

Kiwic: .NET to RTL compiler
User Manual
Very First Tentative Draft

April 2, 2012

Preface

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

Kiwi is developing a methodology for hardware design using the parallel programming constructs of the C# language. Specifically, Kiwi consists of a run-time library for native simulation of hardware descriptions within C# and a compiler that generates RTL from stylised .NET bytecode.

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 There is as much wrong and outdated information in here as there is correct!

Contents

0.0.1	Background: HPR Library and Orangepath Tool	9
0.1	Get Started (Fsharp/Dotnet)	9
0.2	Old Get Started (MOSCOW ML)	10
0.3	Library Functions	11
0.3.1	Console.Write and Console.WriteLine	11
1	C# Attributes	15
1.0.2	Flag Unreachable Code	15
1.0.3	Hard and Soft Clock Control	15
1.0.4	Loop NoUnroll Manual Control	16
1.0.5	Elaborate/Subsume Manual Control	17
1.0.6	Offchip or Output Memory Array Mapping	17
1.0.7	Hardware Server	17
1.0.8	Register Widths and Wrapping	17
1.0.9	Input and Output Ports	18
1.0.10	Clock Domains	19
1.0.11	Remote	19
2	Synthesisable Language Subset	21
2.1	FAQ	21
2.1.1	Accessing Simulation Time	22
3	Orangepath Synthesis Engines	23
3.1	A* Live Path Interface Synthesiser	23
3.2	Transactor Synthesiser	23
3.3	Asynchronous Logic Synthesiser	23
3.4	SAT-based Logic Synthesiser	23
3.5	Bevelab: Synchronous FSM Synthesiser	24
3.5.1	Bevelab: Internal Operation	25
3.6	PSL Synthesiser	25
3.7	Statechart Synthesiser	25
3.8	SSMG Synthesiser	25
3.9	Restructure Synthesiser	26

4	Output Formats	27
5	General Orangepath Facilities	29
5.1	FILES AND DIRECTORIES	29
5.2	Espresso	29
5.3	Cone Refine	29
5.4	HPR Command Line Flags	29
5.4.1	Other output formats	31
5.4.2	General Command Line Flags	32
5.4.3	Simulation Control Command Line Flags	32
5.5	Diosim Simulator	32

Overview Summary

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 1 shows the main flow through the tool as set up with the provided recipe file (kiwic00.rcp). This is a six stage recipe and the obj folder created by running the tool contains the log files and intermediate forms for each stage. Other output flows and formats can be deployed by changing the recipe. The dotted line shows that using the `simvnl` command line option the internal simulator (Diosim) can be applied to the RTL after it has been round-tripped through Verilog. For debugging, Diosim can be applied to any HPR machine intermediate form, by varying the recipe. (There's also a shortcut '-conerefine disable -repack disable -verilogen disable' that will cause diosim to run the original VM generated by the Kiwic front end without conversion to hardware).

The .NET executable is read using the Mono.Cecil front end and combined with some canned system libraries as an abstract syntax tree.

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

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

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

The conversion to FSM is described in §3.5 and is common to the h2tool flow.. Actually, that is the old scheduler and a new one is now being used.

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

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

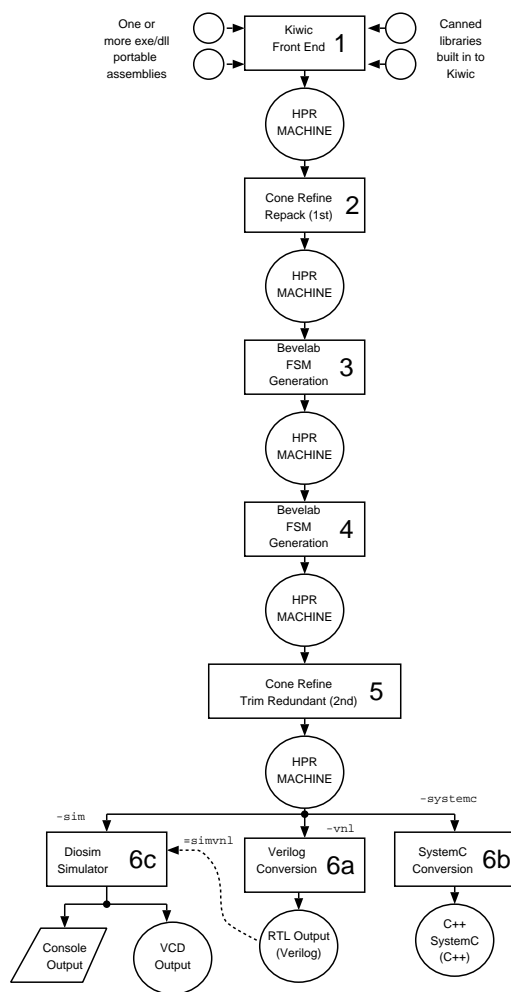


Figure 1: The main flow using the default recipe (kiwic00.rcp) in the Kiwic tool.

0.0.1 Background: HPR Library and Orangepath Tool

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

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

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

The library is structured as a number of components that operate on a VM to return another VM. The opath mini-language enables a ‘recipe’ to be run that invokes a sequence of library operations in turn. An opath recipe is held in an XML file and the default file is `kiwic00.rcp`.

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

The opath core provides command line handling so that parameters from the recipe and the command line are fed to the components. The opath core also processes a few ‘early args’ that must be at the start of the command line. These enable the recipe file to be specified and the logging level to be set.

The Orangepath library has a number of supported input and output formats.

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

0.1 Get Started (Fsharp/Dotnet)

Kiwic is supplied as a zip file that contains folders called lib, bin, doc and so on.

Requirements: Kiwic/HPR is now implemented in Fsharp. You need a working dotnet environment (mono or windows) on your machine and if Fsharp is not locally installed you will need to add at least the FSharpCore.dll added to the kiwic/lib folder.

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/lib folder.

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

```
$ $(PREFIX)/bin/kiwic ... args ...
```

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

Windows users can invoke the `kiwic.exe` executable directly.

The arguments to kiwic should either be portable assembly files (suffix .dll or .exe) or option flags prefixed with a minus sign. Generally you will supply the current design and also the kiwi.dll library as arguments.

Two common flags, always needed in the past, were `-root` and `-vnl`. But now the latter is supplied in the default recipe and instead of using the `-root` command line flag, the `Kiwi.Hardware` attribute can be put on a classes in the input program. Both methods can be used at once: all marked classes are converted to hardware, however specified.

To obtain Verilog RTL output, Kiwic used to require a source file name, a root point, saying what to compile from the source file and an output verilog netlist file name. So the most basic command line is something like

```
$ $(PREFIX)/bin/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;
class tiny
{
    public static int Main (string []argv)
    {
        Console.WriteLine("Hello World");
        return 1;
    }
}
$ gmcs tiny.cs
```

NB: Kiwic will write a disassembly of the pe file to `obj/ast.cil` in the current folder.

0.2 Old Get Started (MOSCOW ML)

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

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

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

The `kiwic` tool also requires the `mono` directory.

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

Before use, please set the `MOSML` and `HPRLS` shell variables

At the computer lab use:

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

`HPRLS` should be set to your copy of the H2 tool. DJG uses:

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

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

If you get this error:

```
Uncaught exception:
```

```
Fail: libmunix.so: cannot open shared object file: No such file or directory while l
```

then the setting of LD_LIBRARY_PATH is not working.

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

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

or

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

also

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

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

```
$(HPRLS)/mono/kiwic TimesTable.il -root 'TimesTable;TimesTable.Main' -vnl TimesTable.v
See also $HPRLS/mono/README
```

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

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

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

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

0.3 Library Functions

Kiwic supports a number of standard library functions, such as `Console.WriteLine`, as well as working with the Kiwi project library that contains functions such as `Kiwi.Pause`.

0.3.1 Console.Write and Console.WriteLine

`Console.Write` and `Console.WriteLine` are supported, producing the equivalent operations in the generated RTL or SystemC. The maximum number of arguments supported is three (because beyond that the C# compiler allocates a local array). The positional style argument references using braces are supported, as well as limited forms of the string concatenation style using the plus operator.

Kiwic Internal Operation

Skip this section if you do not feel you need to know about the internal workings of kiwic.

CIL code is the assembly language used by the mono and .NET projects. The HPR tool can read in CIL assembly code when invoked using the `kiwic` command. This assembly code is generated by a large number of third party compilers from various input languages. All dynamic storage allocation must be made by constructors before the entrypoint method is called. There are numerous other limitations on how the CIL code must be structured.

This stage reads CIL bytecode and generates VM code for the HPR virtual machine. The CIL bytecode is parsed to an AST by a bison parser.

A variable is a static or dynamic object field, a top-level method formal, a local variable, or a stack location. For each variable we decide whether to subsume it at the CIL processing stage. If not subsumed, it appears in the abstract VM code that is fed to the next stage (where it may then get subsumed for other reasons). Variables that are subsumed in this way tend to be object and array handles. Such variables must contain compile-time constant values throughout the execution of the output code.

We perform a symbolic execution of each thread at the CIL basic block level and emit VM code for each block. CIL label names that are branch destinations define the basic block boundaries and these appear verbatim in the emitted VM code.

Although CIL bytecode defines a stack machine, no stack operations are emitted from the CIL processing stage. Stack operations within a basic block are symbolically expanded and values on the stack at basic block boundaries are stored and loaded into stack variables on exit and entry to each basic block. The stack variables are frequently subsumed, but can appear in the VM code and hence, from time-to-time, in the output RTL.

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

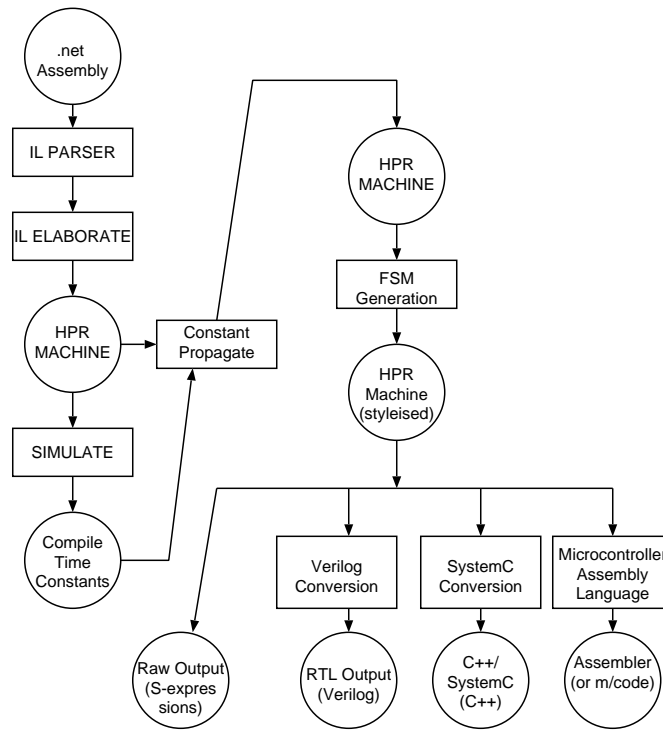


Figure 2: The main flow implemented in the kiwic tool (same as figure ??).

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.

At some point, Kiwic might be extended to also invoke the gmcs C# compiler if given a C# file.

Chapter 1

C# Attributes

The **Kiwic** compiler understands various .NET assembly language custom attributes that the user has added to the source code. In this section we present the attributes available. These control things 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/attribute-definitions` in the distribution.

The Kiwi attributes can be used by referencing their dll during the C# compiler.

```
gmcs /target:library mytest.dll -r:attribute_definitions.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/attribute_definitions.cs`.

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

1.0.2 Flag Unreachable Code

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

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

1.0.3 Hard and Soft Clock Control

Many net-level hardware protocols are intolerant to clock dilation. In other words, their semantics are defined in terms of the number of clock cycles for which a condition holds. A thread being compiled by Kiwic defaults to soft pause control (or other default

set in the recipe or command line), meaning that Kiwic is free to stall the progress of a thread at any point, such as when it needs to use extra clock cycles to overcome structural hazards. These two approaches are incompatible. Therefore, for a region of code where clock cycle allocation is important, Kiwic must be instructed to use hard pause control.

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

The current pause control mode of the current thread can be updated by calling ‘`Kiwi.SetPauseControl`’.

When a thread calls `Kiwi.SetPauseControl(hardPauseControl)` its subsequent actions will not be split over runtime clock cycles except at places where that thread makes explicit calls to `Kiwi.Pause()` or makes a blocking primitive call.

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

A third mode for a thread, called soft, is under development. `Kiwi.SetPauseControl(softPauseControl)`.

Finally, blockb pause control places a clock pause at every basic block and maximal pause control turns every statement into a separately-clocked operation `Kiwi.SetPauseControl(maximalPauseControl)`.

1.0.4 Loop NoUnroll Manual Control

Put a call to ‘`NoUnroll()`’ in the body of a loop that is NOT to be unrolled by kiwic.

If there is a ‘`Kiwi.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 variable)
  // 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.
}
```

1.0.5 Elaborate/Subsume Manual Control

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';

1.0.6 Offchip or Output Memory Array Mapping

The OutboardArray attribute indicates that an array is to be mapped to a region of external memory instead of being allocated a private array inside the current compilation.

1.0.7 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.

1.0.8 Register Widths and Wrapping

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

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

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

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

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

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

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

Compiling this example gives an error:

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

1.0.9 Input and Output Ports

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

I/O ports are inferred from static variables in top-most class being compiled. 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 VHDL 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[11];
[Kiwi.OutputWordPort(11, 0, "dvi_i")] public static int[] dvi_i = new int[11];
```

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;
```

1.0.10 Clock Domains

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

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

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

1.0.11 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 and **kiwic** generates additional I/O terminals on the generated RTL that implement a stub for the call. The currently implemented protocol is asynchronous, using a four-phase handshake and a wide bus that carries all of the arguments in parallel. Another bus, of the reverse direction, conveys the result where non-void. Further protocols can be added to the compiler in future, but we would like to instead lift them so they can be specified with assertions in C# itself.

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.wunit1'. These all have the same meaning as in PSL.

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

Chapter 2

Synthesisable Language Subset

This chapter will explain the synthesisable subset of C# supported by Kiwic.

New is supported in kiwic, provided it is called only from constructors or once and for all on the main threads before they enter an infinite loop.

2.1 FAQ

Q. Why is the reset input not used?

A. The reset net is disconnected unless you indeed add

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

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

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

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

2.1.1 Accessing Simulation Time

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

```
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.

Chapter 3

Orangepath Synthesis Engines

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

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

A synthesis engine rewrites one HPR machine as another.

3.1 A* Live Path Interface Synthesiser

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

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

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

3.2 Transactor Synthesiser

The transactor synthesiser is described on this link

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

3.3 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>

3.4 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.)

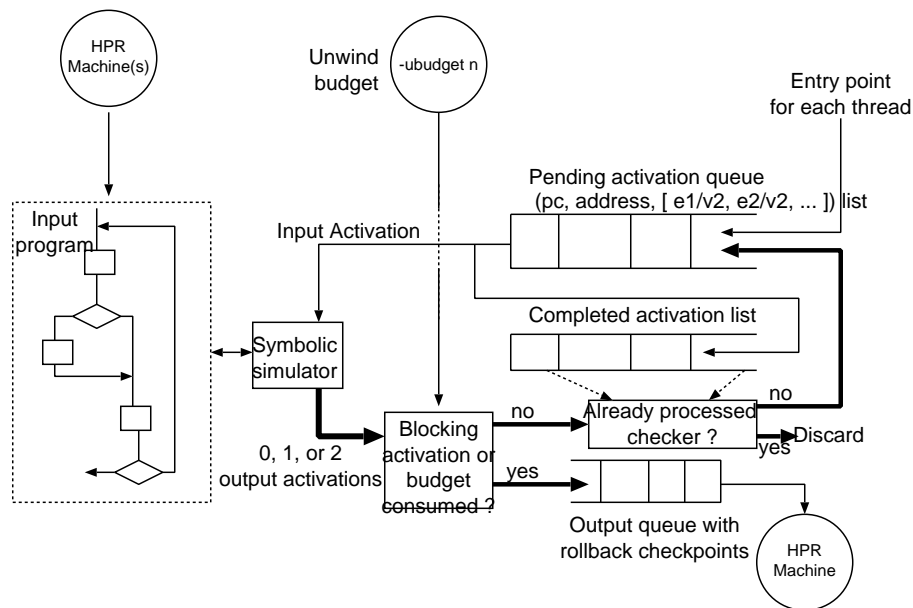


Figure 3.1: The Synchronous FSM generator in the Orangepath tool.

3.5 Bevelab: Synchronous FSM Synthesiser

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

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

Bevelab take an additional input, from the command line, which is an unwind budget: a number of basic blocks to consider in any loop unwind operation. Where loops are nested or fork in flow of control, the budget is divided among 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 vestigial sequencer in RTL output structures. This makes the output code more readable but can make it less compact for synthesis, depending on the capabilities of the FPGA tools to do their own minimisation.

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

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

The string `-resets none` should be passed in to suppress reset logic for FPGA targets. FPGA’s tend to have built-in, dedicated reset wiring.

```
-synthcontrol 'preserve-sequencer;resets:none;sequencer:packed'
```

Bevelab has a number of scheduling algorithms (selectable from recipe of command-

line).

3.5.1 Bevelab: Internal Operation

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

The output is a list of finite-state-machine edges that are finally placed inside a single HPR parallel construct. The edges have to forms (g, v, e) (g, 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.

3.6 PSL Synthesiser

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

3.7 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,

3.8 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.

3.9 Restructure Synthesiser

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

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

Chapter 4

Output Formats

The HPR library contains a number of output code generators. All of these write out a representation of an internal HPR machine. Not all forms of HPR machine can be written out in all output forms, but, where this is not possible, a synthesis engine should be available that can be applied to the internal HPR machine to convert it.

Certain output formats can encode both an RTL/hardware-style and a software/threaded style. For instance, a C-like input file can be rendered out again in threaded C style, or as a list of non-blocking assignments using the SystemC library.

The following output formats may be created:

1. **RTL Form:** The RTL output is written as a Verilog RTL. One module is created that either contains just the RTL portion of the design, or the RTL and instances of each MPU that is executing software parts of the design.
2. **Netlist Form:** The RTL output is compiled to a structural netlist in Verilog that contains nothing but gate and flip-flop instances.
3. **H2 IMP Form:** The HPR form is output to an IMP file. This has the same syntax as the imperative subset of H2.
4. **SMV form:** The HPR VM is output as an SMV code and the assertions that have not been compiled or refined are output as assertions for SMV to check.
5. **C Form:** The HPR VM is output as C code suitable for third-party compilers. RTL forms may also be output as 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 code.

Chapter 5

General Orangepath Facilities

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

5.1 FILES AND DIRECTORIES

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

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

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

5.2 Espresso

Espresso is not currently needed for Fsharp implementation of HPR.

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

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

5.3 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`.

5.4 HPR Command Line Flags

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

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

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

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

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

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

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

Flag `-give-backtrace` prevents interceptions of HPR backtraces and will therefore give a less processed, raw error output from mono.

Flag `-root rootname` specifies the root facet for the current run. A number of items can be listed, separated by semicolons. The ones before the last one are scanned for static and initialisation code whereas the last one is treated as an entry point.

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

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

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

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

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

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

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

See § ??.

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

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

See § ??.

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

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

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

`-rootmodname name`

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

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

The default `ifshare` operation is that guards are tally counted and the most frequently used guard expressions are placed outermost in a nested tree of `if` statements.

The `ifshare` flag turns off `if`-block generation in output code. If set to `'none'` then 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).

The `-restructure` flag controls mechanisms for overcoming static hazards and moving on-chip RAMs to off chip. Currently the argument is the name of the protocol for off-chip RAMs, which may be `BVCI` or `HSIMPLE`.

5.4.1 Other output formats

The `-sysc` flag causes the tool to generate `SystemC` output files.

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

The `-smv` flag causes the tool to generate a `nuSMV` output file.

The `-ucode` flag causes generation of `UIA` microprocessor code for the design.

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

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

`-gatelib NAME` requests that the Verilog output is in gate netlist format. This takes precedence over `-vn1` that causes RTL output.

5.4.2 General Command Line Flags

The `-version` flag give tool version and help string.

The `-help` flag give tool version and help string.

The `-opentrace` flag sets the opentrace level: this alters the debugging output but most debugging is in the `h2log` file anyway.

The `-rwtrace` flag sets the `rwtrace` level, rather like the `-opentrace` option.

5.4.3 Simulation Control Command Line Flags

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

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

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

The `-title title` flag names the `diosim` plot title.

The `-sim-rtl` flag causes `diosim` to simulate the results of the generator processor (e.g. compilation to FSM) rather than the input form.

The `-sim-gates` flag causes `diosim` to simulate the results of compilation to gates (`-gatelib` is used) rather than the input form.

The `-plot plotfile` flag causes plot file output of the `diosim` simulation to a named plot file.

The plot file can be viewed under x-windows and/or converted to a gif using the `diogif` program.

5.5 Diosim Simulator

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

The simulator accepts an hierarchical H2 machine and simulates it.

The simulator will verify all safety assertion rules that contain no temporal logic operators. Other safety and all liveness assertions are ignored.

Non-deterministic choices are made on the basis of a PRBS that the user may seed.

The PRBS is also used for synthetic input generation from plant machines or external

inputs. PRBS values used for external inputs are checked against plant safety assertions and rejected if they would violate.

Output is a log and plot file. The plot file is currently in diogif plot format, but a VCD format should be added.

Detailed logging can be found in the obj/log files. If a program prints the string 'diosim:traceon' or 'diosim:traceoff' the level of logging is changed.

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

Index