2);
  };
Action<int, string> boz_red = delegate(int var1, string var2)
  { Console.WriteLine("  {1} {0} boz red", var1, var2);
  };
for(int pp=0; pp<3; pp++)
  { Kiwi.Pause(); // Pause makes this loop unwind at run time.
    boz_red(pp+100, "site1");
    boz_green(pp+200, "site2");
    var x = boz_red; boz_red = boz_green; boz_green = x; //swap
  }
```

C# 3.0 onwards supports proper closures. These are implemented inside the C# compiler and compile fine under Kiwi provided the static allocation restrictions are obeyed.

Test55 of the regression suite contains the following demo.

```
public static Func<int,int> GetAFunc()
{
  var myVar = 1;
  Func<int, int> inc = delegate(int var1)
    { myVar = myVar + 1;
      return var1 + myVar;
    };
  return inc;
}

[Kiwi.HardwareEntryPoint()] static void Main()
{ var inc = GetAFunc();
  Console.WriteLine(inc(5));
  Console.WriteLine(inc(6));
}
```

This compiles and works fine. But, there is a Kiwi 1 resriction that the `GetAFunc` call must be before the end of static elaboration since this creates the closure that is allocated on the heap.

If no closure is needed, Action and Function delegates suffer from no static allocation restriction.

4.16 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. Enumerations print as integers.

4.17 Accessing Numerical Value of Pointer Variables

IntPtr types.

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("  Ntest14w line0 : pointer={0}", ha.ToString());
  Console.WriteLine("  Ntest14w line1 : left={0}", ha.left);

Might give:

Ntest14w line0 : pointer=Var(test14w/T401/Main/T401/Main/V_0%$star1$/test14w/
                   dc_cls%30008%4, &(CTL_record(test14w/dc_cls,...)), ..., )
Ntest14w line1 : left=32

Ah - this has printed the variable not its value!
```

4.18 Accessing Simulation Time

The Kiwi.dll library declares a static variable called `Kiwi.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

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

becomes

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

when output as Verilog RTL.

The substrate has a tick counter that is instantiated when tnow is used for FPGA execution and so the RTL_SIM code is a now a shim and not a direct call to the non-synthesisable `$time` infact... TODO fix.

4.19 Run-time Status Monitoring, Waypoints and Exception Logging

The following paragraphs are an introduction that should contain links the relevant manual parts.

The user requires various indications of whether an FPGA card is actively running an application. Some applications have a natural finish or exception halt state. Others run eternally using NaN-like schemes to convey errors.

Nearly all FPGA blade have LED outputs controlled by GPIO pins that are useful for basic status monitoring. FPGA substrates can 'virtualise' these LEDs so that host management software can see them. Some FPGAs have LCD or VGA framestore outputs that are also relatively easy to use for monitoring and results.

The Kiwi user may freely connect up LEDs and other I/O devices as part of their application-specific design, but the Kiwi system defines some standard techniques should preferably be used. These are conveyed through the substrate gateway for FP execution and are also implemented in the standard WD and RTL_SIM templates.

The Kiwi system also defines a 'WayPoint' progress indication framework that is used both for progress indication (debugging) and performance monitoring and prediction (§10.2). This is augmented with the abend syndrome register (§10.4).

... The sequencer index and waypoint for each thread can be remotely monitored via the substrate gateway. This provides ... abend syndrome register ... logs thread id, waypoint, pc value and abend reason.

4.20 Client versus Server Designs and Start Commands

An HLS run can generate a client or a server. A server is an accelerator or AFU that will be used by a client: it does nothing by itself. A client, on the other hand, starts work by itself, either straightaway or when given a start command. A client can be software running on a host that invokes accelerators via the Kiwi Substrate, or it may be an HLS design that starts from a `Kiwi.HardwareEntryPoint` attribute.

A client that performs DMA into a host must be told the DMA address before it starts.

The `ksubsRunStop` two-bit field is used to control hardware clients.

```
ksubsRunStop settings
  0  0   Assert synchronous reset
  0  1   Normal running
  1  0   Pause (deassert clock enable)
  1  1   (reserved for single step)
```

`[Kiwi.InputWordPort(‘‘ksubsRunStop’’)] static int ksubsRunStop` could be polled from C# as a potentially sensible design point. But we do not use that. Instead, where the client is instantiated by the substrate, its reset and clock enable inputs are connected to a hardware circuit that interprets the run stop field and which can allow just one single clock cycle of progress in some variants.

The Pause setting requires the client to have a clock enable input. The command line flag that ensures clock enables are present is `-kiwife-directorate-style=advanced` . The clock enable is called `hpr_ext_run_enable`. An AbendSyndrome register is also created in that mode. This external clock enable is ANDed inside the KiwiC-generated code with a local clock enable so that the client system stops (the clock enable is de-asserted) when abending or exiting.

4.21 Exiting Threads

4.21.1 Run Time Errors: Null pointer, Array bounds, Overflow, Divide-By-Zero...

The HPR L/S framework allows an RTL module to select one of three forms of standarised error output:

1. an eight-bit abend syndrome register and
2. a single-bit FAIL output, and
3. nothing.

The Kiwi substrate gateway will log the thread identifier, waypoint and sequencer index for threads that finish or abort. The user can reverse-engineer these via the KiwiC report file. An XML variant of that file for import into IDE needs to be provided in the future.

Sources of runtime error: It is possible to get a run-time **null pointer** exception.

The CSharp language supports arithmetic both with overflow ignored (as in C/C++) and checked. It is possible to get a run-time **checked overflow** exception. (But not yet supported in KwiC as of January 2017.)

It is possible to get a run-time **divide-by-zero** exception.

It is possible to get a run-time **array bounds** exception.

It is possible to get a run-time **sub-component** exception.

It is possible to get a run-time **unhandled C# exception** exception.

(Floating point exceptions are normally handled with via **NaN** propagation.)

4.21.2 Normal Thread and Program Exit

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

NB: We replaced -kiwic-finish with -kiwife-directorate-endmode

When the main thread of Kiwi program exits (return from Main), the generated code may include a Verilog $finish statement if the (OLD FLAG-TODO EDIT THIS) 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 sequencer state is created with no successors and the thread sits in that state forever. Hanging forever is the default behaviour for forked threads.

The argument to the $finish statement, if present, is also written to the abend syndrome register when present (see directorate styles). RTL designs also stop (clock-enable forced deasserted) when a syndrome is stored.

For use with a standard execution substrate, having a $finish statement in the generated design makes no sense,

`Environment.Exit(int syndrome)` can also be invoked within C# to cause the same effect as main thread return. The integer value is stored in the abend syndrome register and the RTL hardware design halts until next reset.

(Pipelined accelerators cannot exit since they have no sequencer (§15 and are permanently ready to compute.)

4.21.3 User-defined C# Exceptions

C# try-except blocks are supported as is exception handling. But no exceptions can currently be caught and all lead to either a compile-time or run-time abend.

In other words, the contents of a C# **catch** block are ignored in the current KiwiC compiler.

The contents of a C# **finally** block are executed under Kiwi as normal.

The following fragment shows how to throw a runtime exception that will cause execution to stop with an abend syndrome readable by the director shim.

Please follow the coding conventions in table XXX and note that the specific error code 128 is not an error and will not stop execution if thrown: it is the default aok code.

```
class myDemoExn: System.Exception
{
  // Note KiwiC latches onto an integer field name in uncaught exceptions containin
  int ecode = 129;

  public int error_code //
  {
    set { ecode = value; }
  }
}

class UncaughtExceptionTest
{
  // Steer away from Kiwi-1 dynamic storage complexity by
  // making the thrown exception a static.
  static myDemoExn my_faulter = new myDemoExn();

  public void runner(int roger)
  { for (int pp=0; pp<10;pp++)
      {
        Kiwi.Pause();
        Console.WriteLine(" runner {0}", pp);
        my_faulter.error_code = 101 + pp;
        if (pp == 5) throw my_faulter;
      }
  }
}
```

4.21.4 Debug.Assert or Trace.Assert

`System.Diagnostics.Debug.Assert(bool cond)` and friends ...

We can raise a run-time assertion problem that is logged in the abend syndrome register with code 0x20.

There is a compile-time variant called - not reached - or something ...

4.22 Pause Modes (within Sequencer HLS Mode)

Kiwi supports several major HLS modes, but the default, sequencer major HLS mode, generates a sequencer for each thread. When creating a sequencer, the number of states can be fully automatic, completely manual, or somewhere in between, according to the pause mode setting.

The mapping of logic operations to clock cycles is one of the main tasks automated by high-level synthesis tools, but sometimes manual control is also needed. Control can be needed for compatibility with existing net-level protocols or as a means to move the design along the latency/area Pareto frontier.

KiwiC supports several approaches according to the pause mode selected. Pause modes are listed 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 pause mode can, most simply, be set once and for all on the command line with, for examples `-bevelab-bevelab-default-pause-mode=soft`.

When in soft mode, the `bevelab-soft-pause-threshold` parameter is one of the main guiding metrics. But it has no effect on regions of the program compiled in hard-pause or other non-soft modes.

Typical values for the soft pause threshold are intended to be in the range 0 to 100, with values of 100 or above leading to potentially very large, massively-parallel designs, and with values around 15 or lower giving a design similar to the 'maximal' pause mode.

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

```
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 `Kiwi.PauseControlSet(Kiwi.PauseControl.softPauseEnable)`.

Nearly all 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 to a sequencer 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

No	Name	Pauses are inserted at
0	auto	?
1	hard	exactly where pause statements are explicitly included
2	soft	where needed to meet soft-pause-threshold
3	maximal	inserted at every semicolon
4	bblock	every basic block boundary

Table 1: Kiwi Pause Modes (within Sequencer Major HLS Mode)

```
<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.

4.23 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.

```
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.

4.24 More-complex implied state machines

```
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.

4.25 Inner loop unwound while outer loop not unwound.

```
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.)

4.26 Entry Point With Parameters

A top-level entry point with formal parameters, such as

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

is currently not allowed in normal sequencer mode, although in future it would be reasonable for these to be treated as additional inputs. This will be relaxed soon.

Top-level arguments are allowed in RPC (§7.1) and Accelerator major HLS modes (§15).

In Kiwi, roots may instead or also be specified using dot net attributes similar to `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.

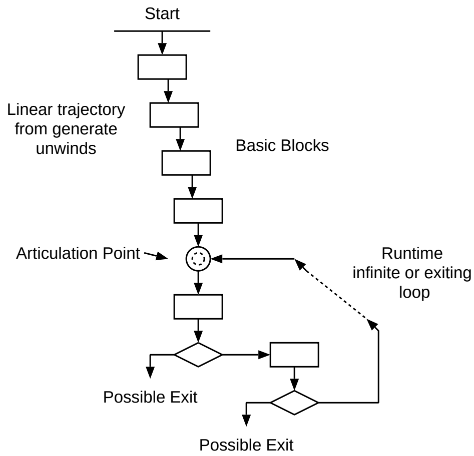


Figure 3: Front End Control Flow after Unwind: Lasso Diagram.

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.

The `-root` command line flag is an alternative to the HardwareEntryPoint marker. Supplying this information on the command line is compatible with multiple compilation appoaches 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 peform 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, in Kiwi 1, 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 3 illustrates the general pattern. The articulation point is called the **end of static elaboration** point. The point selected is the first branch target that is the subject of a conditional branch during an interpreted run of the thread or the entry point to the last basic block encountered that does not contain a Kiwi.Pause() call.

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

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

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

Provided the heap allocator internal state is modelled in the same way as other variables, no further special attention is required. In this fragment the heap values are compile-time constants.

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

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

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

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

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

6 Supported Libraries Cross Reference

We have started documenting our library coverage in this section.

6.1 System.Collections.Generic

Currently (August 2016), none of the standard collection types, such as Dictionary, are provided in the distro. They are now arriving ... Summer 2017.

6.2 Standard System.Math Library

Implementations of double-precision square-root, log, exp, sine, cos and tan are all being added Summer 2017 now that incremental compilation is working. These components are in the `ip0` library in Verilog RTL form with IP-XACT wrappers. You may substitute your own if you wish. A single-precision set might be useful. Dotnet perhaps does not have them in single-precision form?

6.3 Parallel For Loop

```
// Execute N copies of f in parallel.
Parallel.For(0, N, i => f(i));
```

See ParFor1 example

The CSharp compiler maps Parallel.For to a call of System.Threading.Tasks.ParallelLoopResult.

An implementation of this in Kiwic.cs maps it via CSharp backdoors to the Xfork Xjoin forms supported by bevelab as part of the synthesisable HPR imperative language subset. They are turned into XRTL by bevelab along with everything else. Note diosim cannot currently (3Q17) support Fork and Join so a recipe that bypases bevelab on the way to diosim will fail.

6.4 FU Redirects, Autoloads, Fenced IP and Swaps.

For many standard dot net libraries, the normal (mono or MS) release of them is not suitable for use with KiwiC. So KiwiC needs to substitute alternate versions. There are two main forms of alternative: they may be implemented in C# (or CIL assembler) themselves, or may be RTL modules from an IP library. So, on a case-by-case basis, the decision for either a high-level (C#) swap or a low-level (RTL) swap needs to be made.

There are three ways a stateless FU (IP-block) such as Math.Sqrt can be deployed:

1. Inlined. The C# src code for the sqrt is inlined by KiwiC and the ALUs it uses are borrowed from or shared with other parts of the application.

2. Fenced. The square-root FU is compiled to an RTL module containing its own private resources, the FU is instantiated one or more times and these FU instances are shared as per other stateless FUs and and ALUs.

3. Swapped. Again an RTL module is instantiated and ringfenced, but a back-end substitution is made where the body of the C# implementation is ignored and a foreign RTL implementation of the SQRT FU is deployed and sharable.

6.5 System.Random

`System.Random` is automatically mapped in the `library-substitutions` to `KiwiSystem.Random`. Since this is a different generator the sequence of numbers created is different. So you will get different sequences of random numbers under WD from RTL_SIM and FP.

To get the same behaviour for all of WD, RTL_SIM and FP, please directly use `KiwiSystem.Random` instead of `System.Random`.

```
KiwiSystem.Random dg = new KiwiSystem.Random();
```

The redirect can be disabled using `-kiwic-library-redirects=disable`.

The search path for redirected libraries is based on the HPRLS environment variable, but if this is not set, it relative paths names with respect to the `lib` folder where the `kiwic.exe` binary is executing from.

Describe kdistro/support ... which was generated from kiwipro/kiwi/userlib and which is dynamically searched for autoloaded libraries whose names have often redirected names ...

6.6 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
```

6.7 System.Threading.Barrier

... text missing ...

6.8 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
```

6.9 System.BitConverter

6.10 System.String.ToCharArray

- convert a string to an array of chars. Chars are 16 bits wide in dotnet. They are tagged shorts and do not behave quite the same as shorts for various output options.

6.11 System.IO.Path.Combine

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

6.12 TextWriter

6.13 TextReader

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

6.14 FileReader

6.15 FileWriter

6.16 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.

Kiwi supports `Thread.Create()` and `Thread.Start()`.

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

```
    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:

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

Exiting threads can be joined with code like this:

```
  ...  missing ...
  Thread.Join(); // not tested currently.
```

The `lock` statement of C# is supported (but as per C#, not in static methods). This is a convenient paradigm for where the flow control of the critical region has pre and post dominating points. It wraps code in `Monitor.Enter(this)` and `Monitor.Exit(this)`.

For more-complex control flow we must use the lower-level primitives, as per posix. These are `Monitor.Wait()` , `Monitor.Enter()` , `Monitor.Exit()` , `Monitor.Pulse()` and `Monitor.PulseAl`
.

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.

```
      lock (this)
         {
```

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

`System.Threading.Interlocked.Exchange` and `System.Threading.Interlocked.CompareExchan`
...

The atomic add operating `System.Threading.Interlocked.Add`, is ...

The atomic increment and decrement operators `System.Threading.Interlocked.Increment` and `System.Threading.Interlocked.Decrement` are implemented on top of the add primitive.

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

6.16.1 Sequential Consistency

KiwiC does not currently support fine-grained store ordering. Where a number of writes are generated in one major cycle (delimited by hard or soft pauses) the writes within that major cycle are freely reordered by the restructure recipe stage to maximimse memory port throughput. However, KiwiC already maintains ordering in PLI and other system calls, so extending this preservation to remotely-visible writes can easily be added in the near future.

Write buffers and copy-back caches may also be instantiated outside the KiwiC-generated code in uncore structures that are part of the substrate for a given FPGA blade. KiwiC has no control over these.

We are writing a paper that explores this space.

C# provides the `Thread.MemoryBarrier()` call to control memory read and write re-ordering between threads... but in the meantime you have to use `Kiwi.Pause()` to ensure write ordering.

6.16.2 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

```
//tinytest0.cs(16,26): error CS0677: ‘tinytest0.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:

```
[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 .NETassembly 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 [3]. This compiler accepts CIL (common intermediate language) output from either the .NETor Mono C# compilers and generates Verilog RTL.

7.1 Kiwi.Remote() Attribute

Purposes:

1. RPC (Remote-Procedure Call) Interface Between Compilations.
2. Addressing multi-FPGA accelerators.
3. Marking up given methods to be remotely callable.
4. Reducing complexity in classical HLS sequencers.

Object-oriented software sends threads between compilation units to perform actions. Synthesisable Verilog and VHDL do not allow threads to be passed between separately compiled circuits: instead, additional I/O ports must be added to each circuit and then wired together at the top level. Under manual design partitioning, we mark up methods that are to be called from separate compilations with a `Kiwi.Remote()` attribute. (Automatic partitioning discussed in TBD ...)

Methods marked as remote can be used in two different ways by a parent compilation. They may either be called anonymously (in the same way that a floating-point adder will be instantiated and used without any high-level view of the FU instance name) or else a shared-resource name (or indicator: SRI) may be used that identifies a particular instance and an OO (aka TLM) call is made. (In the HPR L/S back end, the first argument to a TLM call is an instance identifier (SRI) in the same way that the first argument in a C++ or CSharp method call is the object handle.)

The argument to `Kiwi.Remote("...")` is a string that contains a list of semicolon-separated key/-value pairs. Keys avaliable include all of the so-called fsems and these specific ones:

- **exernally-instantiated**: true/false
- **protocol**: HSIMPLE, HFAST and perhaps others...
- **overloaded**: true/false — this needs to hold when more than one method with the same name is to be disambiguated by signature.
- **searchbymethod**: true/false
- **posted**: true/false — if this holds, the caller need not wait for any response.
- **reftran**: true/false — if this holds, the caller need not invoke the method in a context where it has the to hand the result of a recent call with the same arguments.
- **mirrorable**: true/false — if this holds, the callee is considered to be interchangable with other instances of the same component, as determined by load balancing and area tradeoffs. This should not nornally be asserted for RAMs and other components that contain local state.

```
[Kiwi.Remote("protocol=HFAST1;externally-instantiated=true)]
public return_type entry_point(int a1, bool a2, ...)
{ ... }
```

If fs_inhold is specified, the caller will maintain the input values throughout the operation, but the default is that the input data is likely to be removed straight a fter the input handshake (fs_inhold=false).

Fsems are [“NONREF”; “EIS”; “YIELDING”; “MIRRORABLE”; “INHOLD”; “OUTHOLD”; “ASYNCH”]

The instance that provides the remote method may be internally or externally instantiated in the generated RTL. External mode requires the system integrator to instantiate the component and to wire it to additional formals to the current module. Internal mode has the instance generated inside the current module and wired up as part of the current module synthesis.

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 originally implemented protocol, HSIMPLE, was 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. Today we use HFAST1 mainly (which can be a form of AXI-3 streaming when all 4 handshake nets are used).

Over two runs, 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.

Where a design contains Remote-marked methods, KiwiC must be invoked more than once. For each run KiwiC must deduce or be instructed the root for the compilation: i.e. whether to compile the remote method body/bodies or the caller. This can either be done explicitly with the `-root` command-line flag or else inferred depending on whether there is `HardwareEntryPoint` as part of the current compilation. The explicit method must be used when there are three or more layers of incrementatal compilation.

Deduced compilation root procedure (text needs revising...):

1. Mark a static method (or several of them) on a class with the Remote attribute. (Actually the static restriction is now relaxed, but the method is treated as though static nonetheless.) Place this code in a .cs file that does not also contain the top-level for the compilation: i.e. one that does not contain a `HardwareEntryPoint` attribute in any of the .exe or .dll files read by KiwiC.
2. Compile that class to a `.dll` file and compile that with KiwiC. The result is a `.v` and some `.xml`. This resulting RTL will be the server — an implementation of the method. It is described in some generated IP-XACT files whose name starts with 'AUTOMETA'.
3. For this next step, a full implementation of the method(s) is not needed (they can have empty method bodies), but identical signatures must be present somewhere in the .CIL code read in for typechecking. Compile the class again (or a stub alternative with null body) but this time with a thread that invokes the method. KiwiC can either read in the IP-XACTfiles from a previous run (on the IP`-ip-incdir` search path) or else regenerate them from the encounted implementation (stub or otherwise). The resulting RTL will be the client.
4. To use the result, combine the RTL from the two runs for an RTL simulation or FPGA build, either manually or using HPR System Integrator.
5. (To package up the result for long-term library use, it is perhaps best manually rename the files without the 'AUTOMETA' prefix and perhaps the attribute squirrelling suffix. The prefix denotes automatically generated files that are likely to be regenerated, whereas without it the files are intended for more-persistent deployment. The prefix is not put on the IP-XACT component definition file, just in those it references. You will have to similarly delete such sub-strings from inside IP-XACT $\dot{x}$ml descriptions too, perhaps using 'sed'.)

One can also envision leaf modules in the design hierarchy making upcalls to parents, but this is not currently implemented in Kiwi. Yes it is, sort of, via 'externally-instantiated' markup, where the callee is outside the generated RTL module structurally. Please explain further.

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

```
    // Note four-phase is old, predating HSIMPLE - we should now be
    // using HFAST1 connection to NoC etc..
    [Kiwi.Remote("protocol=HFAST")]
    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 + `` ``);
       }
       Console.WriteLine(`` Test 10 finished.'');
    }
}
```

See test19 and test67 in the regression suite and the demo on this link

`http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/timestable-demo/rpc.html`

7.1.1 Referentially Transparent and Mirrorable

The following two attributes are commonly used together when a simple library function such as `sqrt` is declared.

The '`reftran=true`' designation is an assertion to the Kiwi toolchain that the method is referentially transparent. This strictly means that the method will always give the same result for the same argument. To KiwiC it is an assertion that the method does not have to be called if the result is not needed and that calling it more times than would happen in the WD execution environment.

The '`mirrorable=true`' designation is an assertion to the Kiwi toolchain that more than one instance of the called component can be deployed by the restructure and/or HPR System Integrator parts of the tool chain.

7.1.2 Remote Method Overloading

As well as overrides, OO languages like C# support method overloading. KiwiC supports method overloading in general. Alternative definitions of an overloaded method differ in terms of their arity and argument types. They may also have method-generic type variables that can be instantiated with different types.

Invokation of an overloaded method defined and called within a single compilation is handled by the lookup methods that match the C# types as normal. No unexpected behaviour needs to be considered.

But an issue related to method overloading arises with incremental compilation under KiwiC. The problem is akin to the C++ linking problem with method overloads: a separate low-level identifier is needed for each overloaded definition in the generated object code. The solution is to squirrel the arity and argument types into the name of the generated object. For instance, a separately-compiled RTL module providing mathematical functions such as `sqrt()` and `exp()` will most likely provide definitions for several precisions. Each definition needs a separate name.

Where a method is to be remotely called and more than one definition of it is to be provided. The `overloaded=true` setting must be added to the `Kiwi.Remote()` markup in both the actual definition of the method and any stub that is used as a proxy by the caller. This causes the generated method's name to be extended with an argument type squirrel. The modified name will be visible in the IP-XACT metafiles and concrete outputs in RTL and SystemC etc..

Alternative overloads of the same method must be provided in a common CIL assembly (a single .dll file). (All parts of a C# partial class definition must be within the one assembly). Currently KiwiC compiles all Remote marked overloads and puts them in the same RTL file which will contain one RTL module for tha assembly with disjoint terminals for the different methods. If not all of the methods are used in a given application, which is typically the case, the unused variants will be removed outside the Kiwi toolchain by subsequent RTL tools owing to their output terminals being disconnected.

Note: methods of the same name in different classes have unique hierarchic and flattened names and are not affected by overloading considerations.

7.1.3 Remote Method Performance

Invoking a remote method in blocking style stalls the sequencer of the calling thread. Parallelism is thereby lost. Asynchronous dispatch using `Kiwi.Remote()` provides a non-blocking interface but the result must be void at the moment. TODO: notes on integrating with the C# asynchronous delegates and `await` ...

7.2 Asynchronous Invokation

await keyword

C# now has the ‘`await` keyword. We have done some experiments with support for this ... whitepaper in preparation ...

7.3 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.

7.4 Hard and Soft Pause (Clock) Control

This section needs joining up with the repeated copy elsewhere in this manual!

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 schedulling mode for a thread can be restored by making the thread calls `Kiwi.SetPauseControl(autoPauseControl)`.

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)`.

The `Kiwi.Pause()` primitive may be called with an argument that is an integer denoting a combination of built-in flags. This enables per-call-site override of the default pause mode.

7.5 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,
    // the user can make a manual call to this at the place where they
    // think elaboration should end for confirmation.
    // This will be just before the first Pause in hard-pause mode or
    // undecidable name alias or sensitivity to a run-time input etc..
}
```

7.6 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.

```
public static void Unroll(int a)
 {  // Use these unroll functions to instruct KiwiC to subsume a variable (or variables)
    // during compilation.  It should typically be used with loop variables:
    //
    // for (int cpos = 0; cpos < height; cpos++)
    //   { Kiwi.Unroll(cpos);
    //     ...
    //   }
 }

 public static void Unroll(int a, int b)
 { // To subsume annotate two variables at once.
 }

 public static void Unroll(int a, int b, int c)
 { // To annotate three variables.
   // To request subsumation of more than three variables note that
   // calling Unroll(v1, v2) is the same as Unroll(v1 + v2).  I.e. the
   // support of the expressions passed is flagged to be subsumed in total or
   // at least in the currently enclosing loop.
 }
```

7.7 Elaborate/Subsume Manual Control

OLD: Ignore this paragraph from 2015 onwards.

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:

```
    [Kiwi.Elaborate()]
    bool empty = true;
```

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

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

7.8 Synchronous and/or Asynchronous RAM Mapping

See §8.

7.9 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
```

Q. Can I pass constant expressions into my attributes, such as `Kiwi.HwWidth()`, to make highly-parameterisable code? When do the constant expressions get evaluated? Can values set via `Kiwi.RtlParamet`
be used within hardware width expressions attributes?

7.10 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 three examples show input and output port declarations, where the first two have their input and output have their width specified by the underlying type and the last by an explicit width attribute.

```
[Kiwi.OutputBitPort("done")] static bool done;
[Kiwi.InputPort("serin")] static bool serialin;
[Kiwi.HwWidth(5)] [Kiwi.OutputPort("data_out")] static byte out5;
```

KiwiC can create obscure names if these I/O declarations are not in a top-level class. So, the contents of the string are a friendly name used in output files.

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

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

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

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

The following examples work

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

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

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;
```

7.11 Wide Net-level Inputs and Outputs

The C# language supports primitive data word lengths up to 64 bits. Sometimes, for high-performance interfaces, we require net-level I/O busses that are wider than this.

C# structs are valuetypes that may contain more bits and, being valuetypes, they are passed by value.

This can be achieved by attaching the net-level attribute markups to arrays.

Coding style 'lostio'. See test51 in the regression deck.

Note: this style stopped working in about 2010 but is just being made to work again (Dec 2016).

```
  // Wide input and output, net-level I/O.
  [Kiwi.InputWordPort("widein")]
  static int [] widein = new int [8]; // 32 byte parallel input

  [Kiwi.OutputWordPort("wideout")]
  static int [] wideout = new int [8]; // 32 byte parallel output

  [Kiwi.HardwareEntryPoint()]
  public static void Dut()
  {
    for (int p=0; p<widein.Length; p++)
    {
      wideout[p] = widein[p];
    }
  }
```

Coding style using structs ... being fixed ...

```
public class WideWordDemo
{
  // Demo of wide input and output words.
  // You may want to overload your arithmetic operators to handle such constructs?

  // Note: this is a C# struct, not a C# class.  Structs behave like valuetypes.
  public struct widenet
  {
    public ulong word1, word0;
  }

  [Kiwi.OutputWordPort("normal")] public static ulong normal;

  [Kiwi.OutputWordPort("word128_in")] public static widenet word128_in;
  [Kiwi.OutputWordPort("word128_out")] public static widenet word128_out;

  static void valuetype_test(widenet bof) // Structs are passed by value, but call-by-value still gives a local
  {
    bof.word0 += 1; // Falls foul of operating on formals if passed by value?
  }
  ...
}
```

7.12 Clock Domains

You do not need to worry about clock domains for general scientific computing: they are only a concern for hardware interfacing to new devices. **KiwiC** generates synchronous logic. By default the output circuit has one clock domain and requires just one master clock and reset input. The allocation of work to clock cycles in the generated hardware depends on the current 'pause mode' and the `bevelab-soft-pause-threshold` *unwind budget* described in [3] and the user's call to built-in functions such as '`Kiwi.Pause`'.

Terminal names `clk` and `reset` are automatically generated for the default clock domain. To change the default names, or when more than one clock domain is used, a '`Kiwi.ClockDom()`' attributes 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 negative edge clock is generated if the third argument is provided `"clockPolarity=neg"`.

Mechanisms for clock enables, overring the default reset synchronicity and clock enable guard will be supported soon, using further colon-separated properties inside the third argument.

Each thread, hardware entry point or remote-callable method has its own, so-called 'directorate' and the clock domain properties are part of a directorate. Only one directorate is allowed for a thread, but that thread may call methods called from (shared with) other threads: their bodies get in-lined in the elaboration of the thread..

7.13 Remote

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

```
[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 originally implemented protocol, HSIMPLE, was 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 have now been added to the compiler.

A remote-marked method is either an entry point or a stub for the current compilation. This is inferred depending 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).

In the regression suite, test19 is an old example and new examples calling to maths modules are being added...

7.14 Elaboration Pragmas - Kiwi.KPragma

```
public static int KPragma(bool fatalFlag, string cmd_or_message)
```

```
public static int KPragma(bool fatalFlag, string cmd_or_message, int arg0)
public static int KPragma(bool fatalFlag, string cmd_or_message, int arg0,
```

Kiwi.KPragma with first argument as Boolean true can be used to conditionally abend elaboration. This behaves the same way as `System.Diagnostics.Debug.Assert` described in §7.15 except that a user-defined error code can be passed in arg0.

Note, you may want to use Trace.Assert instead and to `'export MONO_TRACE_LISTENER=Console.Error'`

With the Bool false, it is used to log user progress messages during elaboration.

Kiwi.KPragma calls present in run-time loops can be emitted at runtime using the Console.WriteLine mechanisms (in the future - current release ignores them beyond elaboration).

Kiwi.KPragma calls with magic string values will be used to instruct the compiler, but no magic words are currently implemented.

7.15 Assertions `Debug.Assert()`

Sometimes it is convenient to generate compile-time errors or warnings. Othertimes we want to flag a run-time abend, as per §2.2.

Typically you might want to direct flow of control differently using the function `Kiwi.inHardware()` and to abort the compilation if it has gone wrong. Call the function `Kiwi.KPragma(true/false, ‘‘my mes` 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

```
int length = (int)fileStream.Length;  // get file length - will be known at runti
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 st
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 ...

7.16 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

```
[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.wunitl`'. 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 [4] can be used, which implements all non-blocking paths through the product of a pair of such interface automata.

7.17 RTL Parameters

Sometimes it is helpful to generate an RTL file from a single run of KiwiC that is to be instantiated many times. Each time will use a different run of the FPGA logic synthesiser tools. It is handy to be able to pass in a constant at the logic synthesis time that might be different for each instance.

A good use-case example is when link editing a number of components into a single entity that will use a shared memory bank. Each component wants its datastructures at a different address in the memory bank. The HPR System Integrator computes base addresses and provides a parameter overide for the KiwiC-generated logic.

Within C#, in order to read in a logic-synthesis constant we use the Kiwi `RtlParamter` attribute as in:

```
class RTLprams1
{
  [Kiwi.RtlParameter("rtl_pram1", 1001)] public static int rtl_pram1 = 1001;
  [Kiwi.RtlParameter("rtl_pram2")] public static int rtl_pram2;
  ...
}
```

Parameters of this nature should generally have the type `int`. For well-formed RTL, those with default values should preceed those without.

This leads finally to an RTL module with signature such as

```
module DUT #(parameter rtl_pram1=32'sd1001,
             parameter rtl_pram2)
            (input clk,
             input reset,
             ...);
...
```

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. Arrays that are only written at each location precisely once with a constant value for each location are treated as read-only look-up tables (ROMs).

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. On 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)

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**, also known as **LUT 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, LUT or slice RAM.
3. **Block RAM**: As well as I/O, flip-flops and LUTs, all modern FPGAs also provide BRAMs (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, load/store ports (I/O pins) are used to connect to external, standard RAM parts or memory resources.

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 fouth 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 implements the protocol to operate the RAM or generate AXI transactions destined for a complex memory subsystem.

Table 2: RAM forms supported by FPGAs.

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 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 `bondout-loadstore-port-count` 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 use a bondout load/store port for memory access (previously known as an off-chip resource), such as TCM, ZBT or cached DRAM. The size in bytes will depend on the word width of that array. The `Kiwi.OutboardArray()` attribute allows manual override.

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 may not be the default for its size. The argument is the number of clock cycles of latency for read. When the argument is omitted it defaults to unity - the standard value for FPGA BRAM.

The `CombSRAM(n)` attribute indicates that an array is to be mapped to an on-chip RAM type that may not be the default for its size. Only small RAMs are mapped to register files or LUT RAM with combinational (zero cycle) read, but this attribute will force any sized RAM to be mapped that way. Note that LUT RAM is very inefficient in FPGA area terms and should be avoided for larger structures of 32 words or more.

TODO: describe PortsPerThread and so on... these control multi-port RAMS and how the number of external ports is configured.

Kiwi has a scheduller 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 runtime 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.

External instantiation is when a component that could logically be an instance within the current module is instead instantiated outside the current module and the current module thereby gets additional I/O nets for connecting to the external instance. Those nets would normally just be local to the current module.

8.1 On-chip RAM (and ROM) Mirror, Widen and Stripe Directives

To increase memory performance, three techniques are generally available (these techniques may not all be sensible for off-chip RAM resources). All of these increase the number of data bus wires to RAMs, thereby increasing available throughput.

1. A `Kiwi.Mirror`**(n)** directive applied to a C# array instructs KiwiC to make multiple copies of the RAM or ROM. This is most sensible for ROMs since all copies of a RAM must be updated with every write.
2. A `Kiwi.Widen`**(n)** directive applied to a C# array instructs KiwiC to pack n words into a single location. This multiplies the data bus width by this factor. For RAMs, a RAM with laned writes may be needed. This will boost performance where an aligned group of n words is commonly read and written at once.
3. A `Kiwi.Stripe`**(n)** directive applied to a C# array instructs KiwiC to allocate n multiple RAMs or ROMs each of $1/n^{th}$ the size with every n^{th} word placed in each of them.

(In order to pack multiple user arrays into a single RAM on the FPGA, additional directives are needed. Not described here currently.)

8.2 ROMs (read-only memories) and Look-Up Tables

Most FGPAs support ROMs. ROM inference is a variation on RAM inference. Combination and registered ROMs are both commonly used, depending on size. KiwiC will deploy ROMs with pipeline latency of 1 when the size in addresses exceeds the size set by `res2-combrom-threshold`.

ROM inference in KiwiC can be turned off with flag `repack-to-rom=disable` in which case RAMs are commonly generated and initialised with the ROM contents after the run-time reset. But, when ROMs are present, they are manifest in the generated Verilog RTL as arrays that have their only write operations embodied in Verilog `initial` statements that install the fixed data.

ROMs can sometimes usefully be mirrored. The `Kiwi.Mirror(4)` attribute can be applied to individual array instances to mirror them.

```
[Kiwi.Mirror(4)]
static readonly uint[] htab4 =
   { 0x51f4a750, 0x7e416553, 0x1a17a4c3, 0x3a275e96,
     ... many more entries ...
   };
```

Or else the command line flag `repack-to-rom=4` can be added, which would replicate all ROMs up to a factor of 4, but the additional copies would not be generated if they cannot usefully be used.

8.3 Forced Off-chip/Outboard Memory Array Mapping

The `Kiwi.OutboardArray()` attribute forces that an array is to be mapped to a region of external memory instead of being allocated a private array (BRAM memory) inside the current compilation. Large arrays are placed off chip in this way by default without using an attribute. (Large is determined by comparing `res2-offchip-threshold`). 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.

With a string argument provided, this is the name of a load/store port name or DRAM bank name (user or logical name). This provides a manual load-balancing override for the memory subsytem allocator.

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, (unit arg `Kiwi.OutboardArray()`) then a default of 'bondbank0' is automatically supplied. Therefore, without using any attributes, all large arrays are mapped into consecutive locations of a memory space called 'bondbank0'.

Note: the older bondbank0 is now renamed as bondbank0.

TODO: profile-directed feedback will balance up the ports in the future.

Using the special argument '-onchip-' the `Kiwi.OutboardArray("-onchip-")` attribute forces that an array is not offboard regardless of size. Clearly this may result in a design that is unsuitable for the target technology.

8.4 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:

```
*-----------+----------+--------+--------+-------+-----------*
```

```
| Name      | No Words | Awidth | Dwidth | Lanes | LaneWidth |
*-----------+----------+--------+--------+-------+-----------*
| loadstor1 | 4194304  | 22     | 256    | 32    | 8         |
*-----------+----------+--------+--------+-------+-----------*
```

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

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

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

bondout-loadstore-lane-width 8 LOADSTORE lane width
```

A lane is normally 8 bits wide. But whatever width a lane is, we have a 'lane-addressed' address bus. The bottom (log2 no_lanes) address bits will then always be ignored (and stripped by logic synthesiser tool).

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
*-----------------+-----------+-------+-----------+-----------*
| Resource        | Base      | Width | Length    | Portname  |
*-----------------+-----------+-------+-----------+-----------*
| D8US_AX/CC/SOL  | 0x1312d02 | 32    | 0x989680  | bondbank0 |
| D16SS_AX/CC/SOL | 0x0       | 32    | 0x1312d02 | bondbank0 |
*-----------------+-----------+-------+-----------+-----------*
```

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. AXI is now being added shortly to KiwiC, replacing BVCI, but there are also some AXI components in the support and subtrates library. Including an HFAST to AXI protocol bridge and AXI master and slave shims for the Zynq substrate for CPU interaction and DRAM access.

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.

KiwiC will shortly be enhanced to issue prefetch bus cycles on off-chip RAMs. These are appropriate for cached DRAM and sometimes appropriate for uncached off-chip RAMs. They serve no useful function for SRAM (static RAM), whether on-chip or off-chip, owing to its uniform access latency.

8.4.1 HSIMPLE Offchip Interface & Protocol

The implementation of HSIMPLE within KiwiC was a low performance. It will be deleted soon as we converge to HFAST and AXI protocols for everything.

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.

```
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.

8.4.2 HFAST Bondout (Offchip) Interface & Protocol

HFAST1 is our primary protocol for load/store ports to DRAM. It has half-duplex and simplex variants. Protocol adapators to AXI4 and AXI4-Lite are in the distribution.

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,             // Any posedge clk with overlap of opreq and opack starts a ne
input hf1_dram0bank_ACK,               // Ack acknowledges the last request is complete.
output reg hf1_dram0bank_RWBAR,        // 1=read, 0=write on request active clock edge.
output reg [255:0] hf1_dram0bank_WDATA, // For write, data to be written, valid on request active cloc
output reg [21:0] hf1_dram0bank_ADDR    // Address, valid on request active clock edge.
input [255:0] hf1_dram0bank_RDATA,      // Read result, valid on ack cycle.
output reg [31:0] hf1_dram0bank_LANES,  // Byte lane qualifiers.
```

A half-duplex port has RWBAR. A storeport has no RDATA and a loadport has no WDATA or LANES. LANES are only present if there is more than one lane per word. There is no full-duplex port: instead one uses a pair of simplex ports.

IP-XACT definitions for all variants are in the Kiwi distribution. Their names follow a scheme such as `HFAST1_M_RONLY` which denotes an outstanding transaction count of 1, master side interface, (simplex) write only.

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 only and d
  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 1=precharge, 0=activate
  input [2*dwidth-1:0]          ddr_wdata, // The wdata and rdata busses are here twice their width
  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 edges instead
  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 this to
   // 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 banks = 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 at 32 by
             With a 32 bit data bus (64 after doubling for DDR) this requires 4 clocks to transfer th
   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.

   // Front-side interface
   input                             fs_rwbar,
   output reg [noLanes*laneSize-1:0]  fs_rdata,
```

```
input [noLanes*laneSize-1:0]       fs_wdata,
input [addrSize-1:0]               fs_wordAddr,
output                             fs_oprdy,
input                              fs_opreq,
output reg                         fs_ack,
input [noLanes-1:0]                fs_lanes,

// Back-side interface
output reg                         bs_rwbar,
input [noLanes*laneSize-1:0]      bs_rdata,
output reg [noLanes*laneSize-1:0] bs_wdata,
output reg [addrSize-1:0]          bs_wordAddr,
input                              bs_oprdy,
output reg                         bs_opreq,
input                              bs_ack,
output reg [noLanes-1:0]           bs_lanes
);

parameter dram_dwidth = 256;          // 32 byte DRAM burst size or cache line.
parameter laneSize = 8;
parameter noLanes = dram_dwidth / laneSize; // Bytelanes.
```

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.

8.4.3 BVCI Offchip Interface & Protocol

Text missing. Not being used.

8.5 AXI and HFAST-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

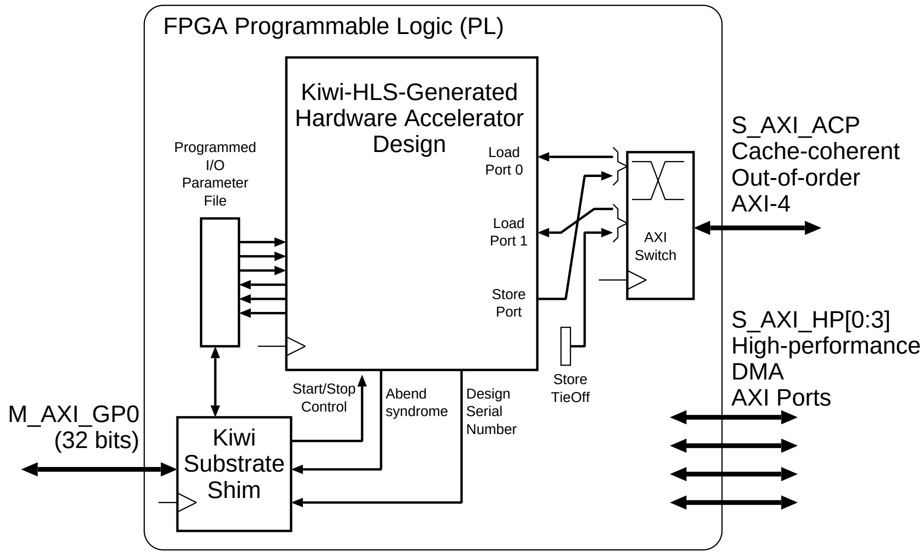


Figure 4: Typical connection of load/store ports to DRAM via AXI (Zynq Example).

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 HFAST interfaces. In the fullness of time, KiwiC will provide automated support for HFAST to AXI adaptation but currently a substrate that manually matches the number of load/store ports is required. Currently they must be instantiated manually (but the new recipe stage that inokes HPR System Integrator should fix that soon). The easiest way is to import the Kiwi design into a GUI-based schematic editor that understands IP-XACT and use a few mouse clicks to instantiate the required protocol convertors and so on. However, the SoC Render extension can soon replace this.

The main substrate shim is boiler-plate RTL code that connects to the `M_AXI_GP0` programmed I/O bus for simple start/stop control and parameter exchange. It is recommended that every design compiled has a serial number hard-coded in the C# source code and that this is modified on every design iteration. The first function of the substrate shim is to provide readback of this value.

The other features of the shim are starting and stopping the design and collecting abend codes. Sources of abend are null-pointer de-reference, out-of-memory, divide-by-zero, user assertion failure, and so on.

A Kiwi design that makes access to main memory will have a number of load/store ports. These can be half-duplex or simplex. Simplex is preferred when main memory is served over the AXI bus, as in the Zynq design. (Of course there may be a lot of BRAM memory in the synthesised design itself, but that does not appear on this figure.) Simplex works well with AXI since each AXI port itself consists of two independent simplex ports, one for reading and one for writing.

In the illustrated example, the design used three simplex load/store ports. These need connecting to the available AXI busses hardened on the Zynq design and made available to the FPGA programmable logic. The user has the choice of a cache-coherent, 64-bit AXI bus that will compete with the ARM cores for the L2 cache front-side bandwidth, or four other high-performance 64-bit AXI busses that offer high DRAM bandwidth. These four are not used in the example figure.

Each KiwiC-generated load-store port is an in-order unit, like a load or store station in an out-of-order processor. By multiplexing their traffic onto AXI-4 busses, bus bandwidths are matched and out-of-order service from the DRAM system is exploited.

Each load/store port in the generated RTL has is properly described in the IP-XACT rendered by KiwiC that describes the resulting design. When this IP-XACT is imported into a design suite, manual wiring of the load/store ports to the AXI switch ports can be done in a schematic editor. (Approaches to automate this stage are ongoing.)

Note that KiwiC as of December 2016 generates so-called HFAST ports, that are either half-duplex, loadonly or storeonly. These are what was described in the IP-XACT. The user also has to manually instantiate, in the schematic editor, little protocol convertors that come with KiwiC and which convert HFAST variants to AXI variants for connection to the vendor-provided AXI switch blocks.

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

8.6 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.

'`bondout-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 '`bondout-loadstore-port-lanes`' and '`bondout-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).

8.7 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.

```
// (C) Xilinx 2009.  Single-Port B-RAM with Byte-wide Write Enable: Read-First mode
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// 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,
   input [NB_COL*COL_WIDTH-1:0] di,
   output reg [NB_COL*COL_WIDTH-1:0] do);

   reg [NB_COL*COL_WIDTH-1:0] RAM [SIZE-1:0];

   always @(posedge clk) begin
     do <= RAM[addr];
     end

   generate
     genvar i;
     for (i = 0; i < NB_COL; i = i+1) begin
       always @(posedge clk)
       if (we[i]) RAM[addr][(i+1)*COL_WIDTH-1:i*COL_WIDTH] <=
                             di[(i+1)*COL_WIDTH-1:i*COL_WIDTH];
       end
   endgenerate

endmodule

// Single-Ported Block RAM with registered output Option
// Please note that XST infers distributed RAM or B-RAM based on the size.
// For small RAMs, you may need to use ram_style constraint to fore the use
of B-RAM.

module TWO_CYCLE_READ_BRAM(
  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 are also the dual-ported equivalents of these styles, supported by both Xilinx and Altera.

8.8 Dual-port Block RAMs

See demo test50.

The FPGA libraries contain (typically) dual-port BRAMs. Where an array is small enough to instantiated as an FPGA on-chip BRAM (block RAM), and overrides are not applied, then such a BRAM will be used. Both Xilinx and Altera provided FPGAs with on-chip, dual-ported BRAMs with synchronous read latency of one cycle.

Such BRAMS are atomatically used for sharing data between up to two threads. Threads can also shared data via a scalar variables. Kiwi supports any number of threads reading or writing shared scalar variables but for BRAMs there are technology restrictions.

What if I want to get increased RAM bandwidth by allocating both ports of a BRAM to the same thread?

By default, KiwiC will use one port on an SRAM for each thread that operates on it. However, by setting the `PortsPerThread` parameter or attribute to greater than one then greater access bandwidth per clock cycle for each thread is possible. Example needed.

8.9 Other multi-port RAMs

If three threads operated on the shared memory, KiwiC could generate an instance of a triple-ported SRAM module but this would likely not be found in a technology library when logic synthesis tools were applied.

Instead, the 'off-chip' approach needs to be used. This works efficiently even for small BRAM subsystems, but additional wiring is needed outside the KiwiC-generated RTL. The HPR System Integrator aims to provide this service.

The approach is

1. Mark the array as off-chip. (Please see example ...missing).
2. KiwiC will then generate as many off-chip load and store ports as is requested for each thread by the `PortsPerThread` mechanism.
3. The outside logic will instantiate arbiters as needed to connect all the ports created to a suitable memory resource. The arbiters needed, for HFAST and AXI, are in the bundled technology library.

... we need to add a little more explanation or forward reference here please ...

9 Substrate Gateway

There is some basic information on the Zynq substrate here: http://www.cl.cam.ac.uk/research/srg/han/hprls/o

The substrate gateway is a hardware/software boundary for use on platforms such as Zynq or others that run embedded linux with a console, network and filesystem. It has an associated protocol for providing operating system access.

9.1 Console I/O

This section will explain how to do console I/O via the substrate gateway.

We also need to explain the ReadKey situation. Windows users often put a readkey call on the end of their programs to stop Windows deleting the output straightaway ...

```
 // Keep the console window open in debug mode.
Console.WriteLine(``Press any key to exit.'');
Console.ReadKey();
```

9.2 Filesystem Interface

The basic dotnet classes for StreamReader, StreamWriter, TextReader and TextWriter are provided via the substrate gateway. Random access using fseek is also supported.

documentation incomplete ... add KiwiFilesystemStubs.dll to your compilation ... documentation for Zynq use will be added here... Satnam's windows version ... It works fine under RTL_SIM with verilator.

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/filesyste` 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 Geneome Sequencer demo for an example of file system use.

9.3 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.

10 Kiwi Performance Tuning

An HLS system can be set to optimise for

1. Performance: achieving the best execution time, aiming for maximal clock frequency and minimal number of clock cycles,
2. Area: using as little area as possible, generally at the expense of many more clock cycles,
3. Debugibility: renaming and sharing registers as little as possible and providing additional debug and trace resources for interative access.

The main parameters for tuning the Kiwi Area/Performance tradeoff, folding space over time are:

1. The `bevelab-soft-pause-threshold` parameter. The nominal range is 0 to 100 with useful values currently being between 5 and 40. A lower value tends towards more clock cycles and possibly less area. Values above 40 may lead to very long KiwiC compile time.
2. The loop unwind limits alter the amount that a loop is unwound at compile time, leading to parallelism. For instance, the `Kiwi.Unroll("COUNT~=4", lvar);` attribute added to the C# source code suggests that the loop whose control variable is called 'lvar' is unwound by a factor of 4.
3. Structural Resource Budgets: The restructure phase accepts ten or so recipe settings that limit the maximum number of structural resources, such as floating-point ALUs allocated pre thread. Smaller settings lead to smaller designs that use more clock cycles.
4. RAM thresholds: Settings such as `res2-offchip-threshold` alter the amount of block RAM allocated. This is faster than external (off-chip) SRAM or DRAM but uses more FPGA resources.
5. The setting `bondout-loadstore-port-lanes` alters the number of external memory ports used. These each operate in order, so if you have more of them and mux them externally onto separate resources or an out-of-order bus then you get more parallelism and external RAM bandwidth.
6. ALU latency: Settings such as `fp_fl_dp_div` describe the type of divider to generate. For such components you can provide your own implementations, alongside those provided in the Kiwi libraries like `cvgates.v`, and specifiy whether they are fixed or variable latency, fully-pipelined and what the fixed or expected latency in clocks cycles is.
7. Register colouring affinty: The `kiwic-colour-enable` setting alters the amount to which KiwiC reuses registers. With it disabled, the hardware is easier to inspect/debug, but many more registers are generated. An experimental, spatially-aware binder is being added to Kiwi at the moment. This will handle both registers and ALUs and gives a floorplan plot.

Commonly, the system DRAM will run at a hardwired clock frequency, such as 800 MHz. This is too fast for most current FPGA logic, Kiwi-generated or otherwise. An integer divisor of 4 or 5 typically needs to be applied to bring the logic speed below 200 MHz. Getting KiwiC to hit a target clock frequency is a common requirement ... TBC ...

10.1 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.

... Kiwi also uses the simple performance predictor from the HRP L/S library to estimate run time for a thread. This is used as a source for PlanOpt metric, when exploring the implementation space ...

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 call strings 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.

Profiles for performance prediction can be sourced from various places, including diosim, but RTL simulation is used in the following, step-by-step, example.

1. Preferably denote several waypoints in the application C# program `Kiwi.KppMark()`.
2. Generate an RTL design using KiwiC and an RTL testbench using the standard flow for your envionment, but with the following minor changes
   - OLD:Stop KiwiC generating any `$finish()` calls with the `-kiwic-finish=disable`

command line flag. NEW: We replace -kiwic-finish with -kiwife-directorate-endmode.

- Augment your C# program to make it drive a top-level net called 'finished' high at the end of simulation by declaring a satic boolean OutputBitPort and assigning true to it at the program end (you will typically also include a waypoint called FINISH at that site too.
- textually include `kpp_testbench_mon_onethread.v` in the testbench using an RTL include statement.

3. Run your RTL simulation. The included material will write out a file file called 'profile.xml' or similar. (You can also get this file from diosim without an external RTL simulator).
4. Invoke the performance predictor (hpr/kpredict.fs) using ... and you will see
5. With a suitable Makefile, you can make the web page redisplay automatically after every high-level edit ...

10.2 Phase Changes, Waypoints and Code-point Markers

A waypoint is a marker or milestone that shows a program has reached a certain point in its execution path. Where a program is remotely started and ultimately exits, a given waypoint should only be encountered once. For eternal servers with an outermost dispatch loop, a given waypoint should be encountered at most once per dispatched command.

Waypoints can be used in Kiwi Sequencer Mode for the main thread. They should not be used for child threads. In Pipelined Accelerator mode there is no control flow when II=1 and generally no signficant control flow, so waypoints should not be used.

Hardware itself does not have a start and end time. Instead, performance metrics are always comprise 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'. Each of these should be a waypoint.

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 at a phase boundary is called a **waypoint**. `Kiwi.KppMark()` dummy calls and/or Kiwi.KppMarker attributes are used to define waypoints. Each waypoint has a manually-allocated number and name and all but the last start a phase that optionally is given a name via the third argument. The numbers must be disjoint but their ordering is unimportant. The entry and exit waypoints should conventionally be called START and FINISH respectively. The program's control flow should not loop around a waypoint (except for eternal servers as mentioned above). 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).

```
// Typical pattern of waypoint markup.
Kiwi.KppMark(1, "START", "subsequent-phase-name1");
...
Kiwi.KppMark(2, "waypoint-name2", "subsequent-phase-name2");
...
Kiwi.KppMark(3, "waypoint-name3", "subsequent-phase-name3");
```

```
...
Kiwi.KppMarker(0, "FINISH");
// Note - only first two arguments are compulsory, which should be a disjoint nat
// number and needs not be strictly ascending.
```

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.

An exiting application has precisely one entry point. It has one exit point if other exits are are routed to a singleton exit point. Way points should appear once. Given expected visit ratios for each basic block, the problem is overconstrained and the frequency of visiting each way point and the singleton exit point can be inspected as a confidence indicator: they are all nominally visited once.

Note: many older designs have defined a net-level output called done or finished and assigned to it at the end of the main thread. Today we prefer to use `Kiwi.ReportNormalCompletion()` which also counts as a waypoint. We need to direct exit and so on to it...

10.3 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 lables, 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.

10.4 Debug, Single Step and Directorate Interface

There is no explict support for hardware debug currently in Kiwi, other than single stepping and PC value collection when the abend syndrome is non-zero. User logic can readily provide PIO access to major state holding RAMs [LINK TO EXAMPLE NEEDED]. Note that user variable mappings to RTL registers is typically many to one and the mapping is reported in the KiwiC.rpt file generated on each run.

The directorate interface adds the following features to the generated RTL that can be hooked up to a management CPU via the substrate gateway. They each add hardware overhead but this can be trimmed out mostly by FPGA tools when reporting resources are left disconnected.

1. Clock, Clock Enable and Reset inputs. Clock-enable is optional and can be used for single-step or other purposes.

2. Abend syndrome register - successful halt/array bounds/integer overflow/null pointer run time errors agumented with PC value or waypoint per thread.
3. Waypoint and/or PC value monitoring for each thread. Waypoint indicates not started, running, exited and various user-defined intermediate points.
4. Generic unary LED readback.
5. CPU register debug access ports: additional read/write logic is generated enabling programmed I/O access to every register (in the future).
6. Argument/result handshake and run/stop control in one of several styles:
   - startmode: self-start or wait-start;
   - endmode: auto-restart, hang or finish;
   - ready-flag: present or absent.
7. PC breakpoint control (in the future).

Nearly all FPGA blades have a some simple LED indicators connected to I/O pads. Kiwi has the concept of the ‘generic unary LEDs’ for each FPGA. Kiwi defines a uniform way to drive these and the substrate makes their values available to the host CPU, which is useful when the LEDs are in a different room or continent from the application user. They will commonly be used as a user-defined mirror of the Waypoint code (§10.2).

The directorate complexity is controlled with the recipe/command-line flag `-kiwife-directorate-style`

The single-step and breakpoint registers are/will be present with directorate style advanced `-kiwife-directo` in the future. Single-step can be achieved with suitable user logic connected to the clock-enable input for a thread. Note that clock enable is not a simple synchronous clock gate owing to the presence of pipelined components that cannot be freely stopped (such as BRAM).

Watchpoints are currently best implemented by the user in the C# source code and recompiled, or else use vendor tools like ChipScope etc..

The abend syndrome register is present with directorate styles normal and advanced `-kiwife-directorate-`

When a component is compiled as a module to be instantiated in later KiwiC runs, it needs to have an HFAST interface (when in classical HLS major mode). The HFAST interface is generated with the command line flags

```
-kiwife-directorate-startmode=wait-start
-kiwife-directorate-endmode=auto-restart
-kiwife-directorate-ready-flag=present
```

A top-level HFAST interface can be wrapped as an AXI-S interface with an externally-instantiated adaptor (from the HPRSHIMS library) that itself can be instantiated by HPR System Integrator.

The abend syndrome codes used by Kiwi in classical HLS major mode are:

- Abend code **0x00 — Not yet started code.**
- Abend code **0x01 — Normal Errorless Exit/Completion.**

- Abend code **0x01 — Unspecified Abnormal Exit/Completion**
- Abend code **0x03-0x7f — User Exit Codes** from `System.Environment.Exit(int code)`
- Abend code **0x80 — Normal operation in progress.**
- Abend code **0x81 — Paused indication during normal operation.**
- Abend code **0x83 — Suspended (breakpoint/!single step etc.)**
- Abend code **0x90 — Abend on Heap Memory Fault**
- Abend code **0x91 — Abend on Heap Memory Exceeded**
- Abend code **0x92 — Abend on Integer Divide-By-Zero**
- Abend code **0x93 — Abend on Null Pointer Dereference**
- Abend code **0x94 — Abend on Array Subscript Out-of-Bounds**
- Abend code **0x95 — Abend on C# Safe-Mode Checked Overflow**
- Abend code **0xA0 — Debug.Assert Failure**
- Abend code **0xAn — Other User Thrown Abend** from `hpr_abend()`
- Abend code **0xFF — No abend, still running.**

11 Spatially-Aware Binder

An experimental, spatially-aware binder is being added to Kiwi at the moment. This will handle both registers and ALUs and gives a floorplan plot.

Register colouring, RAM binding with memory maps and ALU binding is reported in the KiwiC report file. Only a static mapping, generated at KiwiC compile time, is used.

12 Generated RTL

Kiwi generates Verilog RTL for synthesis to FPGAby vendor tools. It can also generate SystemC and CSharp but we do not commonly use those flows at the moment and their will be some regressions.

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 libary blocks are generally provided in the following source files:

```
CV_FP_ARITH_LIB=$(HPRLS)/hpr/cv_fparith.v
CV_INT_ARITH_LIB=$(HPRLS)/hpr/cvgates.v
CVLIBS=$(CV_INT_ARITH_LIB) $(CV_FP_ARITH_LIB)
```

12.1 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 by synthesisable by FPGA tools.

Parameter overrides set the address range and word and lane width.

12.2 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, doub

CV_FP_FL_SP_MULTIPLIER

Key: FLASH=combinational.
     FLn = fixed latency of $n$ clock cycles, VL variable latency with handshake wi
        blocking while busy,
     DP=double precision,
     SP=single precision.
```

13 Incremental Compilation and Black Boxes

The IP-XACT-based incremental compilation features are being released 2Q2017.
This section of the KiwiC manual is going out of date now — please see §39 for up-to-date information.

Compiling everything monolithically does not scale to large projects. Separate and incremental compilation is needed in large projects to handle scale, component reuse, unit testing, revision control and is the basis for project management. It can also be a basis for parallelism. So, for larger designs, to manage complexity, it is always desirable to designate subsystems for separate compilation.

Also, the classical HLS approach embodied in the normal KiwiC compilation mode, in-lines all method calls made by a thread into one flat control-flow graph. KiwiC reuses ALUs and local variable registers in both the spatial and time domains, but tends to generate the largest and fastest circuit it can, subject to ALU instance count limits per thread set in the recipe. Even though FPGA/ASIC logic synthesiser tools typically re-encode the resulting state machine so that the output function is simple to decode, having more than a few thousand states becomes impractical. It makes sense for complex subsystems to be synthesised separately so that a call to them takes one state in the caller's sequencer. Any sequencer in the called component has its states shared over all calls. All standard library functions of any complexity are better handled in this way. Prime examples are trig and log functions and I/O marshalling such as ASCII to/from floating point. When these components are referentially transparent, KiwiC can deploy as many instances as it likes, guided by metrics.

Multi-FPGA designs require the logic to be partitioned between logic synthesis runs using separate RTL files. Again this requires incremental compilation and established protocols between the

FPGAs. The approach is to use HPR System Integrator to instantiate SERDES links at the FPGA boundaries, potentially multiplexing a number of services onto the available links.

The ability to use separately-compiled components also forms the basis of a **black box** import mechanism for third-party IP blocks. In principle, instantiating a black box containing third-party IP is no different from instantiating a separately synthesised Kiwi module. Example Kiwi modules are standard trig and log functions, random number generators and subsystems from user designs. The CAMs on the NetFPGA boards and the new Xilinx hardened FIFOs are typical third-party black-box componenets. See test72.

Third-party IP blocks and existing hardware interfaces are typically described in terms of net-level timing waveforms or formal specifications thereof. To exploit these components from a high-level language via HLS, wrappers need to be manually written.

```
class blackbox_wrapper_tx_demo
{
  [Kiwi.OutputWordPort("wdata")] static byte wdata;
  [Kiwi.OutputWordPort("n_wstrobe")] static bool n_wstrobe;
  [Kiwi.InputWordPort("n_rdy")] static bool n_rdy;
  [Kiwi.OutputWordPort("n_sop")] static bool n_sop;
  [Kiwi.OutputWordPort("n_eop")] static bool n_eop;

  [Kiwi.Remote("protocol=HFAST")]
  public static void SendPacket(byte [] darray, int len)
  {
    Kiwi.PauseControlSet(Kiwi.PauseControl.hardPauseEnable);
    for (int i=0; i<len; i++)
      {
n_wstrobe = !true;
n_sop = !(i==0);
n_eop = !(i==len-1);
wdata = darray[i];
        while (!n_rdy) Kiwi.Pause();
Kiwi.Pause();
      }
    n_wstrobe = !false;
  }
}
```

In some design styles, subsystems can also best be placed in a server pool with dynamic load balancing. Design-time manual control sets the number of instances generated. KiwiC will share such server instances in the time domain rather than instantiate as many as it needs (subject to ALU count limits). Note: Server pools are not currently automated within Kiwi but should involve little more than a C# library that the current KiwiC can compile.

Method designated as top-level entry points must be static. But for incremental compilation, entry points are commonly not static.

13.1 IP Integration via IP-XACT

There are several cut points in the Kiwi design flow where separately-compiled modules can be combined:

1. KiwiC will accept any number of `.dll` or `.exe` files on its command line. These will have been generated, typically, from separate invokation of the C# compiler.
2. The `Kiwi.Remote()` attribute described in §7.1 enables a designated class or method to be cut out for separate compilation with its own IP-XACT description.
3. Incremental invokation of FPGA tools is also typically possible, where some RTL files have been seen before and others are new, but is beyond the scope of this document.
4. (In principle it is possible to load and save VMs to disk (serialised in XML) and so incremental compilation at intermediate points in the opath recipe is a future option.)

Numbers 1 and 3 in the following list are relatively obvious, so we discuss only number 2.

IP-XACT is an IEEE standard for describing IP blocks and for automated configuration and integration of assemblies of IP blocks. All conformant documents will have the following basic titular attributes spirit:vendor, spirit:library, spirit:name, spirit:version. A document typically then represents one of:

1. a bus specification, giving its signals and protocol etc;
2. a leaf IP block data sheet with links to the design files;
3. a heirarchic component wiring diagram that describes a sub-system by connecting up or abstracting leaf components.

Today, the predominant protocol for interblock communication is AXI in its various forms. A block with AXI interfaces should be accompanied with an XML description using the IP-XACT schema.

13.2 The `Kiwi.Remote()` Markup

Separately-compiled modules will not share hardware resources (such as registers, ALUs or small RAMs) between them. Also, each will, in general, have its own (set of) bondout load/store port(s) for access to centralised resources such as DRAM.

We use the terms, module and IP block interchangably with the term FU (functional unit). An AFU is an application-specific functional unit. There is no technical disticintion between an FU and an AFU, but the built-in diosim simulator cannot always find suitable simulation models for AFUs whereas it is expected that all standard FUs, such as ALUs and SSRAMs have simulation models (sometimes automatically created by cvipgen.fs).

Restriction: A module for separate compilation by KiwiC cannot have free parameters at the moment, as would be used to statically set a dictionary maximum contents size for instance.[1] For example, a generic dictionary component [insert link here please] cannot be compiled, even though the

[1]We mean structural parameters in the style of Verilog. The heap base for link editing is now being added (§7.17). A separately-compiled method/function will accept its arguments (a.k.a. parameters) of course.

basic data operations on it are marked up as remotely callable with `Kiwi.Remote()` or otherwise. The dictionary example fails for these reasons:

1. the content type is typically polymorphic and hence the item size is not known when compiled to hardware standalone,
2. the capacity of the dictionary might be compile-time fixed and set via its constructor, but the constructor will not be called,
3. the dictionary component is an instance class and KiwiC can only compile static methods at the top-level.

The solution is to compile the dictionary with a minimal testbench that calls the constructor, passes in a data type and re-exports the data handling business API. Example here ... TBA

13.2.1 FU Method Groups and Instance Mirroring

An FU (IP Block or AFU) may support more than one operation or more than one port that provides the same operation. Alternatively, an FU may be mirrored, meaning more than one instance can be created to provide more processing capacity. Certain FUs may never be mirrored, especially those holding state that cannot be replicated, such as a heapspace manager.

For management purposes, the methods provided by an FU are partitioned into method groups (MGRs). Each method group has a disjoint set of nets at the hardware level (except for directorate nets such as the clock and reset). Each method group has its own IP-XACT bus interface defined in the XML component definition and each bus interface (method group) has its own pair of XML files definining its bus abstraction and RTL bus interface. There may be a third file for each method group which describes its TLM bus interface signature

A method group corresponds to a physical bus. Only one method in a method group can be invoked at once, such as the read or write methods of on one port of a RAM.

TODO - in this release the general means of describing which methods are in which mgr is not fully implemented. It works as follows: Canned IP blocks, such as the SSRAMs, can have multiple methods per method group. These have been hand-crafted. Imported and exported IP blocks have exactly one method per methodgroup.

A component is mirrorable if all of its invoked methods have the mirrorable attribute.

There may be re-entry restrictions on which methods can be concurrently invoked, but low-level handshake can control these.

13.2.2 Required MetaInfo

A pre-compiled IP block is described with two (or more) metainfo files in extended IP-XACT format. KiwiC will generate such files for each block compiled but they may also be provided from other sources, such as IP libraries or hand-crafted.

The IP-XACT standard schema provides all of the information needed for net-level structural IP block interconnection.

bool	Referentially Transparent	Always same result for same arguments (statless/mirrorable).
bool	EIS (An end in itself)	Has unseen side effects such as turning on an LED.
bool	FL or VL	Fixed or Variable latency.
bool	External	Whether to instantiate outside the current module.
int	Block latency	Cycles to wait from arguments in to result out (or average if VL).
int	Initiation Interval	minimum number of cycles between starts (arguments in time) (or average if VL).
real	Energy	Joules per operation (for power modelling via SystemC virtual platform output).
real	Gate count or area	Area is typically given in square microns or, for FPGA, number of LUTs.

Table 3: Kiwi Extensions to IP-XACT for HLS

Beyond providing the block name and version number, it gives a full description of the net-level interface and any TLM interfaces in higher-level models. The precision of the implemented function is manifested by the bit-widths of the busses.

Hence the HPR System Integrator mode of compilation, illustrated below for the peered instances, is readily supported without extensions. Afterall, this is the primary use today for IP-XACT.

We currently do not support automatic selection of sub-assemblies based on non-functional parameters, such as area and energy, but method overloading within the API of a given block works. Also, we do not automatically partition a design for incremental compilation according to the scale of the blocks or other heuristics: instead `[Kiwi.Remote()]` attributes must be manually added.

Where a custom block is separately compiled for use in an incremental compilation project, it, generally, has a custom interface. Hence there are two IP-XACT documents associated with an incremental compilation step: a so-called 'spirit:abstractionDefintion' that defines the interface and the 'spirit:component' that defines the child component, making reference to the interface document and also other interfaces, such as management and services ports, also sported by the child.

The parent compilation will read in these documents. And further IP-XACT documents will be written to describe the parent block by the parent compilation.

A final document may ultimately be written by HPR System Integrator that is a 'spirit:design' for the whole structure.

We use a squirrelling function, akin to the one used for C++ link editing, to generate an almost-human-readable kind name for the the interface. Alternatively, a kind name can be manually specified in the C# `[Kiwi.Remote()]` attribute.

The abstraction definition describes the transactional method names associated with the net-level ports. For instance, a child component might have three methods, such as `read(a)`, `write(a, d)` and `flush()`.

The default approach is that each method has dedicated handshake, argument and result nets (as in Bluespec). The default approach is not always suitable, especially for pre-existing IP blocks. For example, on a single-ported RAM the address bus will be shared between the `read(a)` and `write(a, d)` methods. A second example is a general trig block ALU that implements ten different trig functions (sin, cos, tanh, ...): the argument and result busses will be shared over each invokable opertion.

One way to achieve sharing of argument and result busses, while retaining the default approach where each function has dedicated nets, is to write in C# a shim with a single callable method around the bock's natural API and direct operations to this target. This simply requires adding one further, public, method to the component's C# class definition and making sure that all required calls

```
class Server1
{
  void flush()              { ... }
  int read(int a)           { ...;  return foo; }
  void write(int a, int d)  { ...  }


  [Kiwi.Remote("HFAST")]
  public int transact(enum cmd, int a, int d)
  {
     switch (cmd)
     {
       case server1.cmd_t.flush:   flush();
       case server1.cmd_t.read:    return read(a);
       case server1.cmd_t.write:   write(a, d);

     }
    return 0;
  }
}
```

Figure 5: Monomethod API example. Several methods in a component are made accessible via a single shim method. This will reduce wiring between separately-compiled coponents, which may or may not be helpful (e.g. helpful when interconnected between FPGAs), but is also a good way to connect to existing IP-blocks that were defined to share the same net-level pins over various transactions.

pass through that method. An example is in Figure 5.

To exploit an existing component as a black box, the RTL result of synthesising the child component is not needed. The IP-XACT defining the child should be manually edited in the place where it refers to the RTL filename to instead refer to a manual implementation that uses the third-party component, such as the CAM on the NetFPGA board (see ... to be added).

Alternatively, going beyond the default method, so-called 'meld' code can be provided that defines the transactional protocol at the net level.

TODO: define re-entrant synchronisation aspects and sharing of resources over entry points...

IP-XACT only provides about half of the information needed to import a hardware IP block for HLS so we use extensions for this purpose. Additional information is needed for replication and schedulling of such blocks in an HLS flow. A summary of the additional information needed is in Table 3. We use the `<spirit:VendorExtensions><hprls:...>` namespace for our extensions. The schema is here: LINK MISSING.

13.2.3 Instantiation Styles

There are two main module instantiation styles: IP blocks can be instantiated as peers or with hierarchy.

Each instanced block needs to have both a C# implementation and an RTL implementation packaged with an IP-XACT wrapper. The RTL and IP-XACT may have been generated by earlier runs of

KiwiC or else may have been created by hand or have come from a third party. The C# version is required for two reasons: 1. so that the instantiating C# file will compile without a missing class error, and 2. so that the the system as a pure dotnet design in WD (workstation development) environment. Only a stub implementation (null method bodies) is needed for C# compilation to succeed. And for the dotnet run, only a high-level behavioural model is needed in the C# when the real implementation comes from elsewhere, such as when it is hardened IP like the NetFPGA CAM.

Peer interfacing requires both sides to import a shared interface declaration so they may be compiled separately at the C# stage, yet still communicate afterwards. This could be a TLM abstraction of a standard interface, such as an AXI variant, or it could be a custom application-specific interface. And a TLM2-style socket set might be used to falicitate the binding.

Peer instancing skeleton example:

```
// See http://www.cl.cam.ac.uk/research/srg/han/ACS-P35/obj-2.1/zhp283300d5c.html
  RAM r = new RAM(...);     // Create peer instances
  CPU c = new CPU(...);     //
  IO  i = new IO(...);      //
  c.axi_m0.bind(r.axi_s0);  // Establish wiring between them.
  c.axi_m1.bind(i.axi_s0);  // bind is provided by SystemCsharp TLM.
```

Hierarchic instancing is where one C# file is compiled first and a second has an instance of it available during its own compilation.

Hierarchic instancing skeleton example:

```
  [Kiwi.Remote(...)] ALU a = new ALU(...);

  int foo(int x, int y) = { return x * a.f1(y/121); }
```

KiwiC will be invoked several times in either of these coding styles and each run generate a set of output files. Each set consists typically of some RTL and/or SystemC files and an IP-XACT meta file describing the set.

In the peer instancing example, each of the three instantiated components is defined as a class that is itself marked up with the Kiwi.Remote() attribute. In the hierarchic example, the attribute is instead applied to the instance. Also, in the hierarchic example, the ALU instance may actually be placed outside the rendered containing RTL with additional top-level ports provided for wiring it up.

Note that the ALU in the hierarchic example might typically be stateless and hence replicatable. If so, its invokation will be completely on a par with the multiplier and divider instances also needed for method foo. The HLS binder will decide how many instances of it to make and the HLS scheduller will factor in the appropriate fixed pipelining delay or variable delay and handshake nets.

13.3 Subsystem Abend Syndrome Routing

Kiwi defines that if any subsystem stops with an abend syndrome code, this must be passed up through parent modules to the substrate wrapper. And all modules must halt at that instant so PC values can be collected.

An example of glue logic being inserted by HPR System Integrator is when it must collect these abend syndromes and PC values from each instantiated module and combine them into a larger abend code and to halt the composite when any component abends.

In the peer instancing example, the KiwiC front end will invoke the HPR System Integrator function (§39) of the HPR library that underlies Kiwi.

The HPR System Integrator compiler takes a set of HPR VMs and generates SP_RTL constructs to wire up their ports following the VM instantiation pattern or an input IP-XACT document. It will instantiate protocol adaptors and glue logic based on pre-defined rules.

Please see SoC render part of the manual: Section 39.

HPR System Integrator supports:

1. Creating inter-module wiring structures with tie-off of unused ports.
2. Working both at the TLM level and structural net list level.
3. Outputs are in Verilog, IP-XACT, SystemC TLM, SystemC behavioural and SystemC RTL-styles.
4. Glue logic insertion in the form of instantiated adapators from the library are readily inserted automatically using rules based on interface type differences.
5. Custom glue logic from the Greaves/Nam cross-product technique can also be rendered.

Another example, at the moment, is that KiwiC generates HFAST load/store ports but the Zynq platform requires these to be adapted to AXI. This can either be done automatically by HPR System Integrator or by using the IP Integrator GUI within Vivado.

14 Design Examples

There are some examples in the standard distribution, such as primes and cuckoo cache.

14.1 A get-started example: 32-bit counter.

Here's how to make a simple synchronous 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 IV

Expert and Hardware-level User Guide

15 Kiwi Hard-Realtime Pipelined Accelerators

Note: real-time Pipelined Accelerator mode is being implemented 3Q16.

Classical HLS generates a custom datapath and controlling sequencer for an application. The application may run once and exit or be organised as a server that goes busy when given new input data. KiwiC supported only, up until now, that classical way for each thread. We call this 'sequencer major HLS mode'.

In 'Pipelined Accelerator' major HLS mode, KiwiC will generate a fully-pipelined, fixed-latency stream processor that tends not to have a controlling sequencer, but which instead relies on predicated execution and a little backwards and forwards forwarding along its pipeline.

Like classical HLS mode, a compilation root is identified in the high-level source, but its manifestation in the hardware is different. The loop is implemented by the subtrate instead of the KiwiC-generated RTL. Hence a different subtrate is needed and different techniques are used to connect such components together by HPR System Integrator.

Rather than using a 'HardwareEntryPoint()' attribute, for accelerator mode we mark up a method with the following attribute. The method can be static or dynamic. It should make its primary I/O through its arguments and result, but may refer to free variables for parameterisations.

/beginverbatim [Kiwi.PipelinedAccelerator("BiQuadxKernel", "II=1:MaxLat=16")] /endverbatim

The root designation for a hardware accelerator is a C# static method with arguments and a return value. This is typically the loop body of a C# iteration where the loop construct itself is only used in C# form in WD (workstation development) execution (§3.2).

A pipelined accelerator mode with latency set to zero results in a purely combinational circuit in terms of input to output data path, but it may post writes to registers and RAMs that still need a clock.

The prior `Kiwi.Remote()` attribute, described in §7.1, enables a given method to be cut out for separate compilation. This was non-rentrant and does not enforce hard real time.

When generating a real-time accelerator, a C# function (method with arguments and return value) is designated by the user as the target root, either using a C# attribute or a command line flag to the KiwiC compiler. The user may also state the maximum processing latency. He will also typically state the reissue frequency, which could be once per clock cycle and whether stalls (flow control) is allowed.

```
[Kiwi.HardwareEntryPoint(Kiwi.PauseControl.pipelinedAccelerator)]
static int piCombDemo(int arga) // The synthesis target
{
  // Trival example: probably a combinational design infact.
  return arga+100;
}
```

For a real-time accelerator, multiple 'calls' to (or invokations of) the designated function are being evaluated concurrently in the generated hardware. Operations on mutable state, including static RAMs and DRAM are allowed, but care must be taken over the way multiple executions appear to be interleaved, just as care is needed with re-entrant, multithreaded software operating on shared variables. Local variables are private to each invokation.

Although we default to every concurrent run's behaviour being treated in isolation, we support two means for inter-run communication: we can address the arguments and intermediate state of neighbouring (in the time domain) runs and, as mentioned just above, we can read and write mutable state variables that are shared between runs.

Variable-latency leaf cells cannot be instantiated (currently) in accelerator mode where the latency varies by more than the reinitiation interval. Further details need defining, but, for now, we need to avoid off-chip DRAM and KiwiC will request fixed-latency integer dividers (latency equal to the bit width) instead of the more commonly instantiated variable-latency divider.

15.1 Pipelined Accelerator Example 1

A simple example is test54 in KiwiC regression suite. Alternative mark up illustrated ... final system under design.

```
static readonly uint[] htab4 = { 0x51f4a750, 0x7e416553, 0x1a17a4c3, 0x3a275e96,
                                  ... many more entries ...
                                };

// We require a reissue interval of 1 (fully pipelined)
// We want a maximum latency of 16.
[Kiwi.PipelinedAccelerator("accel1", "nostall", 1, "maxlat", 16)]
static uint Accel1(uint a0)
{
  uint r0 = a0;
  for (int p=0; p<3; p++) { r0 += htab4[(r0 >> 6) % htab4.Length]; }
  return r0;
}
```

We can specify the reissue interval via the C# attribute. In this example, a reissue interval of 1 is specified. This generates fully-pipelined hardware that can be supplied with fresh arguments every clock cycle.

We also specify the maximum result latency as 16. KiwiC will determine its own latency, up to this value, guided by the logic cost settings, and report it in the KiwiC.rpt output file.

The ROM, in the full source code of the example, has 256 entries, and so is implemented as a statically-initialised block RAM on most FPGAs. This has a synchronous access time of one clock cycle. For multiple, concurrent accesses, as required by the reissue interval of 1, the ROM must be mirrored. Owing to loop-carried ROM address dependencies, the minumum implementation latency, by inspection, is 5 cycles.

All loops offered in pipelined accelerator mode must be fully unwindable by KiwiC. This means they must have a hard and obvious upper iteration limit, but they may have data-dependent early exit.

Internally, in our first implementation, the bevelab recipe stage unwinds all loops. This gives a single superstate to the restructure recipe stage which operates in a mode where all holding registers and input operands are replicated as needed in pipeline form and where mirroring of structural resources, such as the ROM in the above example, is used to avoid structural hazards arising not only for multiple use by a single run, as normal, but over different stages in that run that are separated by more than the reissue interval.

16 Designing General/Reactive Hardware with Kiwi

Kiwi can be used in an RTL-like style for some applications. This is where the user takes more active control over clock cycle mapping than is required or desired by scientific users.

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 bit-bang coding to connecting to existing hardware interfaces like AXI, I2C and LocalLink. Ideally, protocols are supported natively by Kiwi and bit-banging can be avoided.

16.1 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.

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

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

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

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

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

16.2 Register Widths and Wrapping

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

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

When running the generated C# natively as a software program (as opposed to compiling to hardware), the width attribute is ignored and wrapping behaviour is governed by the underlying type, which in the example is a byte. We took this approach, rather than implementing a genuine implementation of specific-precision arithmetic by overloading every operator, as done in OSCI SystemC [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 assign wrap error:
(widthclocks_fivebits{storage=8 }+1)&mask(7..0):
assign wrap condition test rw=8, lw=5, sw=8
```

The following examples work

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

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

A short-cut form for declaring input and output ports

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

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

16.3 How to write state machines...

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

16.3.1 Moore Machines

First compare the Moore machines define by main_pre and main_post:

```
[Kiwi.Input()] int din;
[Kiwi.Output()] int q;

main_pre()
{
  q = 100;
  while (true) {  q -= din; Kiwi.Pause();  }
}

main_post()
{
  q = 100;
  while (true) {  Kiwi.Pause(); q -= din;   }
}
```

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

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

16.3.2 Mealy and combinational logic:

Coding Mealy-style logic and purely combinational sub-circuits is not currently supported (but will be via pipelined accelerator mode where latency is set to zero cycles). Purely combinational logic could possibly inferred from an unguarded infinite loop, such as main_comb

```
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.

```
[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?

16.4 State Machines

Explicit state machines can be coded fairly naturally:

```
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:

```
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.

16.5 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 [3] and the user's call to built-in functions such as '`Kiwi.Pause`'. By default, one clock domain is used and default net names `clock` and `reset` are automatically generated. To change the default names, or when more than one clock domain is used, the '`ClockDom`' attribute is used to mark up a method, giving the clock and reset nets to be used for activity generated by the process loop of that method.

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

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

17 SystemCSharp

SystemCSharp follows the design of SystemC using C# instead of C++. Currently there is a very initial version of it in existence. Please see the README.txt in its folder.

SystemCsharp is a library, written in C#, that provides RTL semantics for hardware modelling. In particular, it provides signals that support the evaluate/commit paradigm of synchronous digital logic, where all variables in a clock domain take on their new values, atomically, one the active edge of the relevant clock.

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 `-cgen-fn=filename` flag can be used to change the output filename.

Several of the C++ output flags affect the way that C# is generated but these may be decoupled in the future.

Note that emitting C# or C++ with the standard recipe writes these output files at the same point in the system flow as used for RTL output. Hence a large number of parallel, RTL-style assignments will be used. Using a shorter recipe or with some of the intermediate stages disabled, output closer to the input form can be rendered: for instance, with bevelab turned off assignments will be made in order using a thread instead of an HLS sequencer.

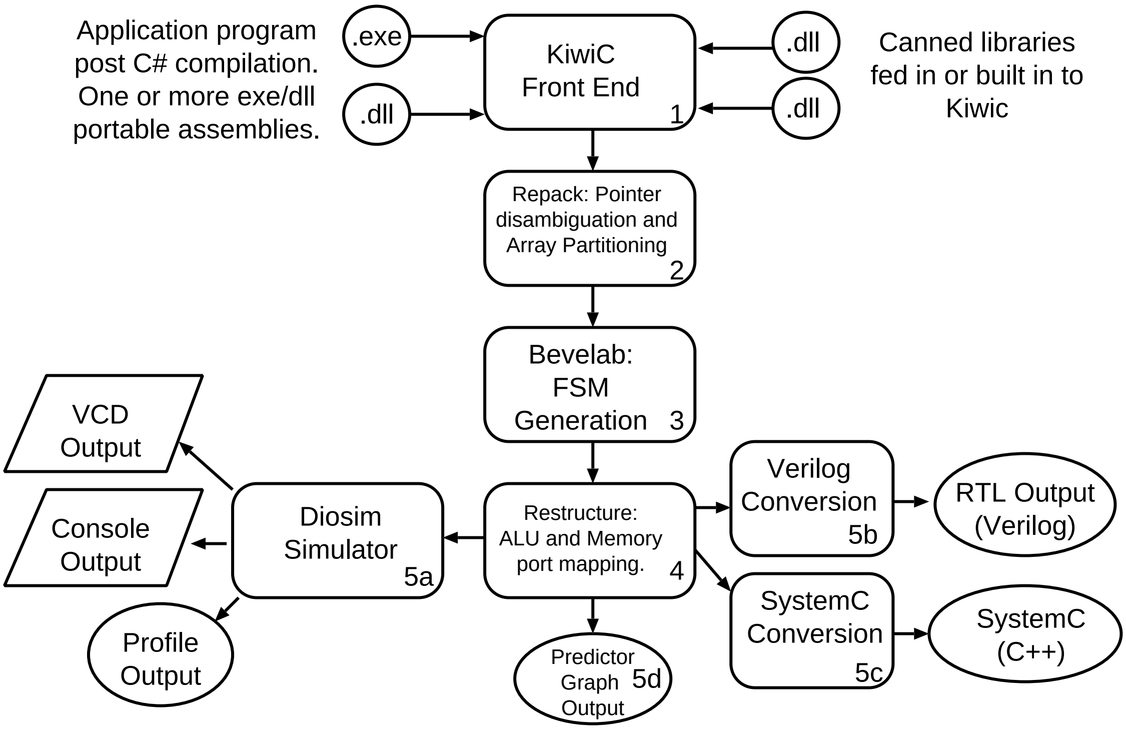


Figure 6: The main components of the default KiwiC flow using the default recipe (KiwiC00.rcp) in the KiwiC tool.

Part V

Kiwi Developers' Guide and Compiler Internal Operation

18 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 6 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

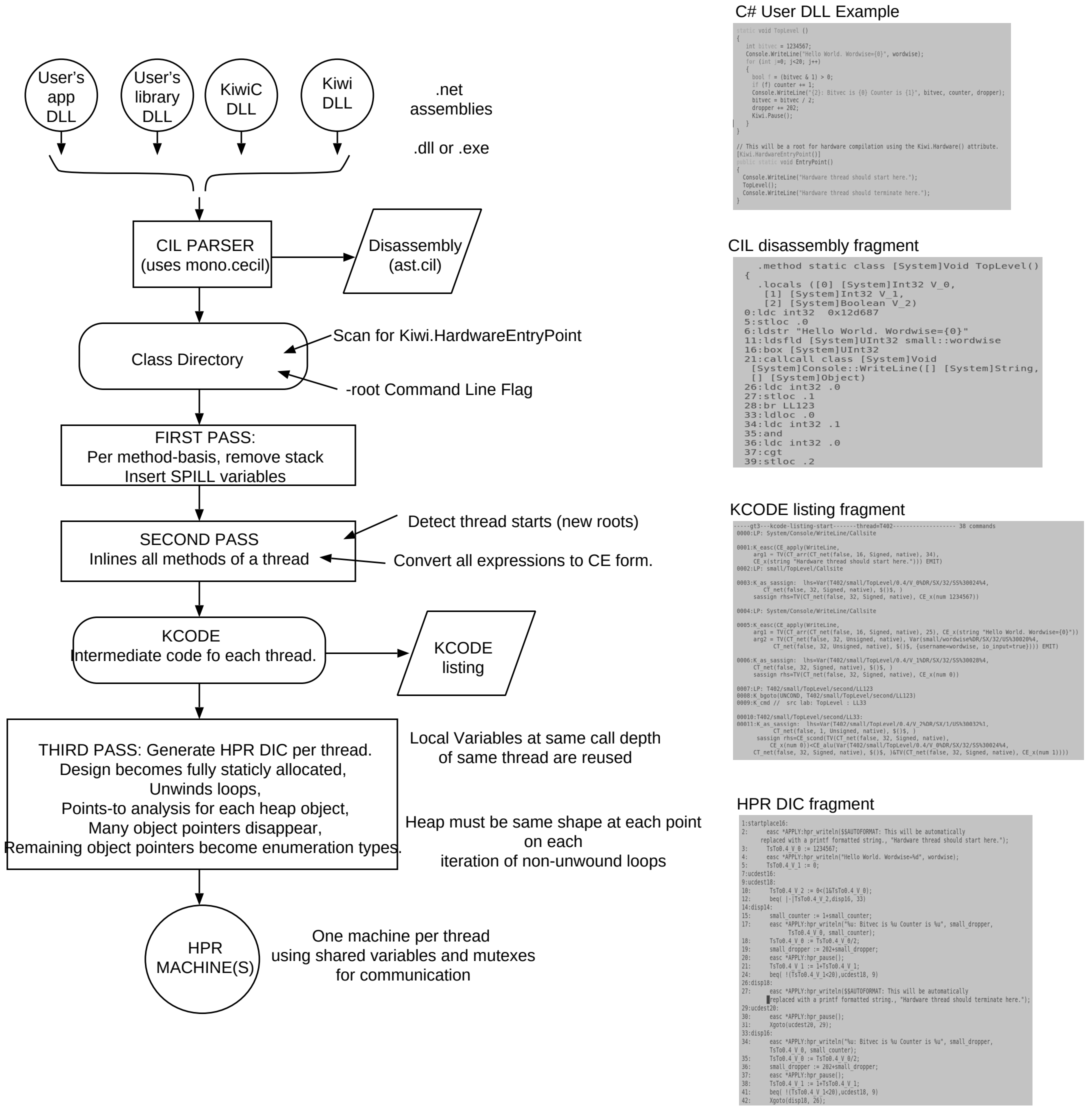


Figure 7: The internal flow of the KiwiC front-end recipe stage.

through Verilog. For debugging, Diosim can be applied to any HPR machine intermediate form, by varying the recipe. (There's also a shortcut '-conerefine=disable -repack=disable -verilogen=disable' that will cause diosim to run the original VM generated by the KiwiC front end without conversion to hardware). This is needed for the profile-directed feedback.

The .NETexecutable 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 (using the `kiwic-cil-dump` flag).

The KiwiC front end (IL Elaborate stage) converts the .net AST to the internal representation used by the core HPR/LS library. This is the HPR VM2 machine.

The VM code emitted by KiwiC front end is a set of parallel 'DIC' blocks. These are 'directly indexed code' arrays of imperative commands and there is one for each user thread. 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).

Memory disambiguation and partitioning into statically-sized memories and DRAM is done by the repack receipe stage (§29) . The KiwiC front end has labelled every storage operation with a storage class. Repack conglomorates classes that are assigned between and then uses arithmetic pointer analysis rules for alias analysis. Its input is an HPR VM where every variable and array location has a virtual address (hidx) in a so-called wondarray. A wondarray is allocated for every dotnet datatype (except structs). The wondarray contains $2^6 4$ words of that datatype but only the words on integer multiples of the datatype's size in bytes are used. The output from repack has had all of these mapped to scalars or to smaller 1-D arrays and each is branded with an identifier. Some input variables to repack have been allocated a reserved 'unadressable' hidx which means they are scalar and do not have their address taken. These go through repack without modification and appear as identical scalars in the repack output. In Kiwi use, these correspond to static variables.

The conerefine recipe stage 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. Object and array handles that are not manipulated actively by the program are removed.

The conversion from imperative code to FSM is performed, normally, by bevelab, described in §24. This allocates work to clock cycles based on the `Kiwi.Pause()` statements manually embedded by the designer or automatically inserted by the KiwiC front end. The bevelab output is an HPR machine where every statement from every thread nominally operates in parallel — i.e. pure RTL. However, some PC-like annotations are retained for easily projection (and re-encoding) in FSM form. FSM re-encoding for thread's controller will later typically be done by the FPGA tools to simplify the controller output decode function.

The restructure recipe stage (§30) binds and schedules operations and storage to physical resources. Storage 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. ALU's and other primitives are also instantiated and bindings of program operations are made. Owing to the FSM annotations preserved by bevelab, the binder can easily determine which RTL statements are disjoint. Each state in the input FSM potentially becomes multiple, so-called, microstates in the output as structural hazards on memory ports are avoided and pipelined ALU operations are composed. Allocation decisions are based on heuristic rules parametrised by command-line flags and recipe file values, such as the number of floating-point multipliers per thread.

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 non-blocking assignments.

18.1 Background: HPR/LS Library (aka Orangepath)

HPR L/S (aka Orangepath) is a library and framework designed for synthesis and simulation of a broad class of computer systems, protocols and interfaces in hardware and software forms. The Orangepath library provides facilities for a number of experimental compilers.

The primary internal representation (IR) is a so-called HPR VM2 virtual machine. The framework consists of a number of plugins that operate on this IR. Hence, in type terms at least, all operations are 'src-to-src'. But in practice, certain forms cannot be used in certain places: for instance a VM2 containing RTL code cannot be rendered directly as C++ (it would have to be passed through the bevelab plugin first).

HPR virtual machines and the operations to be applied to them are stored in a standard opath command format to be executed by an Orangepath recipe (program of commands).

A characteristic feature of Orangepath is that plugins can potentially, always be applied in any order and often have inverses. For instance a plugin that outputs RTL is reveresed by a plugin that reads in RTL. A plugin that performs HLS from behavioural code to RTL would be reversed that by a plugin that gives a single-threaded imperative program from a large body of parallel RTL code.

A simulator plugin, called diosim, is able to simulate the IR in any form and, in particular, is able to simulate interactions between parts of the system defined in different styles. For instance it can simulate a pair of CPU cores communicating with each other where one is modelled in RTL and the other as a cycle-callable ISS. Asynchronous I/O and network hardware is also modellable with these primitives.

A so-called recipe, which is an XML file, invokes the plugins in a particular order, supplying parameters to them. The input and output of each recipe stage is a so-called HPR VM2 machine. 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 recipe and the command line are combined and fed to the plugin components as they are invoked. 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 plugins that support a variety of external input and output formats.

An HPR VM2 machine contains scalar and 1-D array declarations, imperative code sections and assertions.

Values are signed and unsigned integers of any width and floating point of any width is also supported in the framework but library components currently only work for IEEE 32 and 64 bit formats.

Enumeration types are also supported, the most important being the boolean type. For all enumerations, an exclusion principle is applied, in that if an expression is known not to be any but one of the values of enumeration, then it must be that one value. Booleans are held differently from other enumerations internally but all expressions on enumerations are only stored in minimised form (using Espresso or otherwise). The library supports a great deal of constant folding and identity operation elimination (such as multiplying by zero or one). It has limited handling for strings and string constants, which are either treated in the same way that they are handled in Verilog, which is as an expression or register of width 8 times the string length in characters, or as a special string handle type (where widtho=-1). But the Kiwi front-end and the repack stage can map a fixed set of strings to an enumeration type of a suitable width with the strings stored only once and indexed by the enumeration.

Expressions are held memoised, and in a normal form, as far as possible, that makes identity checking and common sub-expression reuse easier. This is especially useful to be able to rapdily confirm, as often as possible, index expression equality or inequality, to avoid name alias RaW/WaW dependencies on arrays and loop value forwarding for sequential access patterns.

The imperative code is in any mix of RTL and DIC forms. RTL contains register transfer assignments, partitioned into clock domains, where all assignments in a clock domain run in parallel on the active edge of the clock. There is also a combinational domain that has no clock. The DIC imperative form (directly indexed code) is an array of statements indexed by a program counter, where the main statements are: scalar assignment, 1-D array assignment, library call and conditional branch within the array. Code sections can be in series or parallel with each other, using CSP/Occam-like SER and PAR blocks. Assertions are coded in temporal logic and associated with a clock domain, just like PSL (property specification language). And a dataflow/transport-triggered IR form is being implemented at the moment.

Dynamic storage allocation is also being added.

HPR L/S (aka Orangepath) represents a system as an hierarchy of abstract machines in a tree structure. Its aim is to 'seamlessly' model both hardware and software in a common intermediate form that suits easy co-synthesis and co-simulation.

Each machine is a collection of declarations, executable code and assertions/goals. But typically, an individual machine only uses on form of representation.

Plugins convert the machines from one form to another.

Other plugins generate machines, read them in from files or other front-end languages, or write them out.

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.

It also includes some temporal logic for assertions. Software can exist as both machine code/assembler and a high-level, block-structured, AST form.

A VM contains variable declarations, executable code, temporal logic assertions and child machines.

A system is a tree of VMs where each may be the root of a tree of VMs.

Variables are signed and unsiged integers of various precisions, single and double precision floating point and 1-D arrays of such variables. A small amount of string handling is also provided. All variable are static (no dynamic storage) and must be unique in a single namespace that spans the system. The variables are declared inside a given VM and may be global or local. Global variables may be accessed by code and assertions in any VM and local ones should (not enforced) only be accessed in locally (or in son machines?).

Expressions commonly use the hexp_t form and commands use the hbev_t form. Single-bit variables have hbexp_t form. A library of 'ix_xxx' primitives can be called as functions or procedures from hexp_t, hbexp_t and hbev_t respectively. Expressions are all stored in a memoising heap using weak pointers.

The executable code of a VM has several basic forms (dic, asm, rtl, cmd, fsm). All code and assertions access the variables for read and write (but assertions don't tend to write!) regardless of form.

18.2 DIC

DIC - Directly-indexed array: Imperative program (assign/conditional branch/builtin call) stored in an array indexed by a PC.

18.3 ASM

ASM - Assembler for a local family of microprocessors

18.4 RTL and FSM

RTL - Register transfer-level code - a set of parallel assignments to be executed on an event.

18.5 CMD

Abstract syntax tree of a block-structured imperative program (for/while/break/continue/assign/if etc) or single assigment statement.

18.6 Finite-State Machines

FSM - Finite-state machine form - like RTL but the assignments are collated into disjoint sets that are separated by a current-state variable or program counter. Any RTL can be factored out in this way (a Shannon Decomposition) using any set of bits as the program counter. The decomposition is a reversible transform.

18.7 CSP/Occam

Message-passing, CSP-like channels are another thing that should perhaps be added as a primitive form in future. They make perfect sense in the overall framework. CSP communication primitives should really be added ... to add channels and complete the picture.

The executable code may be clocked or nonclocked. Fragments may be put in serial or parallel using the SP_par and SP_seq combinators. There are two variants of SP_par, for lockstep and asynchronous composition.

Further executable forms, just being added are executable dataflow graphs:

VSDG - a dataflow graph for a single basic block with additional state edges representing memory order constraints. VSFG - an executable form of the VSDG where back edges in the control flow graph are represented using nested graphs.

The library is structured as a number of components that operate on a VM to return another VM. The opath (orangepath) 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.

Automatic recipes: The overall systems is a pluggable library. Where certain components only accept certain input forms and such a component is specified to be used by a recipe, it is envisioned that automatic invokation of the other components to serve as input adaptors will be triggered. Otherwise it is necessary to manually instantiate additional recipe stages.

For Kiwi use, the opath default recipe file is `KiwiC00.rcp`.

In this manual, we concentrate almost entirely on the .NETCIL input format and the Verilog RTL output format.

18.8 Internal Working of the KiwiC front end recipe stage

The IL Elaborate stage is implemented by the the FSharp files `kiwipro/kiwic/src/*.fs`. It reads in CIL code and writes out HPR 'dic' form code. Internally it converts from CIL to, so-called, kcode, before generating HPR code. The kcode can be rendered to a file for debugging/inspection using the `kiwic-kcode-dump` flag. The dotnet VM is a stack machine and the dotnet code is stack code. The stack is removed during the conversion to kcode. Kcode is neither stack or register code: all data is instead stored in wondarrays or global static variables.

CIL code is the assembly language used by the mono and .NETprojects. Like other assembly languages, it has an assembler and disassembler for converting between binary and human-readable forms. KiwiC originally read the assembly using a bison parser but now reads the binary using the mono.cecil libraries.

Front end flow steps are:

1. Perform first pass of each invoked method body in isolation.
2. Perform a symbolic execution of each thread at the CIL basic block level and emit kcode for each block. CIL branch instructions and CIL label names that are branch destinations define the basic block boundaries. This inlines all dotnet method applications.
3. Optimise the kcode within each thread using constant folding.
4. Analyse kcode to find the end of static elaboration point in each thread's lasso structure.
5. Perform register allocation (colouring) for the run-time part of each lasso.
6. Prefix start-up code from static class and method constructors to the lasso stem of the main thread.

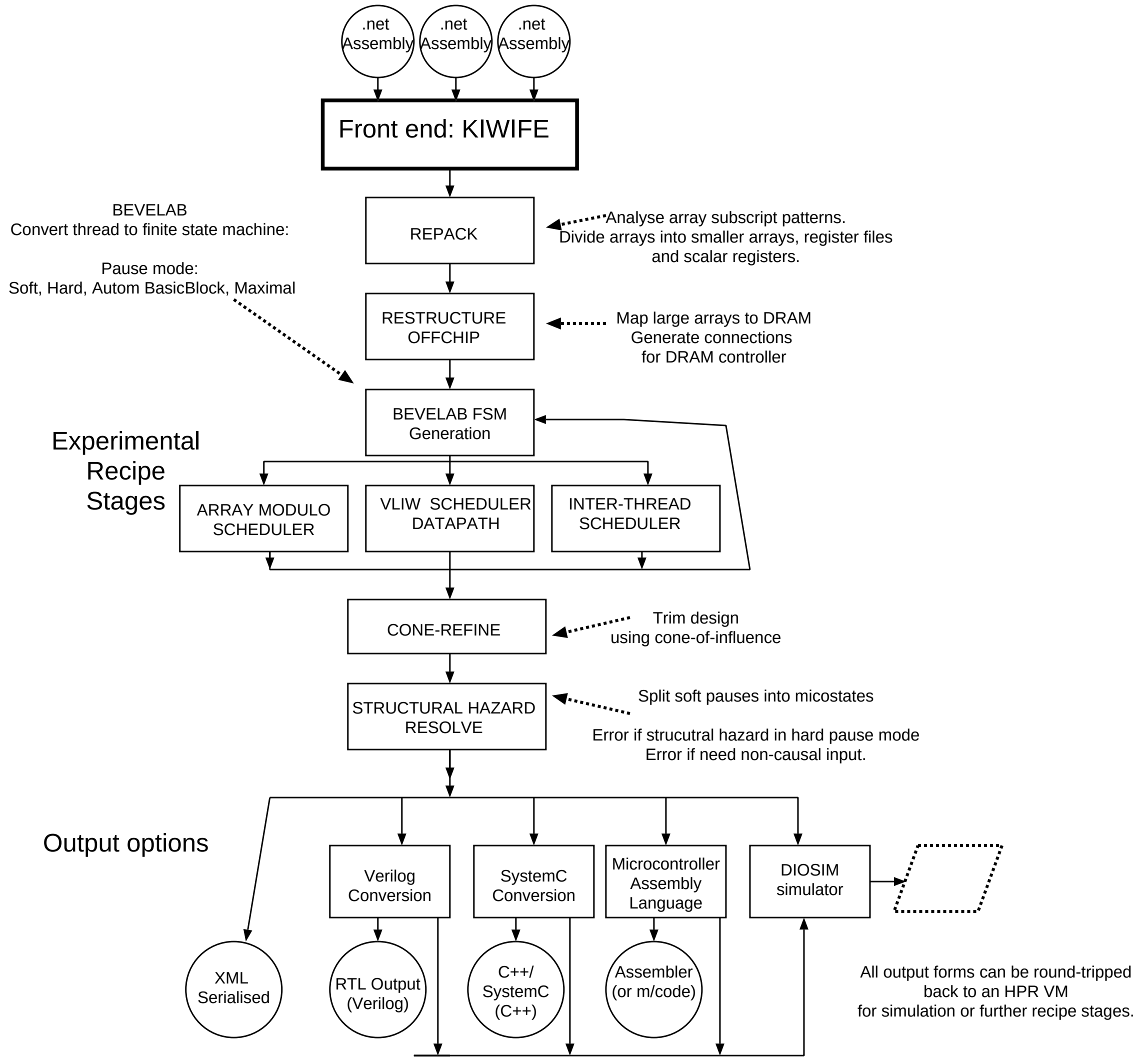


Figure 8: General flow implemented in an early version of the KiwiC tool (same as figure **??**). This diagram does not clearly show the recipe stages now used. The DRAM restructure is now part of the general binding done by the later restructure phase.

7. Perform symbolic evaluation of the kcode and emit HPR code. Further thread starts may be detected, which causes recursive activation of most of the steps above. Each thread becomes a separate HPR dic.

8. Perform dataflow analysis of the kcode to establish and conglomorate label region names (storeclasses) and points-at relationships.

The front end peforms a first pass of every method body that will be needed. This finds the basic block boundaries and the dotnet stack depth at every branch or jump. It gives a symbolic name to every code site where a type is needed. It symbolically executes the code using types without data and ignoring the control flow. Basic blocks that commence or resume with values on the dotnet stack are modified to avoid this situation by defining additional local variables, known as spills, and byprefixing with loads and postfixing with stores. These spill variables are frequently optimised away within the front end, but if they hold data over a `Kiwi.Pause()` they may appear in the output RTL. All return statements within a method are replaced with a branch to the end of the method. This sets up all the ground work for removing the dot net stack, on the fly, each time the method is called.

A `-root` command line flag or `HardwareEntryPoint` attribute enables the user to select a number of methods or classes for compilation. The argument is a list of heriarchic 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.)

Part VI

Miscellaneous

19 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. My design takes forever to compile but seems to make more progress with `-repack=disable`.

A. `-repack=disable` will cause all arrays to be of size 2**64 words. The only thing you can usefully do with repack disabled is run the internal simulator, Diosim. Diosim models enormous arrays as dictionary-based sparse structures. It is nice to see the Diosim output, but the resulting RTL will break most back-end simulation or synth flows (unless they too are able to handle arrays like that).

Q. Can I use Kiwi for Visual Basic?

A. Kiwi has not been directed to address Visual Basic but there is a little trial/demo on the following link:

`http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/kiwic-demos/kiwi-visual-b`

Q. If I multiply by a constant, floating-point number, will specialist FP ALUs be made or will KiwiC use a standard FP adder with a tied-off argument?

A. Currently it is the latter, although the argument may not be tied off in all cases: generally the multiplier will be being used for various operations with multiplexing of provided arguments. Also, where it is tied off, the FPGA tools will typically perform some (considerable?) constant folding.

Q. I am converting from C code that contains legacy unions ...

A. KiwiC is not set up to handle unsafe unions at all. It mostly works on the basis that the input code is strongly typed, but there is a little backdoor (called FastBitConvert) somewhere for floating point operations. The standard GetBytes forms in BitConverter should also work, but they produce a lot of intermediate code that goes all down the KiwiC recipe until, hopefully, allmost totally disappearing in load/store elides in the final output.

From test56 - Adding the FastBitConvert attribute makes KiwiC ignore the bodies of functions such as these and replaces the body with its own fast-path identity code based only on the signatures of the functions.

```
[Kiwi.FastBitConvert()]
static ulong fast_from_double(double darg)
{
  byte [] asbytes = BitConverter.GetBytes(darg);
  return BitConverter.ToUInt64(asbytes, 0);
}

[Kiwi.FastBitConvert()]
static double fast_to_double(ulong farg)
```

```
{
  byte [] asbytes = BitConverter.GetBytes(farg);
  double rr = BitConverter.ToDouble(asbytes, 0);
  return rr;
}

[Kiwi.FastBitConvert()]
static uint fast_from_float(float darg)
{
  byte [] asbytes = BitConverter.GetBytes(darg);
  return BitConverter.ToUInt32(asbytes, 0);
}

[Kiwi.FastBitConvert()]
static float fast_to_float(uint farg)
{
  byte [] asbytes = BitConverter.GetBytes(farg);
  float rr = BitConverter.ToSingle(asbytes, 0);
  return rr;
}
```

Q. KiwiC stops with an incomprehensible error. How can I tell how far KiwiC is getting through my compilation?

A. The most simple approach, with a fragile tool, is to build up your application slowly and check whether KiwiC keeps compiling it successfully as you go. Visibility can be gained by adding command line flags to write out the disassembled PE file and intermediate kcode. The PE file can be found in `obj/ast.cil` if you add flag +-kiwic-kcode-dump=enable+. You should get one kcode listing file for each thread of your design. These can be found in files such as `obj/kcode.T403.gt4.txt`. These contain low-level imperative code generated from the C# method bodies. If the full Kiwife recipe stage runs successfully, you should see a file called `obj/h02_kiwife/report-full` which is the input to the HLS toolchain implemented by HPR in its subsequent receipe stages. You may need to add `-report-each-step` to get each report file added. Also, there are serveral verbose logging modes that can be enabled from the command line with flags called loglevel which should be set to zero for maximum output.

Q. Can we have 2 [Kiwi.HardwareEntryPoint()] in the same class? Are the threads being translated as different always blocks to Verilog?

A. There are three ways to make new threads.

1. I normally create a second thread from the first using the C# standard approach that you show and as used in some of the tests like test44.cs

```
Thread threadx = new Thread(newThreadStart(reader.ReceiveProcess));
```

but 2. having more than one hardware entry point attribute or 3. more than one entry on -root cmd line flag should all also work fine. The threads do not have to be in different classes but techniques 2 and 3 can only be added to a static method.

Note: Join is not supported at the moment.

Regarding the number of always blocks resulting, I am not too sure off hand. The compose recipe stage combines updates from different VM2s and this should perhaps ensure there is only one. But most designs, I think, run the same or and/or compile faster with -compose=disable. So the verilog_gen stage is also doing the same trick I think. Certainly a shared variable needs to be only written by one always block in the standard synthesisable Verilog subset. Or if it is an on-chip RAM then two threads maximum owing to dual-port RAM available in FPGA.

Q. ... but the compiler exhausts all of the memory and the machine crashes ...

A. Which stage is taking all the time ? Can you see the relative timestamps of the create time of the various folders in the `obj` folder?

Are you in hard pause mode and is all the time time being taken in the kiwife or bevelab? If so, make sure that every control flow path in your non-unwound loops contains a `Kiwi.Pause()`. You should be able to set the unwind budgets to smaller values to make the compiler stop attempting earlier. Defaults are large:

```
-cil-uwind-budget=10000
-bevelab-ubudget=10000
```

Q. I got another 2 warnings:

```
+++ precision failure? ::: diadic_promote_and_resolve did not know
 what to do with CT_cr(Emu/debug_operands, <<NONE>>) V_minus
 CTL_net(false, 32, Signed,[native])

+++ precision failure? ::: diadic_promote_and_resolve did not know
 what to do with &(CT_arr(CTL_net(false, 64, Unsigned,[native]),
 <unspec>)) V_bitor CTL_net(false, 64, Signed,[native])
```

A. This first one is a subtract of a 32 bit integer from a class reference (object pointer). The second one looks like you are doing bitwise or of a 64-bit value with with the address of an array.

Neither of these is allowed in safe C# although you can do what you want in unsafe C#. These operations are not supported. Kiwi only supports comparisons, multiplexing and assignment of array bases.

Q. If I want to multiply a pair of 32-bit numbers to get a 64-bit result I would typically use something like

```
int a, b;
long p = ((long)a) * b;
```

but won't this instantiate a 64-bit multiplier component?

A. The multipliers that KiwiC (restructure2) instantiates from cvgates.v, such as `CV_INT_FL3_MULTIPLIER_S`, are just soft macros that the FPGA tools will flatten and optimise on a use-case basis. If that multiplier is used just for the one multipication, the FPGA tools will trim the internal logic of the multiplier to handle only 32-bit inputs, using fewer DSP splices. If the instantiated multiplier has been schedulled for use at other use sites that use higher-order input or output bits, the multiplier will be trimmed less. But, the latency allocated to the 64-bit multiplier will be a couple of cycles more than the smaller one and the FPGA tools do not, of course, retime the design such that this can be reclaimed.

Q. I get a postscript file called 'nolayout.eps' what is this?

A. The HPR library contains a constructive placer that writes a graphical floor plan to an eps PostScript file. This is used for net-length power analysis on output RTL. It is also being used in the constructive placer to decide how best to colour registers and bind functional units such as ALUs.

Q. Do you have any Xillybus or JetStream (Manchester) demos?

A. No, but we expect these to be contributed soon ... Perhaps start with the the Zynq director substrate.

Q. KiwiC is generating a circuit with too many output terminals to fit in my FPGA. Why is this?

A. You may be directly instantiating the Kiwi-generated RTL as the top-level of your FPGA. This is not a normal design route: you should most likely be using a standard Kiwi substrate for your FPGA and it is the substrate that instantiates the Kiwi code. The problem most likely arises from the Waypoint outputs. These are only for simulation purposes and they can be safely ignored. If they are left disconnected in the component that instantiates the Kiwi-generated RTL the FPGA tools will delete the logic that drives them instead of attempting to route them to a lot of output pads (IO BLOCKS).

```
output reg [639:0] KppWaypoint0,
output [639:0] KppWaypoint1,
```

You can also use command line flag `-vnl-keep-waypoints=disable` to turn off their rendering.

Q. What IP-XACT support does Kiwi have?

A. There is a new feature (1Q17) to report each component synthesised using IP-XACT. The IP-XACT output should be the same for RTL and RTL-style SystemC outputs, but will be different for TLM style SystemC output owing to method calls being used instead of nets. The substrate access port for debug and directing also appears in the reported in IP-XACT (§10.4).

The cell libary of RAMs and other components that KiwiC instantiates is currently hard-coded in KiwiC, but as part of the increased support for incremental compilation and black boxes we will soon allow Kiwi to instantiate components described with IP-XACT.

The HPR L/S HPR System Integrator is a simple IP-XACT-driven wiring generator. This can be accessed via Kiwi's new HPR System Integrator facilities in early 2017.

Q. I tried more ideas for one-liners, such as:

```
exist = Array.IndexOf(LUT, tmp) > -1 ? true : false;
```

but it didn't work.

A. Since Kiwi imports very little of the standard C# libraries, the .Index method of the Array class is most likely missing. For 2-D and greater arrays, Kiwi uses an implementation in Kiwic.cs and it is easily possible to add the implementation of Index into those implementations in C# src code form and it should then work. For 1-D arrays, the bulk of the implementation is hardcoded inside

KiwiC, but there should be potential to extend the hardcoding with additional C# code and place that, ultimately, in Kiwic.cs as well. Its a matter of knowing what to put in there. In short you should easily be able to contribute your own implementation of such things.

Q. Why do I get KiwiC error: 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 was missing. Just copy your formal into a local var at the start of the function body for now. Fixed in version 2.16 onwards, August 2016.

```
void myfun (int fp)
{
  int copied_fp = fp;
  copied_fp += 1;   // Do not directly modify your call by value
                    // formals before Sept 2016.
                    //  (Pass by reference works fine).
}
```

Q. What does this mean: System.Exception: `CV_INT_FL2_MULTIPLIER_S` unrecognised gate for presim: arity=6

A. This is from the built-in simulator, diosim. The design has used a fixed-latency of 2 multiplier component (from cvgates.v or elsewhere) but the simulator does not know how to simulate it. Restructure2 should have included its own simulation model for each component it deploys, but one fix is to not apply diosim to this design (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. §10.2

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 builtin 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 in the generated RTL?

A. See §38. 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] FIFO_FIFO2_result; instead of reg FIFO_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/s00... 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. OLD: If the main entry point to the C# program allows its thread to exit then a finish will be put in the output code by default. This is indeed not synthesisable. Quite often one wants the program to exit when run native but not when synthesised. One 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/entry point for KiwiC and this will avoid a finish statement in the generated code.

NEW: We replace -kiwic-finish with -kiwife-directorate-endmode. OLD: Suppressing the default operation on main thread exit statement can be controlled with a command line flag `-kiwic-finish`.

```
-kiwic-finish= [ enable | disable ]
```

Another solution is to mark up the main body subroutine with the `Kiwi.Remote()` attribute. This places it in an infinite loop, where it will become ready to serve again once the body has finished, and adds handshaking wires to synchronise 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 error 'kiwife: ran out of lasso steps, please increase fe unwind budget' ?

A. If your program has no input, compiling it is the same thing as interpreting it. KiwiC is probably trying to run the whole program at compile time. To give it something to do at run time, a `Kiwi.Pause()` should be inserted before you enter the main outer loop of your application.

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. I get an error like [ERROR] FATAL UNHANDLED EXCEPTION: System.Exception: thread-start//T403/Main/t55_2: Creating class instance this/uid token=System/Action‘2/$star$1/@/16/SS/TX1/SINT/T Bad form heap pointer for obj_alloc of type System/Action‘2/$star$1/@/16/SS/TX1/SINT/TX0 post end of elaboration point (or have already allocated a runtime variable sized object ?). storemode=STOREMOD sbrk=/tend:nota_const constant_fold_meets entry_point=0

A. This is a Kiwi 1 restriction - most heap objects need to be allocated before the end of static elaboration. Consider moving the code that allocates the heap object to the class constructor or else to another method that you call earlier. (For allocate-once items, this code migration will become automatic soon.)

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 little support form them that is properly well tested. This is being fixed 4Q2016. 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. What string formatting is supported in Console.Write or WriteLine?

A. Up to three arguments are supported. String, integer decimal, integer hex and floating point should all work. String catenation is also supported provided it is done a KiwiC compile time.

Q. I get FPGA or RTL SIM error regarding `CV_SP_SSRAM_FL1` missing.

A. This is a single-ported synchronous static RAM with fixed latency of 1 read cycle. It will most likely be mapped to block RAM by FPGA tools. There are a number of such components that KiwiC instantiates. Please include a Kiwi technology library such as `distro/lib/cvgates.v` in your back end compile

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. What Console.Write formatting is supported?

A.

```
examples - all are standard dot net
  {0} - arg 0 in decimal or floating
  {1} - arg 1
  {2} - arg 2
  {1:x} - arg 1 in hex
  {1:X} - arg 1 in upper case hex
  {1:3} - field width of 3 decimal
  {1:03X} - field width of 3, hex with leading zeros
```

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 (but this is being added for ROM mirrors) 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-entrant 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.

Also, certain keeps can be marked up on the command line so that conerefine uses these as roots.

Another common cause of an empty or near-empty RTL file is that no compilation roots were specified. This can be spotted when the file `obj/h02_kiwife/report-full` contains no executable code. You then need to add something like `-root=MyApp.MyMain`. You also see in KiwiC.rpt that no root was processed, except for perhaps the odd class constructor.

Q. If I want a net-level I/O bus wider than 128 bits (the size of a ulong), what can I do?

A. There is some support for this that needs documenting, where an array is passed as I/O. The colourbars example illustrates this style, but it is not in the repo and has not been tested for a while. However, having a static C# struct (not a class) as an I/O ought to work. However, C# structs is not mature in KiwiC. We can easily fix a few basic cases now however. See test51.

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. Also, try setting the unwind attempt limits (cil-unwind-budget, bevelab-ubudget, etc.) to smaller values to discover the error earlier or to larger values if you think the effort is warranted.

Q. Icarus Verilog reports buffer overflow.

A. This results from too many commments in the RTL files. Add `-vnl-add-aux-reports=disable` to your command line.

Q. Icarus Verilog 10.0 gives fails on test0 and elsewhere.

A. That is a duff version, build Icarus 10.2 from source.

Q. KiwiC is trying to start wine and creating file paths with backslashes in them, even though I am running on Linux. It also reports it is running on NT 5.2 when there is no windows machine anywhere involved.

A. On recent linux systems, on encountering a .exe the shell will start wine and try to open windows and so on. The KiwiC shell scripts enable you to define MONO and you should set this in your environment to 'mono' or '/usr/bin/mono'. If this still does not fix the problem please set you shell env var MONO_OS_OVERRIDE to something begining with 'l' such as linux64 and KiwiC will override the installed path combiner and related options.

```
+++ 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;
```

Q. Hi, I was looking at the Kiwi project for compiling C# Programs into FPGA, what the tool does is convert the C# program to a logic circuit? is there is a way to visualize the logic circuit associated to program?

A. You can look at the circuit in the FPGA tools schematic viewer. But the generated circuit is typically very large indeed and you need to look at a block diagram of the datapath and a flowchart of the controller relating to each thread. The controller flowcharts are rendered in GraphViz dot but is often too large for that tool if it has 1000 or so codepoints. Graphical output for the datapath is being worked on at the moment as part of the new spatially-aware register colouring system that tries to minimise wiring and multiplexor complexity.

Q. Can I use Xilinx FIFOs? pg057-fifo-generator.pdf

A. The CAMs on the NetFPGA boards and the new Xilinx hardened FIFOs are typical third-party black-box componenets. These are accessible to Kiwi users by treating them as separately-compiled components to be invoked via `Kiwi.Remote()` See test72 under construction. Test72 shows both halves of the separate compilation needed to wrap up a third-party IP block for structural instantiation. But the wrapped up result for Virtex-like FIFOs is also going to be placed in the Kiwi distribution (folder name TBD) so that end-users need only do the easy half.

OLD ANSWER : To use them in Kiwi I would probably (currently) split the code for the source and sink units such that each can be separately compiled by Kiwi but so that the composite design can also be run as a mono program where the FIFO functionality is supplied by a fifo.dll generated from C#. For the FPGA implementation I would read the separate Verilog outputs from the two Kiwi compiles into the FPGA tools along with an implementation of the FIFO. My first implementation would be some simple hand-crafted RTL and then later I would replace this with the output of the Xilinx FIFO generator. The two stages are to retain ease of debugging and design portability, where an RTL simulation of the system without Xilinx IP remains possible.

Additional answer: please see test72 for a worked example that needs writing up proprely.

Q. The burning question for me is, what options are available for exploiting parallelism that are not explicitly referred to from the C code? Does your converter alleviate the Von Neumann ALU bottleneck from critical paths or is an imperative C description unsuitable for substantial acceleration opportunities?

A. With KiwiC, all the standard HLS limits on parallelism apply. This means a program that can be executed in one clock cycle will be executed in one clock cycle provided sufficient budgets on hardware resource use and and logic in a clock cycle are set.

The is no intrinsic parallelisation limit arising from a single-threaded, imperative description. But limits arise in practice from data and control dependencies/hazards.

Regarding data dependencies, where array subscript comparison is undecidable at compile-time (name aliases), the resulting h/w design from trying to go massively parallel is generally dominated by spurious multiplexing paths and not a good design. When making array subscript comparisons at compile time, KiwiC can spot common paradigms, such as identical expressions, constant expressions and mainfestly unequal expressions like x and $x+1$. Computing theory states that there will always be decidable equalities outside those KiwiC is programmed to decide.

Regarding control dependencies, the current KiwiC elaboration algorithms do not dynamically unwind outer loops when inner loops are still being unwound - this will be addressed in the VSFG replacement to bevelab. But a programming style where the loop exit predicate is determinable near the head of the loop body always helps in sequencer modes, as it does with Von Neumann computers, and the compilers always try to hoist it. There is no problem, of course, with data-independent loop control.

All object fields and static variables are currently strictly updated in program order. Additional annotation or policy control as ’non-architectural’ or ’relaxed’ for for fields or static variables may be supported in the near future. These will enable KiwiC to do more speculative execution but make debugging harder because program order will not be followed. To help this, architectural ’slave’ registers may be added for debug viewing that can simply be deleted by the FPGA toolchain if not being monitored in any way.

Q. What endianness is Kiwi - I need this for unsafe bit conversion routines ?

A. KiwiC supports only little-endian operations. There are various dot net API calls that you can make to interogate this at run time and Kiwi’s libraries provided this information. For your code to remain portable you should invoke this API and KiwiC will progagate the constants accordingly, discarding any code for big-endian support.

Q. From what I can tell the `kiwi.dll` is not being taken into account at all the kiwi specific stuff when creating the `.exe` ?

A. The Kiwi-specific ’stuff’ just adds a few attributes to the `.dll` — it will normally still run as a mono/dotnet program with those attributes in it. The KiwiC compiler invokes a multi-stage recipe with reports for each stage written to separate sub-folders in the ‘obj’ folder that it creates for itself. Their detail level can be increased with -report-each-step and various verbose and tracelevel settings. If the recipe gets as far as making something like ‘h10_verilog-gen’ in the ‘obj’ folder you should find the primary Verilog output file has been written to your filesystem in folder containing ‘obj’.

Q. Sorry to take your time again but I’m new to this and I wan’t to be sure of something, what is implemented on the FPGA is a processor that runs the program or is directly the representation of the program as a logic circuit?

A. Short answers: it is a circuit, not a processor plus firmware. There are various compilation styles. The fully-pipelined accelerator will run the whole program every clock tick, accepting new data every clock cycle, allbeit with some number of clock cycles latency between a particular input appearing at the output. Sequencer mode will generate a custom datapath made up of RAMs, ALUs and external DRAM connections and fold the program onto this structure using some small number of clock cycles for each iteration of the inner loops. Compilation directives alter the trade off between silicon used and the number of clock cycles needed. No standard processor is used. (High-level synthesis of this kind is used in your mobile phone and enables it to compress motion video from the camera without instantly flattening the battery.)

For larger programs, a good deal of the code tends to be start up and reporting code that is executed far less frequently than the main inner loops. This code can be placed on a standard processor and coupled to the HLS-generated hardware or else the datapath for the higher-performance parts can also be used as an unoptimised datapath for the less-commonly-executed code.

Q. Can continuous assignment be achieved between Kiwi net-level I/O descriptions.

A. There is no analogous behaviour for a C# program, but this facility might be useful in various debug lashups perhaps. See notes elsewhere on SystemCSharp. But a Kiwi.Hardware attribute given to an infinite loop in hard pause mode with no Pause statements inside it should probably generate combinational logic, or the pipelined accelerator mode with a re-initiation interval of zero should also serve. We need to check whether these currently work, but probably not at the moment (Jan 2018).

Q. KiwiC is taking a long time. Can I examine its progress?

A. Progress for each recipe state is logged in the obj/log folder. Also, if you add -progress-monitor as an early argument to KiwiC command line, a log file showing current progress is written every 10 seconds or so to the currently open log file or Console.

Q. Can I implement my own multi-port BRAM structure that is always ready or with handshake signals and will KiwiC then use disjoint ports for each thread?

A. Yes, you must first define a new RAM FU in IP-XACT and RTL. At a pinch you can replace the IP-XACT with attribute strings (which need only be defined once inside your C# and whose string handle is used at each instance). It can have then be instantiated by KiwiC. Or you can mark the RAM as 'offchip' and it will be accesed via bonout ports and the whole external wiring and instancing can (at some point soon) be automated with System Integrator. Demo example needed ...

The handshaking will sometimes stall the static schedule generated by the res2 recipe stage, but this should only affect performance marginally provided there is not excessive delay/contention inside your implementation. You can even implement an FU that embodies 'Efficient Multi-Ported Memories for FPGAs' by LaForest and Steffan, but this avoids all contention using BRAM replication proportional to the product of the number of read and write ports and it may be preferable to use handshaking instead.

Q. How do I spawn a thread in a different clock domain or II target using C# delegates?

A. missing ...

Q. Is the dotnet reflection API supported at all?

A. You can use Object.GetType and Object.ToString in certain places found so far to be useful. The results are not guaranteed to be the same as mono returns, but are nonetheless helpful.

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

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

Q. Do you support the BRAM enable power-saving feature of Vertex 7 and other FPGA families?

A. Block RAMs are always instantiated via the templates in `cvgates.v`. These templates have a '`ren`' input that is high on read cycles and low otherwise. You are free to rewrite the templates so that this read enable signal is used appropriately under RAM inference or direct instantiation. We have not checked whether this works for any FPGA families using the current coding style in that file.

Part VII

Orangepath Synthesis Engines

The HPR L/S (aka Orangepath) library supports various internal synthesis engines. These are plug-ins.

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

All plugins rewrite one HPR machine as another. But some that read in an external file, like the Kiwi front end or the deserialiser or the RTL front end simply ignore the input machine they are fed by the Orangepath recipe.

20 A* Live Path Interface Synthesiser

The H2 front end tool allows access to the live path interface synthesiser. The A* version is described on this web page. http://www.cl.cam.ac.uk/ djg11/wwwhpr/gpibpage.html

This plugin has not been tested recentl.

21 Transactor Synthesiser

The transactor synthesiser is described on this link

http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/transactors

This plugin has not been tested recently.

22 Asynchronous Logic Synthesiser

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

http://www.cl.cam.ac.uk/ djg11/wwwhpr/dsasynch.html

This plugin has not been tested recently.

23 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.

24 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 readibility, debugging and ease of determining disjoint structural operations in restructure (and perhaps to assist proof tools), and for the HPR Performance Predictor that needs to track the control flow graph through the complete toolchain.

(Alternatives to Bevelab are Systolic and VSFG. VSFG (§25) can achieve greater throughput with heavily-pipelined components in the presence of complex control flow. Systolic requires bounded loops and projects to a systolic array (not described in this manual).)

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. 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 reality they 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, re-pipelines 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:

Parameter	Style	Default Max		
Maximum number of name aliases array read		0		
Maximum number of multiplexors in logic path		10		
Maximum default number of iterations to unwind	loops	4		

Table 4: Bevelab Heuristic Table.

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-annotatated 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 'Pause()'. This is needed for net-level protocols, such as parallel to serial conversion in a UART, and for connecting to hard IP blocks that have synchronous interfaces. 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 4. It 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 capabilites of the FPGA tools to do their own minimisation.

The string `-vnl-resets=synchronous` should be passed in to add synchronous resets to the generated sequencer logic. This is the default.

The string `-vnl-resets=asynchronous` should be passed in to add assynchronous resets to the generated sequencer logic.

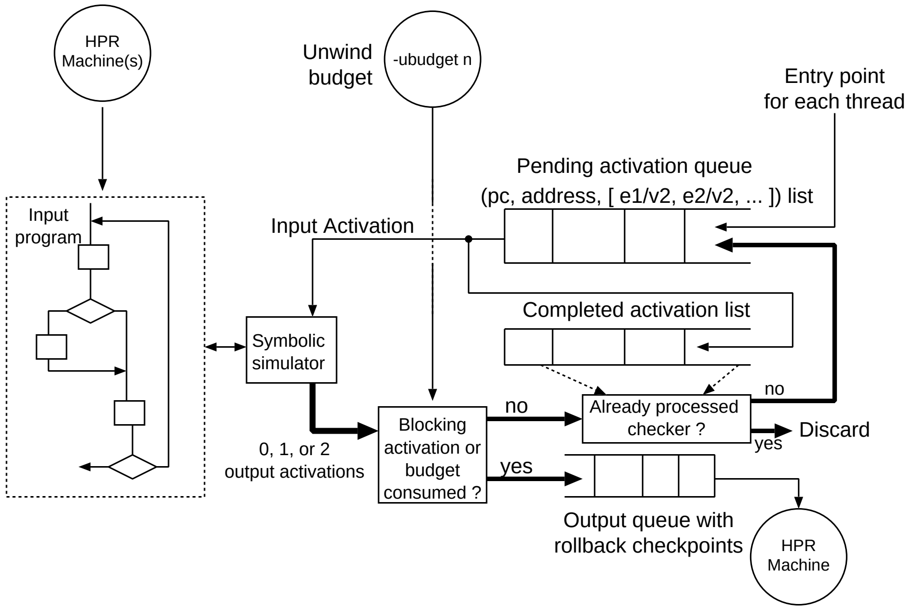


Figure 9: Details of the Hard Pause Mode algorithm, as provided by Bevelab plugin in the HPR L/S library.

The string `-vnl-resets=none` should be passed in to supress reset logic for FPGA targets. FPGA's tend to have built-in, dedicated reset wiring. See §38.

Bevelab has a number of scheduling algorithms (selectable from recipe or command line). Alternatively, Bevelab can be replaced with a different opath plugin, such as VSFG or Systolic.

24.1 Bevelab: Hard Pause Mode Internal Operation

This section describes only Hard Pause Mode. This is where the position of clock pulses is under explict programmer control via the insertion of `Pause()` calls.

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, fname, [args]) where the first form assigns e to v when g holds and the second calls function fname when g holds.

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

The pending activation list is initialised with the entry points for each thread. Operation removes one activation and symbolically steps it through a basic block of the program code, at which time zero, one or two activations are returned. These are either added to the output list or to the pending activation list. An exit statement terminates the activation and a basic block terminating in a conditional branch returns two activations. A basic block is terminated with a single activation at a blocking native call, such as hpr_pause. When returned from the symbolic simulator, the activation may be flagged as blocking, in which case it is fed to the output queue. Otherwise, if the unwind budget is not used up the resulting activations are added to the pending queue.

A third queue records successfully processed activations. Activations are discarded and not added to the pending queue if they have already been successfully processed. Checking this requires comparison of symbolic environments. These are kept in a "close to normal form" form so that syntactic equivalence can be used. This list is also subject to rollback.

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

24.2 Bevelab: Soft Pause Mode Internal Operation

Classical HLS operates by loop unwinding to expose parallelism. This is achieved by Bevelab when running in Soft Pause Mode. It reorganises the input program in terms of the number and size of basic blocks. Instead of expecting explict `Pause()` calls in the input language, as in Hard Pause Mode, basic blocks of appropriate size and with other properies are automatically generated from raw procedural programming. This is the approach needed for general acceleration of scientific (aka Big Data) programs.

The transforms available are :

1. Loop fusion: combining the operations of two successive loops with identical trip counts into one loop;
2. Loop unrolling: expanding the body of a loop by an unwind factor;
3. Predication: replacing control flow with conditional expressions;
4. De-predication: converting conditional expressions into control flow.

25 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' [5]. It can achieve greater throughput with heavily pipelined components in the presence of complex control flow compared with traditional loop unwinding and static schedulling.

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

26 PSL Synthesiser

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

Parameter	Style	Default Max		
Max no of integer adders and subtractors per thread	flash	unlimited		
Max no of integer multipliers per thread	one-cycle	5000 bit products		
Max no of integer dividers per thread	vari-latency	5		
Max no of F/P ALUs per thread	fixed latency of 5	5		
Max size register file (bits)		512		
Max size single-port block RAM per thread				
Max no of single-port block RAMs per thread		2		
Max no dual-port block RAMs shared over threads		2		
Max size dual-port block RAMs shared over threads		bits		
No of DRAM front-side cache ports		unlimited		
No of DRAM banks		platform-specific		

Table 5: An Example Structural Resource Guide Table.

27 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,

28 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.

29 Repack Recipe Stage

The repack function is essentially KiwiC-specific. It is therefore described in the KiwiC chapters of this manual (§4.8.1).

30 Restructure Recipe Stage

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 5 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 (§29) 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 scheduller 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 VIII

Output and Analysis Recipe Stages

The HPR library contains the Diosim simulator, output generators and other analysis tools. Each is a plugin invoked by an Orangepath recipe stage.

31 HPR Output Formats Supported

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 are created by selecting plugins:

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. Discontinued now.

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++ and CSharp Forms:** The HPR VM is output as C++ or C# code suitable for third-party compilers. RTL forms may also be output as synthesisable SystemC.

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

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

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

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

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

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

32 C++, SystemC and C# Output Generators

The **cpp-gen** recipe stage writes the current design as C++ or SystemC depending on options supplied to it. This can render any mixture of behavioural or structural code, depending on which processing steps come before it in the Orangepath recipe.

It also can generate C# code.

The `-cgen2=enable` flag causes the tool to generate SystemC output files.

The `-csharp-gen=enable` flag causes the tool to generate C# 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.

C# does not use header files as such, so files with suffix `'.cs'` are emitted. Classes may be spread over a number of files according to undocumented commandline options.

Note that emitting C# or C++ with the standard recipe writes these output files at the same point in the system flow as used for RTL output. Hence a large number of parallel, RTL-style assignments will be used. Using a shorter recipe or with some of the intermediate stages disabled, output closer to the input form can be rendered: for instance, with bevelab turned off assignments will be made in order using a thread instead of an HLS sequencer.

33 RTL Output Generator

The **verilog-gen** recipe stage writes the current design as Verilog RTL.

It is not a totally straightforward projection as RTL since sub-expressions of significant complexity that occur more than once are rendered only once and assigned to intermediate nets using continuous assigns under a greedy algorithm. This keeps the file size sensible with certain functions that would become exponential (e.g. a barrel shifter). The quality of the sharing is not optimisied owing to the assumption is that a subsequent logic synthesis tools will revisit these sharing decisions.

It also can convert the design to a netlist (i.e. do logic synthesis) and estimate the area of the result. This functionality should be split out into a sepearte recipe stage so, for instance, the net list could be rendered in SystemC instead.

It also contains a roundtrip function, such that the RTL it has generated is converted back into HPR internal form. It does this from the RTL AST so cannot serve for textual RTL input in its current form ... the RTL parser is in cv3cv3.zip and needs integrating ...

The RTL Generator can provide area and wiring length estimates and generate a graphical floorplan to help visualise the circuit structure and understand how much area is devoted to which resources.

Wiring length estimates based on the design hierarchy and Rent's Rule are fairly accurate and do not require an actual layout.

The flag `-vnl-layout-delay-estimate=enable` will create a layout.eps plot file.

34 IP-XACT Output Generator

The **ip-xact-gen** recipe stage writes the current design as an xml document following the IP-XACT 'design' schema.

It can also write out bus specs and individual components used in the current design as IP-XACT xml documents.

This plugin is/was formerly not freestanding and could only be invoked via the verilog-gen recipe stage.

34.1 Built-in report writers

The Orangepath framework has two built-in rendering tools that produce a textual listing file (called report or report-full) and Graphviz dot figures.

The `-report-each-step` flag causes textual report files for each recipe stage to be written into the obj folders. Alternatively, a pseudo plugin can be put in the recipe at a stage where such a report should be written.

The `-cfg-plot-each-step` flag causes the control flow for each recipe stage to be written into a report file in the obj directory. You will typical want to render the dot files with something like `dot -Tpng a.dot > a.png; eog a.png`.

The restructure stage accepts some older flags such as `-dotplot-plot=combined` but these may be discontinued.

35 Arithmetic and RAM Leaf Cells

The tool will expect the user to provide definitions of various leaf cells with the output from the tool at the input to the RTL synthesis step. A number of suitable definitions are included in `cvgates.v` and `cv_fpgates.v` and it may commonly be sufficient just to include these two files in the RTL compilation.

The leaf cell names follow a few conventions:

1. All have a clock and reset intput, even if not needed.
2. All have a fail output, even if they cannot fail or will report their error in-band using, for example, NaN.
3. The main outputs is listed before inputs, but associative instantiation is normally used anyway. For divide and mod the numerator is listed before the denominator. For subtractors the lhs is listed first.
4. The naming convention has the letters `VL` for a variable-latency component and this has handshake wires. Otherwise `FLn` denotes a fixed-latency of *n* clock cycles, fully-pipelined. The tool will schedule an average budget for variable lantency components.
5. Parameter overrides, listed in the order output, first intput, second input, set the precision of ALU connections and RAM dimensions.

For variable-latency leaf cells in the libray, the VLA protocol is used. The VLA handshake protocols is as follows:

- Handshake uses a `req` input and a `rdy` output.
- New input args are read in on a cycle where `req` is asserted, which will be just one cycle in response to a `req`.
- Results are ready in a cycle when `rdy` is asserted.
- New work may be presented with `req` during the same cycle that the output data becomes live (the `rdy` cycle).
- Asserting `req` before the last `rdy` has been delivered will be ignored.
- The output, once live, remains valid until another operation starts (i.e. until the cycle after `req` next holds).
- No combinational path between inputs and outputs, including `req` and `rdy`, is allowed inside the component.

Components following the AXI Streaming protocol are also supported. This is the same as the Xilinx LocalLink protocol in all important aspects. It has a pair of handshake nets (ready/valid) for both the input and the output and does not hold its data on completion. Compared with VLA, the AXI streaming component requires another holding register to be instantiated by the HLS tool when it knows it may need the data in more than one subsequent cycle in its schedule.

Note: The above is for on-chip devices instantiated directly by the tool. Off-chip RAM connections use a separate protocol (HSIMPLE, HFAST, AXI, BVCI).

35.1 Fixed-point ALUs

The RTL backend will use built-in RTL operators for adders and substractors. For multipliers and dividers and modulus with non-constant arguments it instantiates specific units, such as `CV_INT_VL_DIVIDER_US` Very small multipliers are rendered with the RTL asterisk infix operator and left to the FPGA tools as per the adders/substractors.

Kiwi generally calls out to variable latency dividers and fixed-latency multipliers. It uses an estimate for the variable latency computation time in its schedules. When using a fixed latency it increases the latency requested for larger parameter widths. Whether fixed or variable is indicated in the component kind name. Instantiated components cope with any argument width as specified by parameter overloads.

Kiwi does not currently generate the fixed-point ALU implementations and it may request one that is not in the provided `cvgates.v` baseline library, in which case the poor user must provide their own implementation. For example, an extreme design might call for a 512 by 1024 fixed latency multiplier with 5 clock cycle latency.

Recipe parameters alter the points at which the library enlarges the provisioned latency.

35.2 Floating-point ALUs

Floating-point ALUs follow the pattern of fixed-points ALUs, except that add and subtract are also always instantiated ALUs and the RTL compiler is not expected to handle them. A different set of recipe parameters control their structure (fixed/variable latency and expected/required latency).

Only 32 and 64 bit, IEEE standard floating point is currently used by default. A future extension will provide for custom width floating point, since this is a very powerful feature of HLS that can save a lot of energy and area. The extension will give the same behaviour on mono WD as on RTLSIM and FPGA.

A core set of floating point ALUs is provide in `cv_fpgates.v`. These are soft macros that the RTL tools are expected to map to whatever is available in the target FPGA or ASIC library. Specific shims and bindings to assist with Altera and Xilinx are likely to be added to the distro in the near future.

35.3 Floating-point Convertors

There is no budget limit on the number of convertors is currently imposed.

The convertors required normally are

```
CV_FP_CVT_FL2_F32_I32  // Integer 32 to float 32 with fixed latency of 2
CV_FP_CVT_FL2_F32_I64  // Integer 32 to float 32 with fixed latency of 2
CV_FP_CVT_FL2_F64_I32  // Integer 32 to float 32 with fixed latency of 2
CV_FP_CVT_FL2_F64_I64  // Integer 32 to float 32 with fixed latency of 2

CV_FP_CVT_FL2_I32_F32  // Integer 32 from float 32 with fixed latency of 2
CV_FP_CVT_FL2_I32_F64  // Integer 32 from float 32 with fixed latency of 2
CV_FP_CVT_FL2_I64_F32  // Integer 32 from float 32 with fixed latency of 2
CV_FP_CVT_FL2_I64_F64  // Integer 32 from float 32 with fixed latency of 2
```

```
CV_FP_CVT_FL0_F32_F64  // Float 32 from float 64 (FL=0 implies combinational)
CV_FP_CVT_FL0_F64_F32  // Float 32 from float 64 (FL=0 implies combinational)
```

35.4 RAM and ROM Leaf Cells

A set of standard static RAM cells is provided in `cvgates.v`. These are parameterisable in width, length and number of lanes by overrides. They are single and dual ported and of latencies 0, 1 and 2 clock cycles.

Kiwi and other tools based on the HPR library generate instances of these RAMs.

RTL tools are expected to map these to appropriate structures, such as LUT RAM and block RAM on FPGA.

RAM instances are also generated with no write ports and static initialisations using the Verilog `initial` statements. RTL tools will treat these as ROMs. Unlike RAMs, where the user is expected to manually couple a definition from `cvagtes.v` or elsewhere to their RTL synthesis step input, ROMs are are embedded in the main RTL output files from a run of the tool.

Part IX

HPR L/S (aka Orangepath) Facilities

HPR L/S (aka Orangepath) is a library and framework designed for synthesis and simulations of a broad class of computer systems, protocols and interfaces in hardware and software forms.

The HPR L/S library provides facilities for a number of experimental compilers. This part of the manual describes the core features, not all of which will be used in every flow.

36 FILES and DIRECTORIES

36.1 Recipe

Orangepath tools read a recipe file which instructs the sequence of steps used: compilation, analysis, optimisation, output generation and so on.

36.2 Output Log and Report Files

As well as specific output files generated by components of the recipe (such as RTL or Systemc), a log file is created for each recipe stage and an overall report file is made that any recipe stage can write to.

When an Orangepath tool is run, it creates a directory for temporary and report files. This is the obj directory. The name can be changed from the default ('obj' in the current directory) with the early arg `-log-dir-name`.

A similar early arg sets the name of the default directory for the specific outputs generated by a recipe stage.

The obj directory contains a sub-directory for each recipe stage. This is the stage report directory.

The output code from a recipe stage can be printed to a report file in its stage report directory if the `-report-each-step` flag is given.

the `-cfg-plot-each-step` flag

the `-bevelab-cfg-dotreport=enable` flag

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

36.3 Environment Variables and IncDir Search Paths

Tools must load various files from the filesystem and must know where to look.

Environment variables can provide places to look.

An HPR L/S tool itself will expect to have all of its dlls on the system search path or else in the folder accessed by `../lib` from where its binary file (such as `kiwic.exe`) is stored.

A user can specify additional folders to search for loadable files, such as previous outputs from incremental compilation steps and standard IP blocks. These are defined by the incdir path. The `HPRLS_IP_INCDIR` environment variable and the `-ip-incdir` command line or recipe flag can be set to a string that contains a colon-separated (semicolon on Windows) list of search folders. This is the incdir path. Most earlier outputs are described in IP-XACT and it is these metafiles that need to be found in this way, with the actual IP being held in a file named in the IP-XACT xml 'files' section. Where those filenames are non-absolute, they will be looked up in the incdir path.

The `HPRLS` environment variable may be used to specify another search path for core parts of the system, but this would needs better documentation ...

36.4 Espresso

The traditional unix espresso tool is not needed for Fsharp implementation of HPR L/S since this has its own internal implementation.

The Moscow ML implementation of the Orangepath tool required 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.

37 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`.

It may also be programmed to retain other named features of interest.

38 HPR Command Line Flags

The very first args to an HPR/Orangepath tool are the early args that enable the receipe file to be selected and the logging level and location 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 `-verboselevel=n` turns on diversion of log file content to be mirrored on the standard output. 0 is the default and 10 makes everything also come out on the console. Console writes are flushed after each line and this is also a means of viewing the final part of a log that has not been flushed owing to stdio buffering.

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.

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

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

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

Synthcontrol `-bevelab-repack-pc=disable` creates sequencer encodings where the PC ranges directly over the h2 line numbers: easier for cross-referencing when debugging. Otherwise it defaults to a packed binary or unary coding depending on `-bevelab-onehot-pc`.

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

```
-vnl-resets=none
-vnl-resets=synchronous
-vnl-resets=asynchronous
```

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

```
<defaultsetting> resets none </defaultsetting>
```

When doing RTL simulation of the KiwiC-generated RTL output, one can sometimes encounter a 'lock up' where the design makes no further progress. Tracing the 'pc' variable in the output code will reveal it is stuck when trying to make a conditional branch whose predicate evaluates to dont-care owing to un-initialised registers or disconnected inputs.

HPR (KiwiC) (by default) does not generate initialisation code to set static variables to their default values (zero for integers and floats and false for booleans). The same goes for RAM contents.

For RAM contents, with KiwiC, the user code must contain an explicit clear operation in a C# loop.

To overcome the problem with uninitialised registers, we can potentially use -vnl-resets=synchronous or -vnl-resets=asynchronous. This will make the RTL simulate properly and overcomes most lockup problems. But we get additional wiring in the output that can repeat the FPGA's own hardwired or global reset mechanisms.

Clearly the design can be synthesised separately with and without resets. But to avoid the duplication of effort, hence with a common RTL file (one synthesis run only), one must take one of the following five routes, where the first two use a KiwiC compile with the default -vnl-resets=none.

1. use an RTL simulator option that has an option where all registers start as zero instead of X,
2. add a set of additional initial statements to the generated RTL that are ignored for FPGA synthesis (HPR vnl could generate these automatically but does not at the moment),
3. request a reset input to the generated sub-system (using -vnl-resets=synchronous) but tie this off to the inactive state at the FPGA instantiation of that subsystem and expect the FPGA tools to strip it out as redundant logic so that it does not consume FPGA resource.
4. trust the FPGA tools to detect a synchronous reset net as such (by boolean dividing FPGA D-input expressions by it) and map it to the FPGA hardwired reset mechanisms so that it does not consume FPGA resource.
5. use -vnl-resets=asynchronous and trust the FPGA tools to map this to the hardware global reset net.

Note, the vnl output stage always generates subsystems with a reset input but this is (mostly) ignored under the default option of -vnl-resets=none.

See § **??**.

```
"-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`.

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

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

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

38.1 Other output formats

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.

38.2 General Command Line Flags

The `-version` flag give tool version and help string.

The `-help` flag give tool version and help string.

38.3 HPR L/S (aka Orangepath) FAQ

Q. I get the error

`Error: Could not load file or assembly 'FSharp.Core, Version=4.4.0.0`

A. This is not related to any missing files in the Kiwi distro. Instead it is do with FSharp version incompatibilities. The FSharp.Core is part of the FSharp system. If you are using pre-built dll files then the version of mono or FSharp on your system may be incompatible with the pre-built dll files and you would have to change version or else regenerate the dll files by compiling the FSharp source code with the 'fsharpc' compiler on your system.

You may wish to just compile a trivial 'Hello World' FSharp program on your system to check that FSharp is all set up ok.

39 HPR System Integrator

Q. I cannot see how to start using System Integrator?

A. SystemIntegrator is a standalone program, written in FSharp and using the HPR library. It is in its own folder (and the binary is hprls/system-integrator/distro/lib/sysint.exe). Examples are currently missing.

The HPR System Integrator compiler/generator takes a set of HPR VMs and generates SP_RTL constructs to wire up their ports following the VM instantiation pattern or an input IP-XACT document. It will instantiate protocol adaptors and glue logic based on pre-defined rules.

The resulting system can then be emitted without the actual instances using other recipe stages, such as SystemC, RTL or IP-XACT. These output files will typically be combined with the instantiated components in external tools, such as FPGA logic synthesis.

The resulting system can also be passed on to the Diosim simulator for execution within Orangepath, for auditing tools to run, or for any other purpose.

Figure 10 illustrates a typical structural set-up arising from multiple compilation units assembled on a single FPGA. In detail, the figure shows a top-level application (primary IP block) that instantiates a separately-compiled child component that, in turn, instantiates three grand children of two different types. The children and grand children are subsidiary IP blocks. They do not do anything unless commanded by a primary IP block. Each compilation unit connects to its child by an arg/result port that is of a custom design for the current application. It is application-specific (A/S).

In addition, each child component requires access to RAM resources. In this particular example, the top-level module did not require RAM access (although it could well have its own BRAM privately instantiated).

Finally, every component has a directorate port for error reporting. The primary IP block also receives its run/stop control via this port.

The HPR System Integrator compiler takes a set of HPR VMs and generates an hierarchic netlist to wire up their ports using pre-defined rules that are based on the concept of domains of connection. It will instantiate as many protocol adaptors, bus switches and arbiters as is needed. The resulting structure is typically rendered as RTL. In the future it can invoke Greaves/Nam glue logic synthesis or other generators and then instantiate the glue in the netlist.

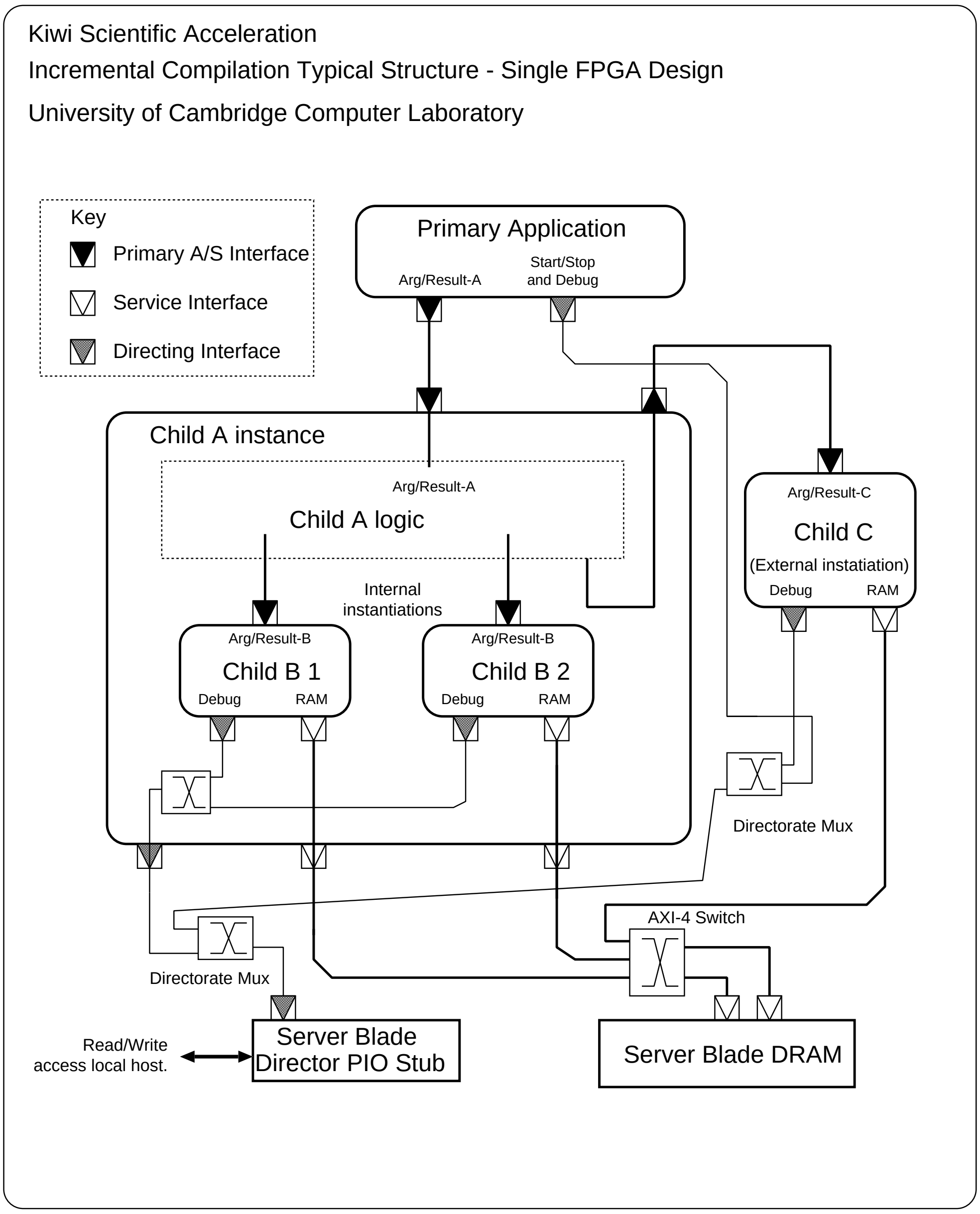


Figure 10: Example of multi-compilation structural assembly with internal and external instances.

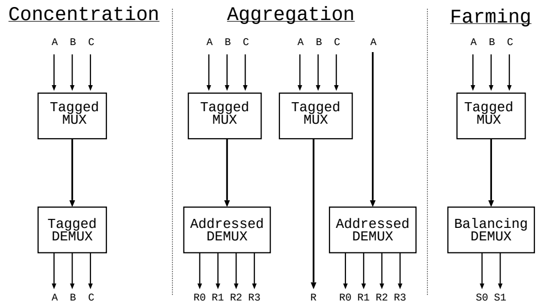


Figure 11: System Integrator Tool: Basic Auxiliary Components.

The resulting system can then be emitted without the actual instances using other recipe stages, such as SystemC, RTL or IP-XACT. These output files will typically be combined with the instantiated components in external tools, such as FPGA logic synthesis.

The resulting system can also be passed on to the Diosim simulator for execution within Orangepath, for auditing tools to run, or for any other purpose.

Its internal datastructure, prior to rendering the output, is in a form that can be output as IP-XACT `spirit:design` document.

A future facility to read in and obey IP-XACT `spirit:design` documents could easily be added, but there are plenty of third-party tools offering that service.

HPR System Integrator supports:

1. Creating inter-module wiring structures with tie-off of unused ports.
2. Working both at the TLM level and structural net list level.
3. Glue logic insertion in the form of instantiated adapators from the library are readily inserted automatically using rules based on interface type differences.
4. Allocation of AXI tag numbers.
5. Custom glue logic from the Greaves/Nam cross-product technique can also be rendered.
6. Outputs are rendered in Verilog, IP-XACT, SystemC TLM, SystemC behavioural and SystemC RTL-styles depending on the subsequent recipe stage the output is passed to.
7. Server farm mode supporting dynamic dispatch will be added during 2017.

A component instance can be internal or external. The distinction is more pronounced for RTL than SystemC. External instantiation is where the instance is inside the current (instantiating) module, in the style of a traditional hierarchic design. An external instance is instead formed outside the current module, resulting in additional bindings in the signature of the current module. External instantiation leads to a flatter design. Its principle advantage is where the instantiated component has a number of service ports whose bindings would instead need to be conveyed through the current instance signature.

The HPR System Integrator rule engine understands the following types of component:

- Primary IP Block — a top-level component of the design, such as a primary output from Kiwi HLS, that embodies an algorithm or processes and generates work for the all the other components.
- Subsidiary IP Block — an IP-block with slave ports that performs an operation. Examples are RAMs, ALUs and HLS outputs from earlier parts of an incremental compilation process.
- External Port — a connection to an externally-instantiated resource, such as a DRAM bank, inter-FPGA bridge or Ethernet port.
- Aggregators — for arbitrating and dispatching between initiators and demultiplexing based on addressed target
- Concentrators — for tagged multiplexing and demultiplexing over a shared channel.
- Protocol Adaptor — for converting between bus standards

Every block is accompanied with non-functional meta-info that gives an area, latency, throughput and energy cost using IP-XACT extensions.

Every external block port and port on a primary IP block must also be manually given a so-called domain name. The standing rules used by HPR System Integrator endeavour, for each domain, to wire everything together, thereby achieving conservation of data. There will generally be at least one domain name for each connection between separately-compiled modules in an incremental compilation. Also, there will be domains associated with each disjoint memory map/space and one for the debug/directing logic.

The system synthesis is guided by a goal function, which is a scalar metric that factors area, delay and energy according to a weights that the user can adjust as desired.

The automatic generation axioms are:

1. The number of primary IP blocks and external ports is set in the initial configuration, together with their instance names. Their plurality may not not be adjusted by HPR System Integrator.
2. The plurality of all other components may be freely adjusted by HPR System Integrator, but it may not replicate state-bearing components (unless they have mirror rules defined in the future).
3. Except for broadcast connections (that have no reverse direction signals), such as clocks, resets and status codes, all initiating ports must be connected to a matching target port with a one-to-one direct connection.
4. The IP-XACT max-masters and max-slaves attributes Ports are either multicast or one-to-one. They may have to be connected or may be left disconnected. All initiating ports must be connected to a matching target port with a one-to-one direct connection.
5. The resulting design should give a low value for the goal function.

   This will tend to minimise the number of additionally instantiated components and typically causes them to be wired in tree-like structures to minimise latency.

Per domain metric functions and upper bounds

Algorithm: for each domain name, while there is an unconnected initiator, create a connection for it to a suitable serving resource. If the serving resource is an external port that is currently disconnected, a direct connection can be made. But if the external port is already bound, an additional bus switch will be instantiated or the arity of an existing one will be increased.

If the serving resource would be an instance of replicatable IP block, ...

If the serving resource would be an instance of mirrorable IP block, ...

39.1 Memory Map Management (Link Editing)

A shared memory resource that is serving a plurality of disjoint requirements needs memory management to statically or dynamically allocate disjoint memory to each component. This is essentially a link editing problem.

Kiwi solves this in two ways. For static allocation in each bank, HPR System Integrator reads in from IP-XACT how much static memory is required and supplies a base address as an RTL parameter to each instantiated component. This base address is promolgated into the core of the logic by constant propagation in the logic synthesiser (FPGA tool) that is applied to the KiwiC output.

For dynamic allocation, an allocator component, coded in C# must implement a free pointer or equivalent policy, be instantiated once, and serve out memory blocks. This will require unsafe C# in each client (or shim thereof) to cast the address to the required struct or object type. Only the alloc/dealloc requests need be sent to the shared component: the data read and write transfers themselves are transferred over a general the AXI switch fabric that can provide as much spatial diversity as is appropriate.

For genuinely shared pools there will inevitibly be a C# module that directs the requests for WD development and this must be separately compiled and connected to by multiple parent IP blocks.

For multiple address spaces it is convenient to add extra fantom bits ...

39.2 Deadlock and Combinational Paths

... TBD

39.3 Constructive Placement

The general flow for the tool is illustrated in Fig. 12. Its inputs are the name of a primary IP block for the top-level, a search path for lookup of the so-called subsidiary and auxiliary IP blocks, and a description of the target platform described in a file `blade-manifest.xml`. The tool operates in two stages. The first is a planner that makes floorplanning and memory layout decisions and instantiates subsidiary and auxiliary blocks as needed. The resulting high-level design is written out as an IP-XACT design report, a graphical plot and a human-readable report that tabulates utilisation metrics. The second stage compiles the design to a structural netlist. This writes out a master RTL file for each FPGA.

The blade manifest lists the number of FPGAs available on the platform, describing their size, inter-

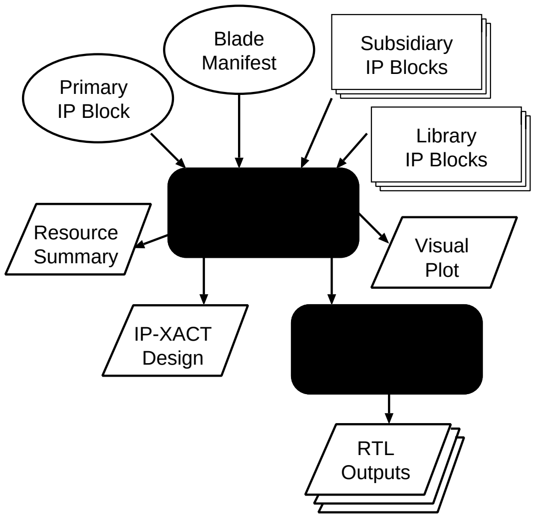


Figure 12: System Integrator Tool: Inputs and Outputs and flow between the two stages.

connection pattern and hardened IP ports and capabilities. It is an XML file crafted by hand or using an XML editor.

The tool can potentially use any standard optimisation procedure to minimise its global cost metric. The current implementation uses a constructive placer that is run about 50 times using different pseudo-random seeds with the best solution and spread being reported. A critical consideration is whether any IP blocks themselves are good candidates for consequential re-synthesis. There are three reasons for re-synthesising a component:

1. **General time/space fold:** Standard HLS tools have considerable freedom to produce large and fast designs or smaller designs that require a greater number of clock cycles.
2. **Degree of Port Mirroring:** Where a subsidiary block can be mirrored, the parent needs to be synthesised with a determined number of master ports when these are connected one-to-one with the children. Moreover, the number of load, store and load/store stations on the component can also be manually controlled with our tool.
3. **Move to variable-latency handshakes:** Where a block instantiates a fixed-latency child connection, but then that connection has to be converted to variable-latency owing to inter-FPGA bridges (or perhaps being in a server farm in the future).

The System Integrator's main job is to generate a design that includes the primary IP block and all the support it needs. Starting from the primary IP block, it adds the subsidiary IP blocks referred to in its port list. These may have further application-specific ports (as shown in Figure 10) that in turn need to be supported. Hence it iterates at this stage. Using its constructive placer, it puts each block on a named FPGA where there is sufficient area remaining. Connections that span multiple dies have their necessary protocol adaptors instantiated straightaway. Where a bridge link is shared between bus connections, concentrators are added (addressing tags are later created in a global colouring step). Any placement attempt where any hard limit is breached is aborted without further study.

Hard limits include any FPGA being full, as just mentioned, or a guaranteed throughput or latency (sequential or combinational) cannot be met.

As illustrated in Figure 10, there are three forms of bus connection understood by System Integrator:

1. A **Primary Application-Specific Interface** enables a component to invoke functions using a custom bus structure on a child component that has a reverse interface of the same type. In our HLS system, such bus specifications are emitted automatically as augmented IP-XACT bus abstraction documents. The same file is emitted when either side is compiled, with the second simply overwriting the first. When the boundary reflects a class definition in the high-level language, the file name and interface name are the same as the class name. Such a class can have any number of methods and each method will use some set of the busses (or 'ports' as they are called in IP-XACT) making up the interface. This sort of connection is also used for connections to the standard libraries of maths functions.

2. A **Service Interface** provides access to main memory resources for the component. The component is free to instantiate its own RAMs where it wishes, such as FPGA block RAM, but larger regions need wiring to DRAM resources. These are either statically instantiated on the server blade or else accessed over AXI or PCIe on some platforms.

3. A **Directing Interface** provides start/stop control of the primary application and collects status and abnormal end codes from subsidiary blocks. It may also provide debug inspection.

A connection between two components is valid when all of the following conditions hold:

- **Kind Name**: the protocol kinds have the same name. Differences in the other three IP-XACT naming attributes, vendor, version and library name, are warned about but otherwise ignored.

- **Connection Rule**: A one-to-one connection must have two peers: one an initiator and the other a target. A multicast connection must have exactly one initiator.

- **Parameters Match**: IP-XACT parameters are key/value pairs, and these must match apart from any that the user specifically annotates (on the command line) as allowed to mismatch. This ensures, for instance, that a 32-bit data bus is not connected to 64-bit data bus. To overcome simple mismatches of any complexity, one side needs to be manually renamed by the user and an additional protocol adapator added on the search path that encompasses the adaption, such as ignoring unused address bits. Automation of this is expected in the future.

- **Unified Domains**: The connection domains must either already match under the current unification or else a fresh, non-contradictorary, unification is added for the remainder of the design construction.

The domain unifier operates over equivalence classes that contain at most one domain constant and any number of domain variables that are unified to that constant or just to each other when a domain constant is not present.

When a protocol adaptor is instantiated, it is given a fresh domain variable that is allocated to both ports.

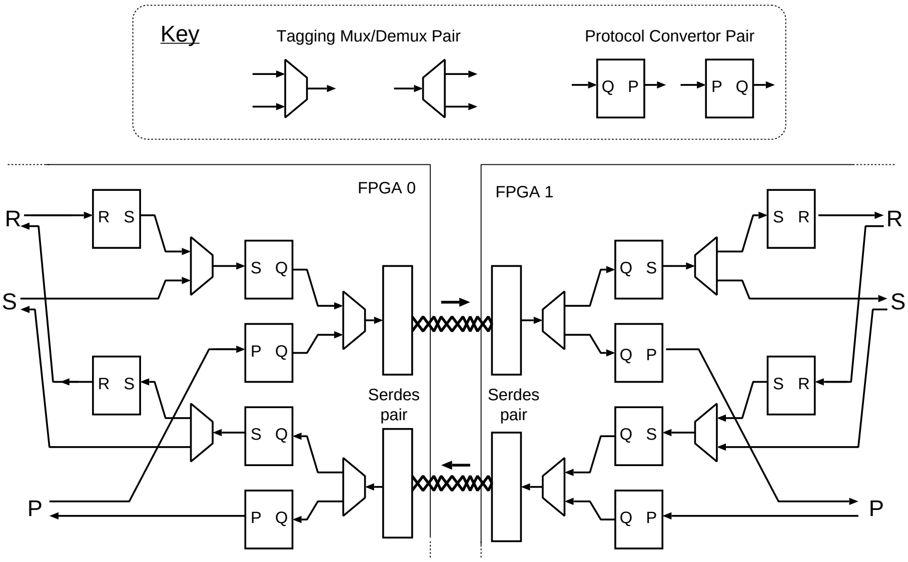


Figure 13: Inter-FPGA bridge structure: typical setup. The SERDES instances, as described manually in the blade manifest, are utilised by the System Integrator's instantiation of protocol adaptors and concentrators as required.

39.4 Multi-FPGA designs

HPR System Integrator can allocate logic between FPGA chips.

As illustrated by the SERDES pair in Figure 13, inter-FPGA bridges are bi-directional and have four ports for binding by the System Integrator as it creates an inter-FPGA network. The two ends of each simplex channel have the same domain name, but the bandwidth and latency for the two channels can be described differently in the associated IP-XACT description. Each of the four bus interfaces is AXI streaming with a specified word width, giving the lossless FIFO paradigm. Each direction of the pair is kept matched by the System Integrator, as it adapts the hardware resource to its needs. The adaption steps are just the same as may be freely used elsewhere in the assembly: they are inserting a protocol adaptor pair on each side or inserting a concentrator pair consisting of a tagging mux and an inverse de-multiplexing component that processes and removes the tags. There is a set of standard protocol adaptors corresponding to all basic method signatures of up to 3 arguments with and without a result in our standard distribution. Others can be created by hand as needed and added to the library, or they can be macro-generated on demand in the future. Glue logic for these purposes can also be synthesised from a non-deadlocking, data-conserving product of protocol state machines by known techniques, such as [2].

39.5 Mux and Demux Blocks

Figure 14 shows three use cases involving tagged multiplexing and differing demultiplexing approaches. The arrows in the Figure indicate direction of initiation, but each underlying bus can normally carry data in either direction according to the whether read-style or write-style operations

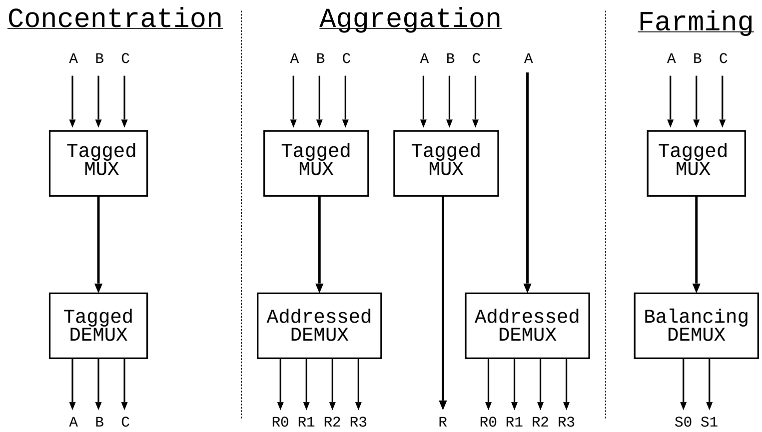


Figure 14: Three forms of multiplexing/demultiplexing where the demultiplexors respectively uses tags, addresses and utilisation to make a routing decision.

are currently being conveyed.

The left of the Figure shows straightforward **concentration**, where multiple logical channels are conveyed over a shared physical channel. A tagging multiplexor is matched with a detagging demultiplexor. The tags inserted at the top are removed at the bottom and are private to the configuration. This configuration provides perfect data conservation with respect to the mulitplexed channels from the point of entry at the top to the point of exit at the bottom.

The centre of the Figure shows shared access to a logical address space by a number of initiators where the address space is served by an **aggregation** of physical memory resources. The demultiplexor operates using address ranges. The multiplexor still inserts tags, but this time these are removed again *also* by the multiplexor. These tags are only examined by the multiplexor that created them: it removes them when the result is forwarded upwards to the originator. The tags are conveyed opaquely within all lower components. Two degenerate forms of the aggregation configuration arise: 1. when there is only a single client for an aggregated resource, the multiplexor is not needed; and 2. when the resource is monolithic the demultiplexor is not needed.

The right of the Figure illustrates the **server farm** configuration, that again uses a tagging multiplexor, but the demultiplexor operates on a load-balancing basis. The server farm is not currently natively supported by HPR System Integrator. Instead, the user must implement this paradigm by writing their own implementations of the multiplexor and demultiplexor. This is easy to do in C# for synthesis to RTL by KiwiC. If the C# is marked up for separate synthesis of the relevant components, the HPR System Integrator will then assemble the system, treating the farming blocks as subsidiary IP blocks to be assembled as normal. An example will be placed here ... TBD.

39.6 Non-uniform Memory Access (NUMA)

It is desirable for traffic to take the shortest route between layout zones. The HPR System Integrator implements Warshall's algorithm to find available routes and to price design solutions that use them. However, Figure 15 shows, on the left, the typical structure that arises when static resources in two

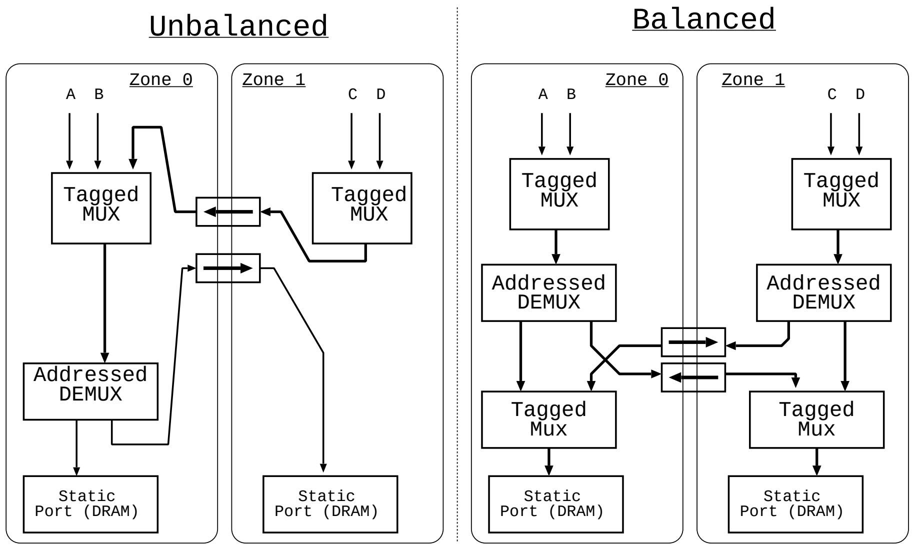


Figure 15: Illustration of non-uniform memory access designs: Left is current implementation style and right is preferred style.

different zones are aggregated and then shared by clients where the clients (A, B, C and D) are also distributed over the zones.

The right-hand side of the figure illustrates a preferred design that is typically exploited in non-uniform memory architectures (NUMA). Although this has a little more logic, the average access latency for Zone 1 is improved.

The HPR System Integrator operates by first creating the required data paths as a rats' nest without regard to layout zone. As mentioned in §**??**, it then inserts bridges and concentrators as it maps that network onto the layout zones. This leads to the left-hand style of design. To achieve the preferred design, greater smartness is needed: a spatially aware design is needed from the outset. We aim to address this in a subsequent release of the tool.

Warshall's algorithm is also applied to protocol adaptors in the library, to see what can be connected to what in principle and the best pattern of adaptors, giving each adaptor a unit cost at this time. // We must avoid building wandering chains that convert backwards and forwards between protocols, but as Warshall considers each protocol a node in a multi-hop journey, it will only instantiated at most one of each type of adaptor in a path.

39.7 Network On Chip (NoC)

It is interesting to examine whether HPR System Integrator can be said to be synthesising a Network-on-Chip (NoC).

Although there may be no absolute definition of what constitutes a NoC, the following defining principles can be identified:

- **Connectivity**: Data can be sent, in principle, from ingress node to any exit node.

- **Sharing**: Traffic for different purposes uses a common bus infrastructure with sharing in the time domain.
- **Route Diversity**: Traffic may take various routes from a source to a destination determined by some static or run-time decision or policy.

The HPR System Integrator will make a custom mesh network as it instantiates concentrators to exploit shared inter-zone bridges. So it does sometimes generate a NoC using the ‘sharing’ principle. Where the inter-zone bridges are arranged just as a physical ring, then the resulting network is a ring network (being a degenerate form of mesh). The ring is bi-directional or uni-directional, in terms of instantiation, according to the same property in the pattern of the available bridges. But each bridge is bi-directional in data terms, in that responses are carried in the reverse direction over the bridge that carried the request. Overall, there is currently no route diversity.

In the future, for large FPGAs, it is sensible perhaps to divide them into several layout zones, perhaps with fluid boundaries where area can be vired between zones. It will then be neccessary to instantiate inter-zone bridges in the blade manifest between these zones. Such bridges will be nothing more than point-to-point wiring, which will be totally reified by the back-end logic synthesis tool, so there is no run-time overhead. The advantage is that the pattern of concentrators will closer resemble a fine-grained NoC and the generated wiring will resemble ...

39.8 Bus Definitions

Bus definitions in IP-XACTare split over two files. The definition and the abstraction. ....

All IP-XACTfiles needed will be search for using the path specified with `ip-incdir=`. This is a colon or semicolon (on Windows) separated list of folders.

39.9 Sewing Kit for Miscellaneous Nets

Any hardware design will have a few extra nets (wires) that System Integrator does not need to understand. Since the tool emits the top-level design file on each run, the simplistic approach would be to reinsert such logic with a sed or perl script. This is a bit messy. Nonetheless, there are some `SED_BIND_POINT` tokens emitted as comments in the rendered output to assist with such an approach.

A more general support mechanism called the Sewing Kit will be added to direct System Integrator to emit miscellaneous nets in the future.

39.10 System Integrator Example Run

Under construction May/Sept 2017...

The HPR System Integrator application is invoked from the command line using the shell script found in `HPRLS/system_integrator/dist/bin` called `sysint` that invokes the `sysint.exe` portable assembly under mono.

You need to also minimally supply

```
-blade-manifest=my-manifest.xml
-root=rootipname
-ip-incdir=folder1:folder2:folder3
```

For a single zone, you can set the Verilog output file name with `vnl=filename.v` but for multiple zones, you are better setting the output folder name with `-obj-dir=outputfolder` and allowing the tool to create its own output files. The output files follow a built-in naming scheme that extends a root name which defaults to 'roger' and which can be set to something else with '-outroot=fred' etc..

All flags can also be set instead from the Organgepath recipe by extending the default `sysint00.rcp` file and redirecting to the extended copy with `-recipe=myrecipe.rcp` which needs to be (pretty much) the first argument after the `mono sysint.exe` part of the overall command line.

The HPR System Integrator generates (by default) a graphical plot like this early example:

The primary and subsidiary I/P blocks are in black. Concentrators are in green. Adaptors are in brown.

For complex designs, a separate plot for each layout zone is helpful. This is achieved with ...

40 Diosim Simulator

The HPR L/S library provides a built-in simulator called Diosim. It is intended to be able to execute any mixture of intermediate codes since all have executable semantics.

Diosim is invoked by the recipe. Typically a recipe may invoke it on the same intermediate form that is being rendered as RTL or SystemC etc..

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 a hierarchical set of VM2 machines and simulates them and their interactions.

The simulator will dynamically validate 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 to files. Several files are generated:

- A log file where individual events are visible if logging level is set high enough, eg. with `diosim-tl=100`.
- A plot file. The plot file is currently in diogif plot format.
- A VCD file - viewable with gtkwave and/or modelsim etc..
- A console spool file, typically called `diosim.out`.

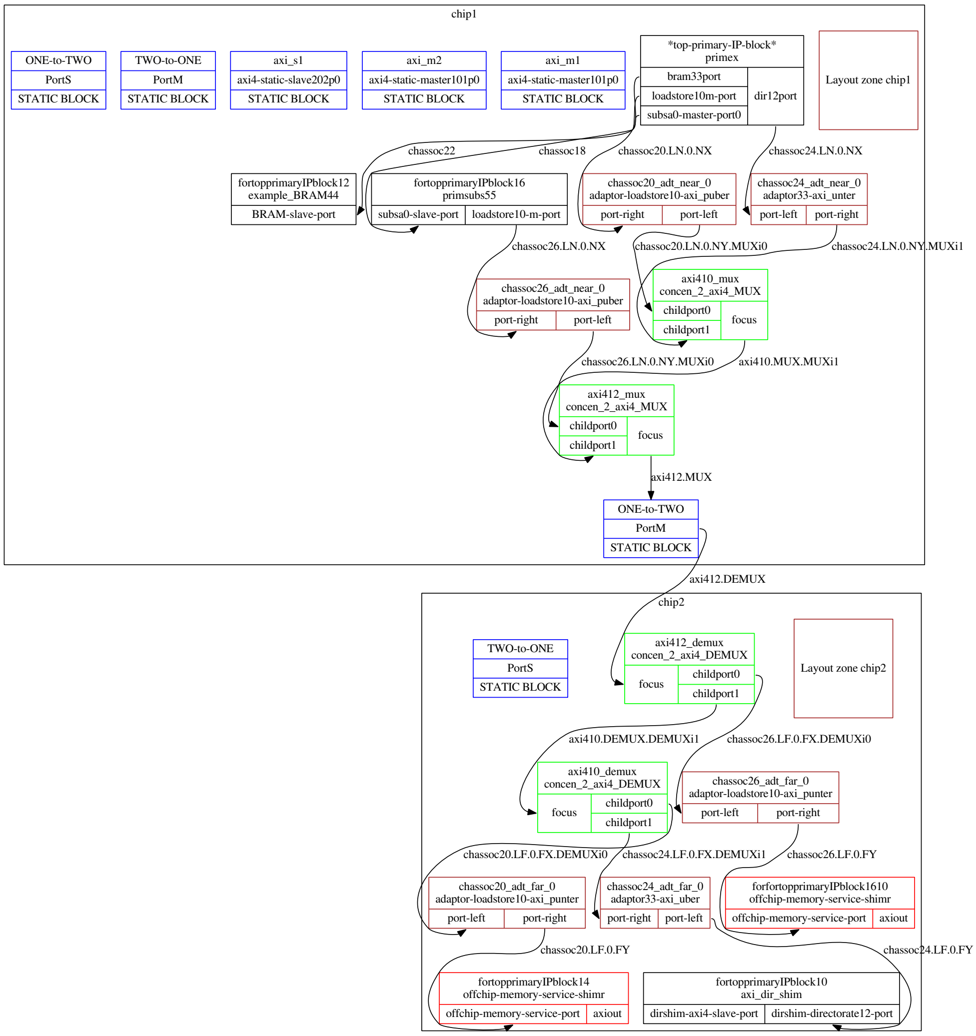


Figure 16: Draft. Output via ‘dot’ of the inter-IP block wiring generated in an example System Integrator run. Two layout zones were used, corresponding to two FPGAs.

40.1 Simulation Control Command Line Flags

As well as providing simulation output in VCD and console form, diosim can collect statistics and help with profile generating. However, it is fairly slow and it is best to collect profiles from faster execution engines, such as via Verilator.

The statistics that diosim can collect range from net-level switching activity to higher-level statistics like imperative DIC instructions executed, RTL sequential and combinational assignment counts.

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 `-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 in diogif format.

The plot file can be viewed under X-windows and/or converted to a gif using the diogif program.

The `-diosim-vcd=filename.vcd` flag causes diosim to write a Verilog Change Dump (vcd) report to the named file.

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 dynamically.

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

KiwiC using C++ instead of C#

Visual Basic, 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>/` directory. To create a CIL file use this compiler with the -S option.

Getting gcc4cil.

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

  2. As Gcc4Cil wants to compile files for the Mono-platform, you
need the Mono-project installed on your system. The easiest way to
install it is to use "Linux installer for x86" that can be found
under http://www.mono-project.com/Downloads . 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 becaus
     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 so 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 wont 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/
     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.
```

Nov 2016 note: The main gcc4cil problem was a lack of any sort of linker, as I recall. I do not recall why a linker was critical since KiwiC and dotnet are both happy to accept multiple dll files. Perhaps there was a related problem with .h files. I don't know whether gcc4cil maintenance is now abandoned.

Of course Visual C++ produces dotnet code that should work pretty much as well as the recent Visual Basic demo. I don't know how much Visual C++ resembles standard C++ or whether it can only be compiled on windows.

All of the HPR recipe stages except for the first, kiwife, are independent of dotnet. The intermediate HPR VM forms between recipe stages are all supposed to be serialisable to disk: you use recipe files that start and end with a load and save of VM code. But that facility has not been used recently. It might become important again to help overcome long monolithic compile times.

References

[1] Francesco Bruschi and Fabrizio Ferrandi. Synthesis of complex control structures from behavioral SystemC models. *Design, Automation and Test in Europe*, pages 112 – 117, 2003.

[2] D. J. Greaves and M. J. Nam. Synthesis of glue logic, transactors, multiplexors and serialisors from protocol specifications. In *2010 Forum on Specification Design Languages (FDL 2010)*, pages 1–7, Sept 2010.

[3] David Greaves and Satnam Singh. Kiwi: Synthesis of FPGA circuits from parallel programs. In *The 16th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2008.

[4] R. Passerone, L. de Alfaro, T. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the International Conference on Computer-Aided Design*, November 2002.

[5] A.M. Zaidi and D.J. Greaves. A new dataflow compiler IR for accelerating control-intensive code in spatial hardware. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 122–131, May 2014.

Index