October, 2008
Version 1.8

From Cryptol to FPGA: A Tutorial

Cryptol:
The Language of Cryptography

| galois |

Galois, Inc.
421 SW 6th Avenue | Suite 300 | Portland, OR 97204
T 503.626.6616 | F 503.350.0833
www.galois.com

Galois, Inc
421 SW Sixth Avenue, Suite 300
Portland, OR 97204

Table of Contents

List of Figures

List of Tables

Preface

This document is intended to teach the reader how to design hardware circuits in Cryptol and use the Cryptol interpreter to produce efficient, pipelined, high-assurance circuits for an FPGA. This document is not an appropriate resource for learning the Cryptol language itself. To learn about Cryptol, see the documents listed in "Resources" on page x.

The intended audience is Computer Scientists and Mathematicians with some knowledge of functional programming. A basic understanding of hardware is recommended. Some experience with hardware design is recommended, but not required.

This document is organized as follows:

- *Section 1, Introducing Cryptol,* introduces relevant features of the Cryptol language and gives an overview of the steps involved in refining a Cryptol reference specification into a synthesizable FPGA implementation.
- *Section 2, The Cryptol Interpreter,* provides an introduction to the Cryptol interpreter, describes the modes relevant to the design and verification of hardware circuits, and explains how to use some of the basic features of these modes.
- *Section 4, Implementing AES,* specializes the reference specification to several different implementations of 128- and 256-bit AES encryption for an FPGA.
- *Appendix A, The AES Reference Specification* describes the AES algorithm and provides a high-level specification written in Cryptol.
- *Appendix 3, Cryptol & Hardware Design,* describes how to use the Cryptol language for hardware design. Discusses limitations of the language, including features that are not supported by the FPGA compiler, and provides examples of using features in the language to make space/time trade-offs, including how to use Block RAMs, how to reuse circuits over time, and how to pipeline a circuit.
- *Appendix B, SPIR and LLSPIR Graphs* describes the circuit graphs that can be generated in SPIR and LLSPIR modes.
- *Appendix C, Sample Utility Functions*, presents a set of useful functions that can be included in Cryptol code.
- A *Glossary* and *Index* complete the document.

System Requirements

The Cryptol interpreter can generate VHDL from the implementations in this document in Windows, Linux, and Mac OS X. However, to synthesize the VHDL

to a netlist or FPGA implementation, Cryptol must run on Windows or Linux with the synthesis tools installed.

- Typographic Conventions

- Resources

  The following documents are included with Cryptol distributions, and may be referenced throughout this document:

  - Cryptol paper: - Cryptol/Documentation/CryptolPaper.pdf
  - Cryptol reference manual: Cryptol/Documentation/Reference.pdf
  - FPGA tutorial: Cryptol/Documentation/FPGA/Tutorial.pdf
  - FPGA user guide: Cryptol/Documentation/FPGA/UserGuide.htm
  - SuiteB reference specifications: Cryptol/suiteB/AES_Revisited.pdf
  - Rijndael specification: Cryptol/base/Examples/AES.cry

  The following third party tools can be useful for equivalence checking:

  - http://www.eecs.berkeley.edu/~alanmi/abc/abc.htm
  - yices
  - ....

1 Introducing Cryptol

Cryptol is a pure, declarative, functional language. A pure function depends only on its inputs, not I/O or any internal state, and does not produce any side effects. Cryptol functions naturally model combinatorial circuits in hardware, because any combinatorial circuit is just a pure mapping from inputs to outputs. Cryptol supports homogeneous fixed-length sequences, which naturally model bitvectors in hardware description languages, and supports mapping sequences over time in order to model sequential circuits.

This document is intended to teach the reader how to design hardware circuits in Cryptol and use the Cryptol interpreter to produce efficient, pipelined, high-assurance circuits for an FPGA. This document is not an appropriate resource for learning the Cryptol language itself. To learn about Cryptol, see the documents listed in “Resources” on page *x*.

1.1 Language Features

This section briefly mentions some of Cryptol's useful features, including available primitives, features of the type system, and other constructs. A complete description of the language can be found in the documentation provided with the release, and on the www.cryptol.net website.

1.1.1 Function Values and Anonymous Functions

In a functional language, functions have values just like any other expressions. For example, `f where f x = x + 1` is a function that increments its argument. It can be bound to variable, which can be applied:

```
g = f where f x = x + 1;
y = g 10;
```

Or, it can be applied in place:

```
y = (f where f x = x + 1) 10;
```

Cryptol supports *anonymous functions*, also known as *lambda abstractions*, functions that are defined without any name. The `f` function above can be defined as a lambda abstraction: `\x -> x + 1`. We can bind `g` to this function just as we did above when it was called `f`:

```
g = \x -> x + 1;
y = g 10;
```

Or, we can apply the anonymous function in place:

```
y = (\x -> x + 1) 10;
```

In Cryptol, *\arg -> body* is simply syntactic sugar for `f where f` *arg* `=` *body*.

1.1.2 Types and Polymorphism

Cryptol supports two interesting polymorphic terms: `zero` and `undefined`. Both of these have the following type:

```
{a} a
```

This means they can be of *any* type. When `zero` is a sequence of bits, then each element in the sequence is `False`. Otherwise, each element in `zero` is itself `zero`. For example, the following are equivalent:

zero : [4][3]	[[False False False] [False False False] [False False False] [False False False]]	[0b000 0b000 0b000 0b000]

Cryptol supports polymorphism and type-level constraints. The type of a function can be parameterized over other types, and these types can be constrained. For example, consider the following function that gets the 4th element of a sequence:

```
f x = x @ 3;
```

The type of this function can be as generic as:

```
f : {a b} [a]b -> b;
```

However, to ensure that we do not try to index outside the sequence, we should constrain a to be at least 4:

```
f : {a b} (a >= 4) => [a]b -> b;
```

This says that, for any types a and b where a is at least 4, the function takes in a sequence of width a, where each element of the sequence is of type b, and returns a single element of type b. The type variable b could itself be Bit or some sequence of arbitrary length.

Type ascriptions can appear almost anywhere within Cryptol code, not just on a line of their own. This is especially useful to prevent Cryptol from inferring a width that is too small. For example, at the Cryptol interpreter prompt, we can observe the following behavior:

```
Cryptol> 1+1
0x0
```

This is because 1 defaults to a width of 1, the smallest width necessary to represent the value. The type of + is:

```
+ : {a b} ([a]b,[a]b) -> [a]b
```

Therefore, the result is the same as the widths of the inputs, so 0x2 overflows to 0x0. To prevent this, we can ascribe a type to either argument to +, which causes Cryptol to infer the type of the other argument and the result:

```
Cryptol> (1:[4]) + 1
0x2
```

All polymorphic types must be specialized to monomorphic (i.e. specific) types at compile time. Cryptol will infer types as much as possible, including defaulting to the minimum width possible needed to represent a sequence. If it cannot reduce a function to a monomorphic type, it will refuse to apply or compile it.

The way to force an expression to a monomorphic type is to ascribe a type, either directly to the expression or somewhere else that causes the compiler to infer the type of that expression. For example, consider a polymorphic function that increments its argument:

```
inc : {a} (a >= 1) => [a] -> [a];
inc x = x + 1;
```

In the expression inc x, the compiler must be able to infer either the type of x or the type of inc x in order to know the type of this particular instantiation of inc. We can explicitly ascribe either of these types using inc x :: [10] or inc (x :: [10]), or we can place an ascription elsewhere in our code that will cause the compiler to infer one of these types.

Cryptol allows constant terms to be used as types. So, we could define a and the type of f as follows:

```
a = 16;
f : {b} [a]b -> b;
```

Notice that, in this case, the type of f is no longer parameterized over the type variable a.

Cryptol supports a primitive called width that can be used at both the expression level and the type level. At the expression level, width x is the number of elements in x, and it must be known at compile time; if x is a sequence of bits, then width x is the number of bits needed to represent x. At the type level, width can be used in a constraint.

So, we can write polymorphic functions whose behavior depends on the width of the inputs, as in the following example that outputs the least significant bit of xs when xs has more than 10 bits and otherwise outputs the most significant bit.

```
f : {a} [a] -> Bit;
f xs = (w > 10 & xs @ 0) | xs ! 0
   where w = width xs;
```

1.1.3 Type Aliases and Records

Cryptol supports type aliases and records. One can define a new type as an alias to existing types, and it can be parameterized over other types. For example, the following defines a type Complex x as an alias to (x, x).

```
type Complex x = (x, x);
```

We can use the type alias to define a function that multiplies two complex numbers.

```
multC : {n} (Complex [n], Complex [n]) -> Complex [n];
multC ((a, b), (x, y)) = (a*x - b*y, a*y + b*x);
```

A record contains named fields. We can define a record for complex numbers that names the real and imaginary fields:

```
type Complex x = { real : x;
                   imag : x;
                 };
```

Then, the multC function can be defined as follows:

```
multC : {n} (Complex [n], Complex [n]) -> Complex [n];
multC (x, y) = { real = x.real * y.real - x.imag * y.imag;
                 imag = x.real * y.imag + x.imag * y.real;
               };
```

1.1.4 Enumerations

Cryptol supports finite and infinite enumerations. Following are some examples:

This expression	is equivalent to	because...
[1..10]	[1 2 3 4 5 6 7 8 9 10]	the .. notation indicates a monotonic increasing sequence
[1 3..10]	[1 3 5 7 9]	the difference between the first two elements establishes a step size for the monotonic increasing sequence
[1..]	[1 2 3 4 5 6 7 ..]	the lack of upper bound indicates an infinite monotonic increasing sequence

This expression	is equivalent to	because...
[10--1]	[10 9 8 7 6 5 4 3 2 1]	the -- notation indicates a monotonic decreasing sequence
[10 6--0]	[10 6 2]	The difference between the first two elements establishes a step size for the monotonic decreasing sequence.
[10--]	[10 9 8 7 6 ...]	The lack of upper bound indicates an infinite monotonic decreasing sequence.

1.1.5 Index Operators

Cryptol supports the following index operators: @, @@, !, and !!. The @ operator indexes from the least significant element and ! indexes from the most significant element. The least significant element is the rightmost for sequences of bits and the leftmost for all other sequences*. The operators @@ and !! lookup a range of indices. So, the following equalities hold:

This expression	is equivalent to	because...
([0 1 2 3 4 5 6 7] : [8][3]) @@ [1..3]	[1 2 3]	
([0 1 2 3 4 5 6 7] : [8][3]) @@ [3--1]	[3 2 1]	
([0 1 2 3 4 5 6 7] : [8][3]) !! [1..3]	[6 5 4]	
([0 1 2 3 4 5 6 7] : [8][3]) !! [3--1]	[4 5 6]	
(0b00011100 : [8]) @@ [1..3]	0b110	
(0b00011100 : [8]) @@ [3--1]	0b011	
(0b00011100 : [8]) !! [1..3]	0b100	
(0b00011100 : [8]) !! [3--1]	0b001	

All sequence widths are fixed and therefore must be known at compile time. The width of the result of @@ and !! depends on the value of the second argument; therefore, it must be known at compile time.

Sequence Operations and Transformations. Cryptol provides the take and drop primitives:

```
take : {a b c} (fin a,b >= 0) => (a,[a+b]c) -> [a]c
drop : {a b c} (fin a,a >= 0) => (a,[a+b]c) -> [b]c
```

As their names imply, they take or drop a certain number of elements from the beginning of a sequence. Because the width of the result depends on the first argument (number of elements that are taken or dropped), this argument must be constant. For example, we cannot define the following function, because n is not known:

```
f n = take(n, [1 2 3 4 5 6 7 8 9 10]);
```

*In Cryptol 2.0, all sequences are "big-endian", that is, @ will always index from left to right.

Cryptol supports several primitives for splitting and combining sequences:

```
split : {a b c} [a*b]c -> [a][b]c
splitBy : {a b c} (a,[a*b]c) -> [a][b]c
groupBy : {a b c} (b,[a*b]c) -> [a][b]c
join : {a b c} [a][b]c -> [a*b]c
transpose : {a b c} [a][b]c -> [b][a]c
```

The split, splitBy, and groupBy functions each convert a sequence into a 2-dimensional sequence. Their inverse is the join function. split and splitBy behave the same; the splitBy function is provided as an alternative to split because it allows the user to explicitly choose the size of the first dimension, rather than forcing the compiler to infer it.

The following equalities show how splitBy, groupBy, join and transpose behave.

This expression	is equivalent to
splitBy (3, [1 2 3 4 5 6 7 8 9 10 11 12])	[[1 2 3 4] [5 6 7 8] [9 10 11 12]]
groupBy (3, [1 2 3 4 5 6 7 8 9 10 11 12])	[[1 2 3] [4 5 6] [7 8 9] [10 11 12]]
join [[1 2 3 4] [5 6 7 8] [9 10 11 12]]	[1 2 3 4 5 6 7 8 9 10 11 12]
join [[1 2 3] [4 5 6] [7 8 9] [10 11 12]]	[1 2 3 4 5 6 7 8 9 10 11 12]
transpose [[1 2 3 4] [5 6 7 8]]	[[1 5] [2 6] [3 7] [4 8]]
transpose [[1 2 3] [4 5 6] [7 8 9]]	[[1 4 7] [2 5 8] [3 6 9]]
join (split x)	x
join (splitBy (n, x))	x
join (groupBy (n, x))	x
tranpose (transpose x)	x

1.2 Using Cryptol for Hardware Design

This section introduces hardware design in Cryptol. It defines some terminology, explains synchronous design, and describes the differences between combinatorial circuits and sequential circuits and between the step model and the stream model.

Clockrate is the frequency of the clock on the FPGA and is measured in Hertz (Hz). In synchronous circuits, all clocked components of a circuit accept input at the beginning of a clock cycle and produce output by the end of a clock cycle, which may or may not be the same cycle.

Latency or *propagation delay* is the amount of time between when an input is fed to the circuit and when a corresponding output is produced. It can be measured either in integral number of clock cycles or real-valued seconds. In this document, latency refers to a number of clock cycles while propagation delay refers to wall clock time. In Cryptol-generated synchronous circuits, the latency is known statically.

Output rate is how often a circuit can accept input and produce output. It is measured in inverse clock cycles (e.g. one output every 3 cycles), and indicates how long one must wait before feeding another input into the circuit. Any inputs that are fed to a circuit before it is ready are ignored. Throughout this document, unless otherwise stated, we assume the output rate to be one element per each cycle.

Throughput is the amount of information that is output from the circuit per unit of time. It is the clockrate multiplied by the width of the output, divided by the output rate of the circuit, and is thus measured in bits/second (bps).

Circuit size is a measure of the number of logic units on the FPGA. It includes the number of LUTs, slices, flip-flops, and Block RAMs.

A *pipelined* circuit divides a computation up into stages, all of which are performed during one or more clock cycles and in parallel with each other. A pipelined circuit typically has a high latency but an output rate close to or equal to 1 datum per cycle, resulting in a high throughput.

A *combinatorial circuit* is one whose output is a pure function of its input; Cryptol functions model combinatorial circuits very well. Every combinatorial circuit is an associated propagation delay, and Cryptol can report the estimated propagation delay of any combinatorial circuit. Combinatorial circuits are basic building blocks for designing complex circuits.

Combinatorial circuits themselves are not clocked*, but they may be used as building blocks in a clocked circuit. The various combinatorial components of a circuit can be synchronized by placing clocked registers between them, so that each one operates as one stage in a pipeline. See “Pipelining” on page 32.

A *sequential circuit* depends on past input and/or internal state. The state itself is a function of the initial state and past inputs; thus, the difference between a combinatorial circuit and a sequential circuit in Cryptol is that the output of a

*Some pure Cryptol functions, which look like combinatorial circuits, may actually map to clocked, sequential FPGA primitives, such as a Block RAM. In this case, they may have a latency of several clock cycles.

combinatorial circuit may only depend on the most recent input, whereas the output of a sequential circuit may depend on multiple inputs across time.

There are two fundamental ways to model a sequential circuit in Cryptol: the clocked *stream model* and the unclocked *step model*. A circuit may be defined using either model, or some combination of them, though all circuits modeled in the step model must eventually be lifted to the stream model.

1.2.1 Stream Model

In the stream model, sequential circuits are modeled using infinite sequences over *time*, so a function in the stream model has some variation of the following type:

```
[inf]input -> [inf]output;
```

Each element in the input or output corresponds to some number of clock cycles, which is the latency of the circuit. To manage state the user may define a stream within the circuit that holds the state, as in the definition of lift_step below.

One can lift a combinational circuit into the stream model using the following function. It takes in the function for a combinatorial circuit and an infinite stream, and maps the function across all elements in the stream.

```
lift : {a b} (a -> b, [inf]a) -> [inf]b;
lift (f, ins) = [| f x || x <- ins |];
```

Note that this may not cause the circuit to become clocked, especially if there is no stateful information passed from one cycle to the next. It is important that circuits generated by Cryptol always be clocked, otherwise the synthesis tools cannot make use of clock constraints to produce useful timing analyses. In general, it is good to latch onto the inputs and outputs of a circuit by inserting registers after the inputs, before the outputs, or both. The following function lifts a combinatorial function into the stream model *and* places those registers:

```
lift_and_latch : {a b} (a -> b, [inf]a) -> [inf]b;
lift_and_latch (f, ins) = [undefined] # [| f x || x <- [undefined] # ins |];
```

See “Delays and Undelays” on page 24 and “Pipelining” on page 32 to learn how to use registers in Cryptol.

1.2.2 Step Model

In the step model, sequential circuits are modeled as combinatorial circuits that are later lifted into the stream model. A function in the step model is defined as a pure mapping from input and current state to output and next state, so it has some variation of the following type:

(*input*, *state*) -> (*output*, *state*)

The top-level function (the argument to the `:translate` command) must be defined in the stream model. One can easily lift a function from the step model to the stream model. The following function lifts any step function, called f_step in this example, into the stream model:

```
lift_step : {in out state} ((in,state) -> (out,state), state, [inf]in) -> [inf]out;
lift_step (f_step, init_state, inputs) = [| out || (out, state) <- xs |]
    where xs = [(undefined, init_state)] #
                 [| f_step (in, state)
                 || in <- inputs
                 || (out, state) <- xs
                |];
```

Like the lift function, this function applies a combinational function f_step to each element in an infinite stream inputs. However, it also starts with an initial state, init_state, and propagates the state as it folds f_step over the inputs.

In this example, a local stream has been defined that carries both the states and the outputs. The final output is simply the contents of the stream after discarding the states. xs must be prepended with an initial output-state pair because it is defined recursively; it needs an initial value on which to base all further computations. init_state should be chosen according to the step function being used.

2 The Cryptol Interpreter

...need intro

Enter the cryptol interpreter by typing `cryptol` at the shell command prompt. The interpreter provides a typical command-line interface (a read-evaluate-print loop).

One can execute a shell command from within the interpreter by placing a ! character at the beginning of the command:

```
Cryptol> !ls
```

The Cryptol interpreter supports a number of commands, each of which begins with a colon (:). Following are the most common commands used in the Cryptol interpreter. For a more detailed discussion of these and other commands, see the reference manual.

TABLE 2-1 *Frequently useful interpreter commands.*

Command	Description
`:load` *path*	Load all definitions from a `.cry` file, bringing them into scope.
`:b`	Browses (lists) all variables and functions that are currently in scope, along with their types.
`:t` *expr*	Prints the type of an expression, which may include variables and functions that are in scope
`:set` *mode*	Switch to a given mode. Each mode supports a different set of options and performs evaluation on a different intermediate form that is translated from the AST. See "Cryptol Modes for Hardware Design" on page 14 for a discussion of the modes that are useful for hardware design
`:set +v`	Enable higher verbosity. The interpreter will inform the user of each step performed during compilation, including external tools that are called. Use `:set -v` to disable high verbosity
`:set outdir=`*path*	Set the directory in which all generated files will be written, including temporary files and the final file generated by `:translate`.
`:set outfile=`*path*	Set the base name for generated files in the output directory. This should not have an extension; Cryptol inserts the extension automatically. Set this to the empty string (`:set outfile=`) to have Cryptol print the result to the screen instead of a file.
`:translate` *function* [*path*]	Compile a function to the intermediate form associated with the current mode (see "Cryptol Modes for Hardware Design" on page 14). This function is known as the *top-level function*. The optional *path* argument is relative to the current working directory, or can be written as an absolute path, and should include the desired extension. If *path* is not provided, then Cryptol uses the `outfile` setting, saves the file in `outdir`, and adds the extension automatically.
`:fm` *function* [*path*]	Generate a formal model from a function. The path is optional, and will be chosen automatically unless provided by the user. All modes that support the generation of formal models produce the same format, so functions can be checked for equivalence across modes.
`:eq` *function*|*path function*|*path*	Determine whether two functions are equivalent. Requires two arguments, each of which is either a Cryptol expression in parentheses or the path to a formal model in quotes.

The *path* argument to any command above must be surrounded in quotes and may include variables from the interpreter's environment, each preceded by a dollar sign. For example, to translate a function `f` to a file `foo.vhdl` in the current output directory, issue the following command:

```
Cryptol> :translate f "$outdir/foo.vhdl"
```

Any settable option can be reset to its default. For example, to reset the output directory, issue the following command:

```
Cryptol> :set outdir
```

2.1 Cryptol Modes for Hardware Design

This section discusses the Cryptol modes that are relevant to hardware design and verification. Enter a mode by typing

```
Cryptol> :set mode
```

where *mode* is one of the modes described in Table 2-2.

TABLE 2-2 *Cryptol compilation modes.*

mode	description
bit	Performs interpretation on the IR. Useful for prototyping circuits. Supports the entire set of Cryptol.
symbolic	Performs symbolic interpretation on the IR. Useful for prototyping circuits. Supports equivalence checking
LLSPIR	Compiles the IR to Low Level Signal Processing Intermediate Representation (LLSPIR), inlining all function calls into the top-level function and performing timing transformations that optimize the circuit. Provides rough profiling information of the final circuit, including longest path, estimated clockrate, output rate, latency, and size of circuit. Supports equivalence checking. Rather than output LLSPIR, The translate command produces a .dot file, a graph of the LLSPIR circuit that can be viewed graphically. See Appendix B on page 73 for an explanation of SPIR/LLSPIR graphs.
VHDL	Compiles to LLSPIR and then translates to VHDL. Evaluation is performed by using external tools to compile the VHDL to a simulation executable and then running the executable. This is useful for generating VHDL that is manually integrated into another design, rather than directly synthesizing the result.
FPGA	Compiles to LLSPIR, translates to VHDL, and uses external tools to synthesize the VHDL to an architecture dependent netlist. There are several options in this mode that control what the external tools should do next, and they are most easily accessed via the following aliases: ■ FSIM - Compiles the netlist to a low-level structural VHDL netlist suitable for simulation only. Evaluation is performed by compiling the VHDL to a simulation executable and running the executable. Produces profiling information that does not take into account routing delays. Reports the maximum theoretical clockrate ■ TSIM - Similar to FSIM, but performs *map* and *place-and-route* when generating the VHDL netlist. This process can increase compilation time significantly, but produces very accurate profiling, including a true obtainable clockrate. ■ FPGA_Board - Compiles the architecture dependent netlist to a bitstream suitable for loading onto an FPGA board. The user can choose which board to generate a bitstream for via `:set fpga_board=`*board*, where *board* is either `avnet_v4mb` or `spartan3e`.

When an expression is entered at the interpreter prompt, it is translated to the intermediate form associated with the current mode and then evaluated. The translate command produces the intermediate form for an expression in the current mode, but does not evaluate it. Regardless of the mode, all concrete syntax is first translated into an abstract syntax tree called the IR (intermediate representation). For some modes, the IR is their intermediate form, while other modes translate from the IR to their own intermediate form.

The :eq command is supported in FSIM and TSIM modes. It changes the synthesis target to a Verilog netlist suitable for equivalence checking; it compiles this netlist to a formal model using a tool included with Cryptol called symbolic_netlist.

The top-level function is compiled to a single VHDL entity whose interface is determined by the type of the function. The top-level function always has some variation of the following type:

```
[inf]a -> [inf]b;
```

For each of a and b, if the type is a tuple, then each element of that tuple becomes a port in VHDL; otherwise the type becomes a bit or bitvector in VHDL. Tuples nested inside a top-level tuple are appended into single bitvectors. For example, if the type of the top-level function is

```
[inf](Bit, [4],([8],[12])) -> [inf](Bit, [10]);
```

then the VHDL entity will have the following interface:

```
port (input1 : in std_logic;
      input2 : in std_logic_vector(3 downto 0);
      input3 : in std_logic_vector(19 downto 0);
      output1 : out std_logic;
      output2 : out std_logic_vector(9 downto 0);
      restart : in std_logic;
      clk : in std_logic);
```

2.2 FPGA Mode Settings

Following are relevant settings that are used by the FPGA modes. Unless otherwise specified, each setting is available in Cryptol 1.6 and later versions. Each of these settings can be modified using the :set command. For example:

```
Cryptol> :set fpga_clockrate=400
```

And, each setting can be reset to its default value by leaving off the assignment:

```
Cryptol> :set fpga_clockrate
```

TABLE 2-3 *FPGA modes*

setting name	description
fpga_simulation	Sets which simulation tool to use to evaluate VHDL. Valid values are: ■ modelsim ■ ghdl ■ ise This setting is also used in the VHDL mode.
fpga_synthesis	Sets which tool to use for synthesizing VHDL to a netlist. Valid values are: ■ xst ■ synplify_pro
fpga_clockrate	Sets the *requested* clockrate. Synthesis and place-and-route tools will try to satisfy this constraint, so setting it higher can yield a higher clockrate.
fpga_optlevel	Sets the optimization effort level used by synthesis tools. Can be any number between 1 and 5, where 1 is the default and the least effort. Note that high values can cause synthesis to take a *very* long time. This does not affect the Cryptol compiler and the VHDL that it generates.
fpga_part	Sets the FPGA part that is used by the synthesis tools and that the Cryptol compiler uses to estimate profiling information. Example values include xc4vlx60-12ff668 and xc4vfx140-10ff1517
fpga_part_stepping	Sets the stepping version associated with the FPGA part. This is usually between 1 and 4, inclusive
cgdebug	Enables very verbose debugging messages, normally intended for developers. Can be used to obtain profiling information in LLSPIR mode if set to c, but the profile is very inaccurate. In recent versions of Cryptol, it is replaced with the fpga_profiling setting and more accurate profiling.
fpga_profiling	Indicates what profiling information should be collected and displayed in LLSPIR and FPGA modes. This is a comma separated list of any of the following values: ■ space: for area profiling ■ timing: for estimated clockrate ■ path: for a longest path analysis This setting is not available in Cryptol 1.6.
fpga_board	When targeting an FPGA board, sets the type of board.

setting name	description
fpga_blockram	Sets what kind of Block RAM to use for certain lookup tables. Possible values are: ■ none: use only primitive LUTs ■ auto: choose part based on fpga_part setting. Default. ■ behavioral: ■ virtex2: ■ virtex4: See “Block RAMs” on page 25 for more information.

2.3 Performance Analysis

The LLSPIR and FPGA modes can report circuit latency, clockrate, space utilization, and the longest path in a circuit. These results can guide the user towards a more efficient (faster and/or smaller) implementation.

In LLSPIR mode, clockrate and space utilization are very rough estimates; LLSPIR usually underestimates the clockrate. FSIM mode reports space utilization accurately, but the reported clockrate is the theoretical maximum, and is therefore overestimated. TSIM mode reports the exact obtainable clockrate for a particular place-and-route attempt. However, the fpga_clockrate and fpga_optlevel settings can significantly influence the place-and-route tool, so the user should experiment with different values of fpga_clockrate and fpga_optlevel to obtain the maximum possible clockrate for a given circuit.

The user is encouraged to refine an implementation as much as possible in LLSPIR mode before beginning synthesis in an FPGA mode. Translating to LLSPIR takes significantly less time than synthesis, yet still provides enough information to help the user produce an efficient, pipelined implementation. Furthermore, LLSPIR is very similar to VHDL, hence the translation from LLSPIR to VHDL is trivial and it is very likely that an implementation that is verified correct in the LLSPIR mode would also be correct in the VHDL mode.

To obtain the circuit latency, longest path, estimated clockrate, and estimated circuit size in the LLSPIR mode, set the following:

```
Cryptol> :set LLSPIR
Cryptol> :set +v
Cryptol> :set cgdebug=c
```

Then use the `:translate` command to compile a function to LLSPIR, producing a `.dot` file and printing the performance information*.

```
Cryptol> :translate function [file]
```

Note that Cryptol expects the top-level function to be defined in the stream model, and will forcibly lift it into the stream model if necessary. If the top-level function is a combinatorial circuit on a sequence, the compiler will force this sequence to be mapped over time. Compiling a combinatorial circuit this way may yield timing results that are misleading or difficult to interpret. To compile a combinatorial function to LLSPIR or VHDL, one should always lift the function into the stream model using the lift function defined in Section 1.2, "Using Cryptol for Hardware Design," on page 7:

```
Cryptol> :translate (\x -> lift(f, x))
```

*.dot file generation is available in recent version of Cryptol, but not version 1.6

2.4 Equivalence Checking

The following modes support equivalence checking: symbolic, LLSPIR, and FPGA (FSIM and TSIM). The user can generate a formal model of a function in any of these modes using the :fm command, and check a function for equivalence to a formal model using :eq. Two formal models generated in two different modes may also be compared using :eq.

There are two main uses of equivalence checking:

- to verify that an implementation is correct with respect to a specification
- to verify that Cryptol compiles a particular function correctly, by comparing a function in LLSPIR or FPGA mode to the same function symbolic mode.

Cryptol supports three specific equivalence checkers: jaig, eaig, and abc. jaig and eaig are shipped with Cryptol, while abc is an open source equivalence checker available at http://www.eecs.berkeley.edu/~alanmi/abc/abc.htm.

TABLE 2-4 *Options for equivalence checking:*

option	description
:set eq_checker=*path*	Set the path to the equivalence checker executable
:set eq_checker_options=*opts*	Set optional command line arguments to the equivalence checker.
:set eq_api=*api*	Tells Cryptol how to communicate with the equivalence checker. Possible values include: ■ jaig: use the jaig/eaig API ■ abc: use the abc API
:set eq_format=*fmt*	Sets the format of the formal models that Cryptol generates. ■ jaig: use the jaig/eaig format ■ abc: use the abc format
:set abc	Shorthand for the following option settings: :set eq_api=abc :set eq_format=abc
:set jaig	
:set eaig	

To verify that an implementation is equivalent to a reference specification, compare the implementation and the specification in symbolic mode:

```
Cryptol> :set symbolic :eq f f_spec
```

To verify that Cryptol compiles the implementation correctly, compare the formal model of the function in symbolic mode to the formal models of the same function in LLSPIR and FPGA modes:

```
Cryptol> :set symbolic
Cryptol> :fm f "f.fm"

Cryptol> :set LLSPIR
Cryptol> :eq f "f.fm"

Cryptol> :set FSIM
Cryptol> :eq f "f.fm"
```

```
Cryptol> :set TSIM
Cryptol> :eq f "f.fm"
```

The equivalence checker outputs `True` if two functions are equivalent; if not, it outputs `False` along with a counterexample.

3 Cryptol & Hardware Design

This section discusses the Cryptol language with respect to hardware design. First, it discusses features in the language that are not supported by the compiler, and other issues that may make it difficult to generate efficient circuits. It also describes techniques for making space-time trade-offs and applies these techniques to several concrete and simple examples. These techniques are the same used to manipulate the AES implementations in Section 4, "*Implementing AES*," on page 41.

3.1 Issues and Limitations

This section discusses some of the limitations of the FPGA compiler, including some techniques for avoiding them.

3.1.1 The FGPA backend supports a subset of the Cryptol language.

The FPGA compiler only supports division by powers of 2. This is a limitation of the underlying hardware.

The FPGA compiler does not support primitive recursion. Most recursive functions can be easily rewritten using value recursion (i.e. stream recursion), which the compiler supports. For example, consider a recursive implementation of factorial:

```
fact n = if n == 0 then 1 else n * fact(n-1);
```

We can re-implement this using a value that is defined recursively:

```
fact2 n = facts @ n
    where facts = [1] #  [| i * prev
                         || i <- [1..]
                         || prev <- facts
                                          |];
```

The FPGA compiler only partially supports higher-order functions (functions as first-class values). Specifically, it does not allow functions to return functions, but it does allow functions to take another function as input as long as that function can be inlined away at compile time. For example, consider the following function that takes in a function and applies it to each element of a tuple:

```
app_tup : {a b} (a -> b, (a, a)) -> (b, b);
app_tup (f, (x, y)) = (f x, f y);
```

This function itself cannot be compiled to hardware. However, we can apply app_tup to a known function as follows:

```
inc_tup : {a} (a >= 1) => ([a],[a]) -> ([a],[a])
inc_tup t = app_tup (inc, t) where inc x = x + 1;
```

Then, the compiler inlines inc into app_tup to obtain code that no longer contains higher-order functions:

```
inc_tup t = (inc x, inc y) where inc x = x + 1;
```

However, the FPGA compiler does not support functions that return non-closed functions, such as the following:

```
f : [8] -> ([8] -> [8]);
f x = g where g y = x + y;
```

In this definition, g is not closed, since it relies on the variable x.

3.1.2 Inefficient Sequence Comprehensions

Hardware performance can vary drastically based on subtle changes in how sequence comprehensions are written. This section explains how to generate efficient circuits from sequence comprehensions.

Although the following two expressions are semantically equivalent, they compile to significantly different circuits:

This expression...	is equivalent to...
take(N, [1..])	[1..N]

The first one generates code to calculate a sequence of numbers at *runtime*. Furthermore, because the sequence is infinite, it is mapped across *time*, so each element is calculated in a different clock cycle. The second expression generates the enumeration at *compile time.* When used in larger circuits, this subtle difference can cause drastic changes in performance.

Consider a function that takes in some fixed number of bytes, N, and pair-wise multiplies each byte by 1 through N, respectively. A naive implementation might look like this:

```
prods : {N} [N][8] -> [N][8];
prods xs =  [| x * i
            || i <- [1..]
            || x <- xs
                  |];
```

However, the infinite sequence [1..] will force the sequence comprehension to take N cycles to complete. It is possible that the user wants to reuse a single multiplier over N cycles, but that is not what happens. To correctly lay this sequence out over multiple clock cycles and reuse a single multiplier, the user should use the seq pragma (see Section 3.4, "Space-time Trade-offs via the par and seq Pragmas," on page 26).

The correct way of implementing this function is to force the sequence that i draws from to be finite:

```
prods_prime xs =    [| x * i
                    || i <- [1..(width xs)]
                    || x <- xs
                                |];
```

This will instantiate width xs multipliers (actually one less, since the multiplication by 1 is optimized away) in parallel. A single multiplier can be reused over multiple clock cycles by using the seq pragma (see Section 3.4, "Space-time Trade-offs via the par and seq Pragmas," on page 26).

3.2 Delays and Undelays

Consider a sequence s, in the stream model, so s is infinite and mapped over time. Assuming an output rate of one element per cycle, we can delay the stream by n cycles by appending n elements to the beginning of the stream. For example, the following function outputs its inputs unmodified, but each output is delayed by one cycle:

```
delay : {a} [inf]a -> [inf]a;
delay ins = [undefined] # ins;
```

And this function delays the output by two cycles:

```
delay2 : {a} [inf]a -> [inf]a;
delay2 ins = [undefined undefined] # ins;
```

Alternatively, we could define delay2 by applying delay to the input twice:

```
delay (delay ins)
```

Note that using zero instead of undefined adds latency to the circuit because it takes some time to initialize it.

While a *delay* causes its output to occur some number of cycles after the input, an *undelay* causes its output to occur *before* the input. One can cause an undelay to occur using the drop construct:

```
undelay : {a} [inf]a -> [inf]a;
undelay ins = drop(1, ins);
```

Undelays are not synthesizable. During an optimization pass, the compiler pushes delays and undelays around in the circuit, introducing new ones and canceling delays with adjacent undelays.

Delays and undelays can be used to synchronize data across time. For example, the following Fibonacci implementation uses drop to look back in time in the stream so that we can add the previous two values together:

```
fib : [inf][32];
fib = [ 1 1 ] #     [| x + y
                    || x <- fib
                    || y <- drop(1, fib)
                                |];
```

One can also use delays to produce pipelines. A delay synthesizes to a *register* (also known as *latch* or the FPGA primitive called *flip-flop*). Section 3.5, "Pipelining," on page 32 shows how to use registers to pipeline circuits.

3.3 Block RAMs

The FPGA compiler automatically maps certain lookup tables to FPGA Block RAMs. If the fpga_blockram setting is not none, then any lookup table that is statically known and contains at least 2^7 =128 entries will be implemented as a Block RAM with one or two clock cycles of latency. The type of Block RAMs used depends on fpga_blockram setting:

setting	description
behavioural	Generate behavioural VHDL for Block RAMs that add 1 cycle of latency. The Xilinx synthesis tools recognize the behavioural definition and instantiate Block RAMs.
virtex2	Instantiate Virtex 2 Block RAMs, which add 1 cycle of latency.
virtex4	Instantiate Virtex 4 Block RAMs, which add 2 cycles of latency but can be clocked higher than Virtex 2 Block RAMs.
auto	Allow the compiler to choose based on fpga_part. This is the default option.

For example, the first 128 values of the sequence fib from Section 3.2, "Delays and Undelays," on page 24 can be implemented using a static lookup table that can be mapped to a Block RAM:

```
fib_table : [128][32];
fib_table =  [
    0x000000010x000000010x000000020x000000030x000000050x00000008
    0x0000000d0x000000150x000000220x000000370x000000590x00000090
    0x000000e90x000001790x000002620x000003db0x0000063d0x00000a18
    0x000010550x00001a6d0x00002ac20x0000452f0x00006ff10x0000b520
    0x000125110x0001da310x0002ff420x0004d9730x0007d8b50x000cb228
    0x00148add0x00213d050x0035c7e20x005704e70x008cccc90x00e3d1b0
    0x01709e790x025470290x03c50ea20x06197ecb0x09de8d6d0x0ff80c38
    0x19d699a50x29cea5dd0x43a53f820x6d73e55f0xb11924e10x1e8d0a40
    0xcfa62f210xee3339610xbdd968820xac0ca1e30x69e60a650x15f2ac48
    0x7fd8b6ad0x95cb62f50x15a419a20xab6f7c970xc11396390x6c8312d0
    0x2d96a9090x9a19bbd90xc7b064e20x61ca20bb0x297a859d0x8b44a658
    0xb4bf2bf50x4003d24d0xf4c2fe420x34c6d08f0x2989ced10x5e509f60
    0x87da6e310xe62b0d910x6e057bc20x543089530xc23605150x16668e68
    0xd89c937d0xef0321e50xc79fb5620xb6a2d7470x7e428ca90x34e563f0
    0xb327f0990xe80d54890x9b3545220x834299ab0x1e77decd0xa1ba7878
    0xc03257450x61eccfbd0x221f27020x840bf6bf0xa62b1dc10x2a371480
    0xd06232410xfa9946c10xcafb79020xc594bfc30x909038c50x5624f888
    0xe6b5314d0x3cda29d50x238f5b220x606984f70x83f8e0190xe4626510
    0x685b45290x4cbdaa390xb518ef620x01d6999b0xb6ef88fd0xb8c62298
    0x6fb5ab950x287bce2d0x983179c20xc0ad47ef0x58dec1b10x198c09a0
    0x726acb510x8bf6d4f10xfe61a0420x8a5875330x88ba15750x13128aa8
    0x9bcca01d0xaedf2ac5
];
```

3.4 Space-time Trade-offs via the *par* and *seq* Pragmas

Cryptol supports two simple but very powerful pragmas that control space-time trade-offs in the compiler. The par pragma causes circuitry to be replicated, whereas the seq pragma causes circuitry to be reused over multiple clock cycles. By default the compiler replicates circuitry as much as possible in exchange for performance, and the user overrides this behavior using seq; the par pragma is only useful for switching back to the default behavior within an instance of seq.

Semantically, both seq and par are the identity, because the types and semantics of Cryptol have no notion of time:

```
seq : {n t} [n]t -> [n]t;
seq x = x;

par : {n t} [n]t -> [n]t;
par x = x;
```

To understand the basic behavior of the seq pragma, consider some combinatorial circuit, f, which is implemented as a sequence comprehension and has the following type:

```
f : [n]a -> [n]a;
```

By default, the compiler will unroll and parallelize the sequence comprehension as much as possible. However, seq (f xs) requests that the circuitry within f be reused over n cycles. This requires extra flip-flops to synchronize each reuse of the circuitry within the comprehension, but can reduce overall logic utilization, resulting in a smaller circuit.

For example, the prods_prime function from "Inefficient Sequence Comprehensions" on page 22 can be defined to reuse one multiplier over multiple clock cycles:

```
prods_prime xs = seq    [| x * i
                         || i <- [1..(width xs)]
                         || x <- xs
                                                |];
```

Sequence comprehensions can be performed either in sequence or in parallel. If the sequence is infinite or if each element depends on the previous element, then it must be implemented sequentially. Otherwise, it may be implemented in parallel. This section provides two examples, one involving a sequence comprehension that can be performed in parallel, and one involving a sequence comprehension that must be performed sequentially. In both examples, the seq pragma is used to map the sequence over multiple clock cycles, and the performance advantages and disadvantages of doing so are discussed.

For both examples we assume there is some function f of type a -> b or b -> b. The advantages of using seq are greater when f consumes more area, because then reusing f will have a greater impact on logic utilization.

3.4.1 Example 1: Parallel Sequence Comprehension

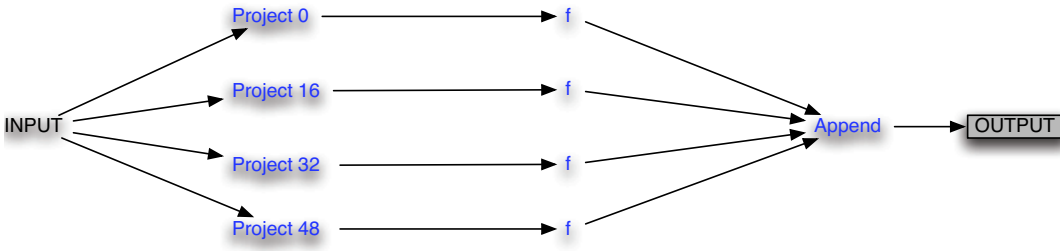
Consider the map function that applies some function to every element in a finite sequence:

```
map : {n a b} (fin n) => (a -> b, [n]a) -> [n]b;
map (f, xs) = [| f x || x <- xs |];
```

By default, f will be instantiated n times and applied in parallel to each of the n inputs. A definition and graph of this function are provided in Figure 3-1, in which n is set to 4. Once lifted into the stream model, this circuit will accept n elements of type a every cycle and output n elements of type b every cycle. To translate this in LLSPIR mode and view profiling information, we lift the function into the stream model:

```
Cryptol> :translate (\x -> lift(map_f1, x))
```

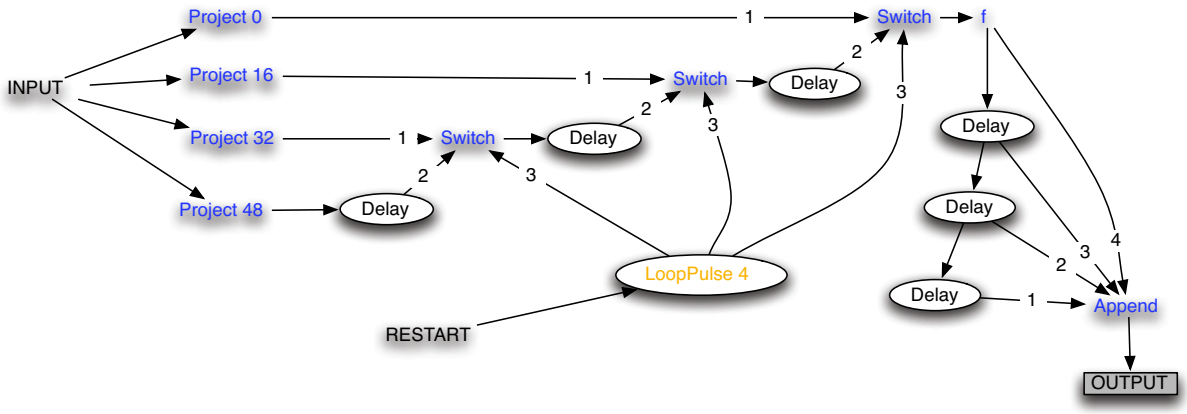
FIGURE 3-1 *Map f in parallel.*




```
map_f1 : [4]a -> [4]b;
map_f1 xs = map(f, xs);
```

This is ideal if f is a relatively small circuit. However, if duplicating f violates a size constraint on the FPGA, then we should trade time for space by instantiating f only once and applying it in sequence using the seq pragma. In this case the output rate is every four cycles. The definition and graph of this function are provided in Figure 3-2 on page 28. There is extra logic needed to facilitate the sequencing of f over multiple cycles, since the input comes in all at once and the output must be produced all at once. Clearly, defining the circuit this way can be beneficial only if f is large.

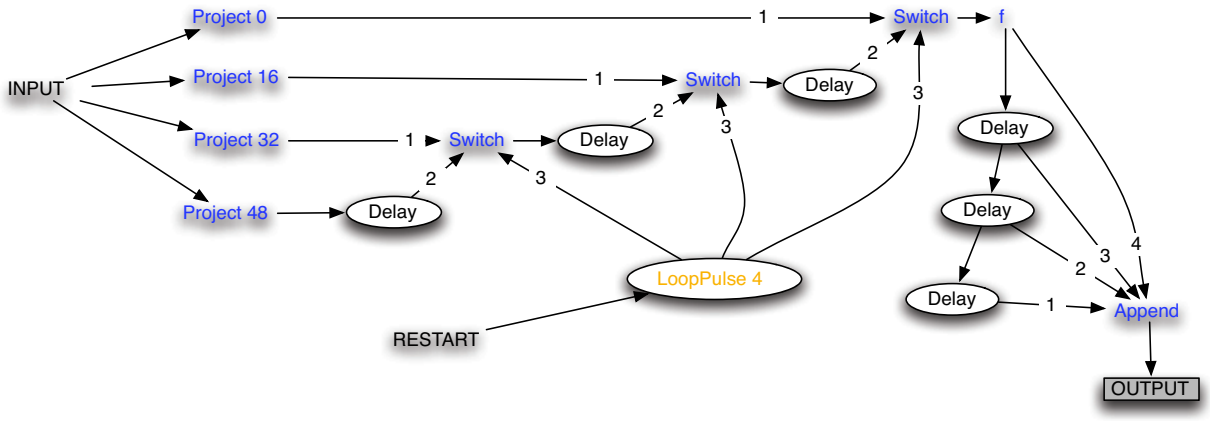
FIGURE 3-2 *Map f in sequence.*




```
map_f2 : [4]a -> [4]b;
map_f2 xs = seq(map(f, xs));
```

We can also define this function using a combination of parallel and sequential computations. For example, we may want to instantiate f exactly two times. Since n is 4, this would cause each instantiation of f to operate on two sequential inputs to obtain an output rate of every two cycles. Such a circuit would provide a balance between space utilization and speed. Figure 3-3 provides the definition and graph of a function that implements this functionality.

FIGURE 3-3 *Map f, two in sequence and two in parallel*




```
map_f3 : [4]a -> [4]b;
map_f3 xs = join [xs0' xs1']
    where {
        xs0, xs1 : [2]a;
        [xs0 xs1] = split xs;
        xs0' = seq(map(f, xs0));
        xs1' = seq(map(f, xs1));
    };
```

Note that the number of elements in each input (currently 4) can be increased without changing the algorithm for map_f3. This function will always generate two instantiations of f.

3.4.2 Example 2: Sequential Sequence Comprehension

Suppose we want to apply the function f : b -> b an arbitrary number of times to an input, where this number of times is k. We can construct the following sequence, where the element at index k-1 is the one that results from k applications of f:

```
[ f(x) f(f(x)) f(f(f(x))) ... ]
```

The following function constructs this sequence:

```
iterate : b -> [k]b;
iterate x = outs
 where outs = [(f x)] #     [| f prev
                            || i <- [1..(k-1)]
                            || prev <- outs
                                                |];
```

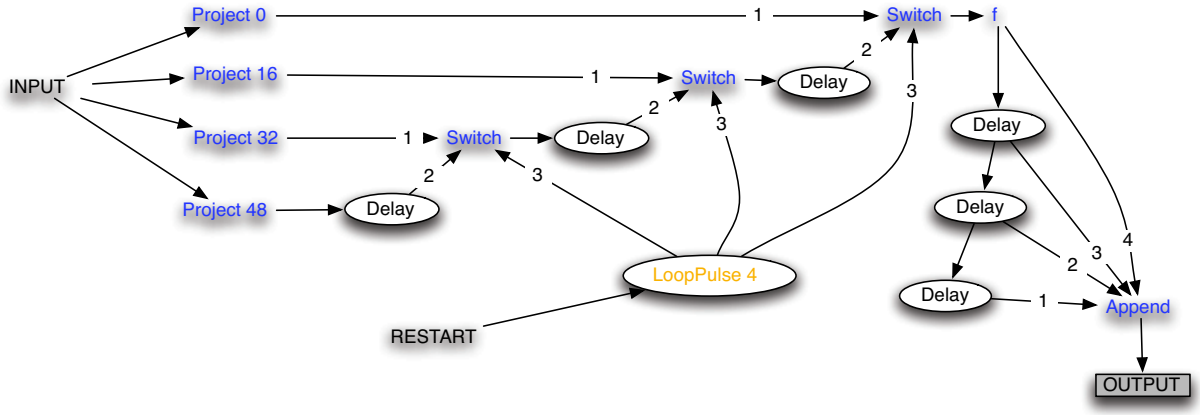
However, f appears twice in this function. We want f to only appear once (inside the comprehension), so that when we use the seq pragma f is only instantiated once. So, we define iterate as follows, where the element at index k is the one that results from k applications of f:

```
iterate : b -> [(k+1)]b;
iterate x = outs
 where outs = [x] #   [| f prev
                      || i <- [1..k]
                      || prev <- outs
                                    |];
```

Note that each element depends on the previous, so the sequence will be evaluated sequentially. Rather than instantiating f in parallel, as in the previous section, the sequence comprehension will be unrolled.

We define two hardware implementations; in the graphs of these implementations below, k is fixed at 4. The first implementation simply returns the last element in the sequence produced by iterate. Its definition and graph are provided in Figure 3-4. It uses k copies of f and chains them together sequentially in one clock cycle.

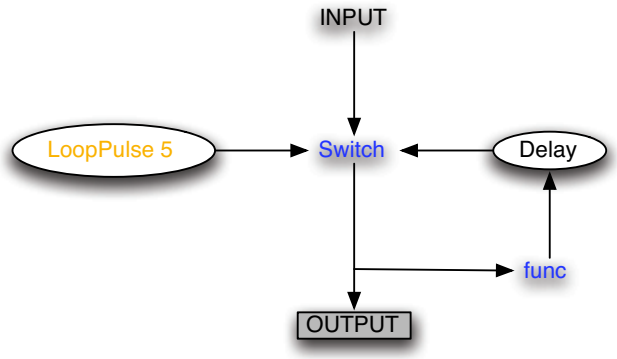
FIGURE 3-4 *iterate (unrolled)*




```
iterate1 : b -> b;
iterate1 x = iterate x ! 0;
```

iterate2 uses the seq pragma to request that f be reused each of k clock cycles. Its definition and graph are provided in Figure 3-5.

FIGURE 3-5 *iterate (reused over time)*




```
iterate2 : b -> b;
iterate2 x = seq (iterate x) ! 0;
```

The result is that iterate1 will have a lower clockrate and will take up more area, but will have an input/output rate of one element per clock cycle. iterate2 will have a higher clockrate and will take up less area, but it will have an input/output rate of one element every k cycles and will require extra flip-flops to latch onto the output of f each cycle.

We can verify that the two implementations are equivalent, using iterate1 as the reference specification:

```
Cryptol> :set symbolic
Cryptol> :fm iterate1 "iterate1.fm"
Cryptol> :set LLSPIR
Cryptol> :eq iterate1 "iterate1.fm"
Cryptol> :eq iterate2 "iterate1.fm"
```

3.5 Pipelining

Sequential circuits in the stream model can be pipelined to increase clockrate and throughput. One separates a function into several smaller computational units, each of which is a stage in the pipeline that consumes output from the previous stage and produces output for the next stage. The stages are synchronized by placing registers between them.

Pipelining an implementation typically increases the overall latency and area of a circuit, but can increase the clockrate and total throughput dramatically. Each stage is a relatively small circuit with some propagation delay. The clockrate is limited by the stage in the pipeline with the highest propagation delay, whereas the un-pipelined implementation would be limited by the sum of the propagation delays of all stages. So, rather than perform one large computation on one input during a very long clock cycle, an n-stage pipeline performs n parallel computations on n partial results, each corresponding to a different input to the pipeline. Sequences provide an appropriate abstraction for representing pipeline stages. Each stage can be represented as one or more parallel infinite streams across time.

Recall from Section 3.2 that one can insert *delays* and *undelays* by adding and removing elements from an infinite stream. We can also use delays to explicitly insert registers in a pipeline and undelays to drop the initially undefined output from a pipeline. A delay in Cryptol maps directly to a *register* (also known as *flip-flop* or *latch*) in hardware.

This section introduces pipelining in Cryptol by defining and comparing many different pipelined implementations of two generic reference specifications. The first specification is a simple combinatorial circuit that takes two inputs, applies two arbitrary arithmetic functions to the respective inputs, and sums the result. The second specification uses stream comprehension to apply a function to the input an arbitrary number of times. Finally, a new reg pragma is introduced which can be used to produce pipelined designs more easily and that more closely resemble their high-level specifications. We show how each of the examples in the following sections can be rewritten much more simply by using the reg pragma.

3.5.1 Example 1: Combinatorial Circuit

Consider the following combinatorial circuit, where g and h are arbitrary combinatorial circuits of type a -> a. This will be our reference specification.

```
add_g_h_spec : ([c],[c]) -> [c];
add_g_h_spec (a, b) = g(a) + h(b);
```

The definitions of g and h are not revealed here, because they are irrelevant to implementing the pipeline unless we want to split g and h themselves into multiple stages (see add_g_h_4 below). The functions g and h can be polymorphic, but for simplicity we fix the width:

```
c = 8;
```

Below are five separate implementations of add_g_h in the stream model. The first simply lifts the original specification into the stream model. The remaining are pipelined implementations. These functions all share the same type:

```
add_g_h_1, add_g_h_2, add_g_h_3, add_g_h_4 : [inf]([c],[c]) -> [inf][c];
```

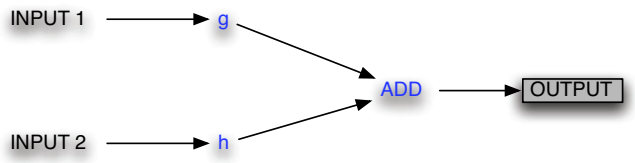
Clockrate, latency, circuit size, and graphs of each implementation can be obtained by entering the following commands in the Cryptol interpreter:

```
Cryptol> :set LLSPIR
Cryptol> :set cgdebug=c :set +v
Cryptol> :translate add_g_h_1 "add_g_h_1.dot"
Cryptol> :translate add_g_h_2 "add_g_h_2.dot"
Cryptol> :translate add_g_h_3 "add_g_h_3.dot"
Cryptol> :translate add_g_h_4 "add_g_h_4.dot"
```

The definitions and graphs of each implementation are provided in the figures below.

First, we implement the specification in the stream model as a circuit that consumes input and produces output on every clock cycle. In this case, g(a) and h(b) are performed in parallel, but the addition operation cannot be performed until g(a) and h(b) both finish. Thus, the total propagation delay is the maximum delay of g and h, plus the delay of the addition. See Figure 3-6 for the definition and graph of this circuit.

FIGURE 3-6 *add_g_h_1*




```
add_g_h_1 ins = [| g(a) + h(b) || (a,b) <- ins |];
```

We can pipeline this circuit by identifying two distinct stages that can execute in parallel:

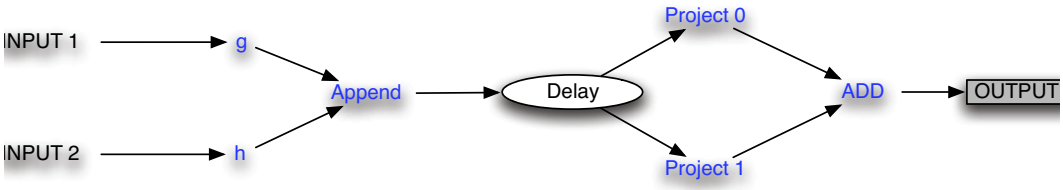
- the applications of g and h
- the addition operation

To implement the pipeline, we evaluate g(a) and h(b) in parallel and store the results in a state in one cycle. On the next cycle, we add the two elements of the state together and make that the output of the circuit. This adds an extra clock cycle of latency so that it now takes two cycles to perform the entire computation. However, the clockrate is only limited by maximum delay of g, h, and the addition, whereas in the previous implementation it was the maximum delay of g and h *plus* the delay of the addition. Therefore, each stage of the computation can execute faster, and the throughput increases.

Note that during any given clock cycle, a pipelined implementation operates on data associated with two consecutive and unrelated inputs; it applies h and g to the most recent inputs, and it applies addition to the state which stores h(a) and h(b) associated with the inputs of the previous cycle.

Our first pipelined implementation, add_g_h_2, is provided in Figure 3-7. It defines a stream for each stage, one to calculate (g(a), h(b)) and the other to calculate the addition, and then place a register between the two stages.

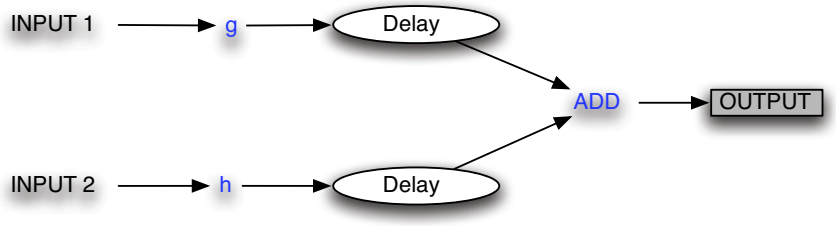
FIGURE 3-7 *add_g_h_2 (pipelining at the stream level)*




```
add_g_h_2 ins = stage2
    where {
        stage1 = [undefined] # [| (g(a), h(b)) || (a, b) <- ins |];
        stage2 = [| g_a + h_b || (g_a, h_b) <- stage1 |];
    };
```

Because we combine g(a) and h(b) into a tuple, these two values will be appended into a single bitvector and stored in a register in VHDL. Because of this, in LLSPIR mode Cryptol reports fan-in and fan-out delays associated with injecting into and projecting out of a tuple. These estimates are probably incorrect, as the synthesis tools should not have a problem routing this. However, to eliminate the possibility of these delays, we can define a separate stream for each element of the tuple, so that we have one stream for g(a), one stream for h(b), and one stream for the addition. See Figure 3-8 for the implementation of add_g_h_3.

FIGURE 3-8 *add_g_h_3 (pipelining at the stream level)*




```
add_g_h_3 ins = stage2
    where {
        stage1_as = [undefined] # [| g(a) || (a, b) <- ins |];
        stage1_bs = [undefined] # [| h(b) || (a, b) <- ins |];
        stage2 = [| g_a + h_b || g_a <- stage1_as || h_b <- stage1_bs |];
    };
```

In each pipelined implementation above, the second stage only performs a single addition operation. Therefore, if either g or h has a propagation delay of more than one addition operation, then the first stage is the bottleneck of the pipeline, and should be split into multiple stages if possible. Suppose g and h are defined as follows:

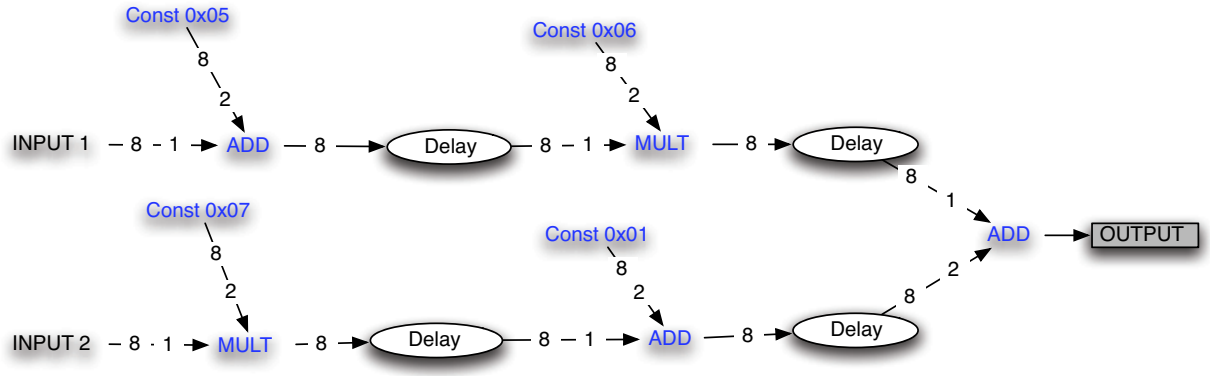
```
g a = (a+5) * 6;
h b = b*7 + 1;
```

We can implement g and h each as a two-stage pipeline, so our entire circuit has three stages:

- perform |a+5| and |b*7|
- apply multiplication by 6 and addition by 1 to the results from stage 1
- add the results from stage 2

This three-stage pipeline, add_g_h_4, is defined in Figure 3-9.

FIGURE 3-9 *add_g_h_4 (three-stage pipeline)*




```
add_g_h_4 ins = stage3
    where {
        stage1_as = [undefined] # [| a + 5 || (a, b) <- ins |];
        stage1_bs = [undefined] # [| b * 7 || (a, b) <- ins |];
        stage2_as = [undefined] # [| a * 6 || a <- stage1_as |];
        stage2_bs = [undefined] # [| b + 1 || b <- stage1_bs |];
        stage3 = [| a + b || a <- stage2_as || b <- stage2_bs |];
    };
```

We can define a utility function that lifts any combinatorial circuit into a stage of a pipeline in the stream model:

```
stage : {a b} (a -> b, [inf]a) -> [inf]b;
stage (f, ins) = [undefined] # [| f x || x <- ins |];
```

Equivalence checking is not possible on infinite streams, so we cannot verify that a sequential circuit is equivalent to its specification for all time. However, we can still provide some level of assurance that the circuit is correct. First, we verify that the function always produces the correct output for the first input. For example, to test add_g_h_1 above, use a function like the following:

```
test_add_g_h : ([c],[c]) -> [c];
test_add_g_h input = add_g_h_1 ([input] # zero) @ 0;
```

This function should be equivalent to the spec, which we can verify with the following command:

```
Cryptol> :eq add_g_h_spec test_add_g_h
```

One may substitute add_g_h_1 with any pipelined implementation above and change the index operand to account for circuit latency. For example, to test add_g_h_2, which has an extra cycle of latency, change 0 to 1.

Next, we can verify that for some fixed number of inputs, the circuit generates the correct outputs. The following function asks this question, returning True if any n

inputs can be found for which the implementation is not equivalent to the specification. The value of n can be adjusted as desired.

```
check_add_g_h inputs = reference != result
    where {
        reference = [| add_g_h_spec x || x <- inputs |];
        result = take (n, add_g_h_1 (inputs # zero));
        n = 1000;
    };
```

The sat solver determines whether any inputs satisfy the query. The circuit is correct if this function is *not* satisfiable.

```
Cryptol> :sat check_add_g_h
```

Again, one may substitute add_g_h_1 with any pipelined implementation above, but account for latency by dropping the initial undefined outputs from result using the drop construct.

3.5.2 Example 2: Sequentially Iterating a Function

In Section 3.4, we defined iterate, a function that sequentially applies some function a certain number of times to an input. In this section we provide a pipelined implementation of iterate.

The reference specification and stream implementation for this function are defined using the iterate function from Section 3.4 on page 26:

```
iterate_spec : b -> b;
iterate_spec x = iterate x ! 0;

iterate_stream : [inf]b -> [inf]b;
iterate_stream ins = [| iterate x ! 0 || x <- ins |];
```

Without pipelining, we must evaluate k applications of f in sequence every clock cycle. This does not scale well to large values of k, since the clockrate drops linearly with respect to k. If we pipeline the sequence to perform an application of f at each cycle, then the value of k does not affect the clockrate and throughput at all; it only increases the number of pipeline stages, and thus the total latency between each input and the output increases too.

First, we can define k infinite streams, where each stream maps f onto the previous stream. We can either define each one of these streams manually, or we can define a sequence comprehension that statically constructs these k streams; that is, the compiler will unroll the comprehension at compile time. We convert the sequence of streams from [k][inf] to [inf][k] using transpose.

```
iterate_stream2 : [inf]b -> [inf]b;
iterate_stream2 ins = [| x ! 0 || x <- transpose sequences |]
    where {
        sequences : [(k+1)][inf]b;
        sequences = [ins] # [| [| f x || x <- prev |]
                               || prev <- sequences
                               || i <- [1..k]
                               |];
    };
```

The graph and circuit generated from this function are exactly identical to iterate_stream. We can verify equivalence over a finite number of inputs, n, by using the following function as a query to the sat solver:

```
test_iterate_stream2 x = out1 != out2
    where {
        out1 = take (n, iterate_stream input);
        out2 = take (n, iterate_stream2 input);
        input = (x:[n]b) # zero;
        n = 10;
    };
```

In iterate_stream2, each infinite sequence in the comprehension could correspond to one stage in a pipeline. All we have to do is insert registers between the sequences, as in the definition of iterate_pipe in Figure 3-10.

FIGURE 3-10 *pipelined iterate*


```
iterate_pipe : [inf]b -> [inf]b;
iterate_pipe ins = [| x ! 0 || x <- transpose sequences |]
    where {
        sequences : [(k+1)][inf]b;
        sequences = [ins] # [| [undefined] # [| f x || x <- prev |]
                               || prev <- sequences
                               || i <- [1..k]
                               |];
    };
```

To understand what is going on within this pipelined implementation, compare the result of sequences in iterate_stream2 to the one in iterate_pipe:

For a given sequence of inputs [v w x y z ...], sequences in iterate_stream2 is as follows:

```
[   [ v f(v) f(f(v)) f(f(f(v))) ... ]
    [ w f(w) f(f(w)) f(f(f(w))) ... ]
    [ x f(x) f(f(x)) f(f(f(x))) ... ]
    [ y f(y) f(f(y)) f(f(f(y))) ... ]
    [ z f(z) f(f(z)) f(f(f(z))) ... ]
    ...
]
```

On the other hand, iterate_pipe calculates sequences as follows:

```
[   [ v   undefined  undefined  undefined ... ]
    [ w   f(v)    undefined  undefined ... ]
    [ x   f(w)    f(f(v))    undefined ... ]
    [ y   f(x)    f(f(w))    f(f(f(v))) ... ]
    [ z   f(z)    f(f(x))    f(f(f(w))) ... ]
    ...
]
```

So, transposing sequences yields an infinite stream of sequences; when k is 4, taking the last element of each of these sequences yields:

```
[   undefinedundefinedundefinedundefinedf(f(f(f(v))))
    f(f(f(f(w))))f(f(f(f(x))))f(f(f(f(y))))f(f(f(f(z))))...
]
```

The number of undefined outputs preceding the first valid output is always k.

We can test the pipelined implementation over a finite number of clock cycles. If we provide n inputs to iterate_stream and iterate_pipe and drop the first k outputs from iterate_pipe, then their outputs should be identical for n cycles. The following function tests if this is true:

```
test_iterate_pipe x = out1 != out2
    where {
        out1 = take (n, iterate_stream input);
        out2 = take (n, drop (k, iterate_pipe input));
        input = (x:[n]b) # zero;   n = 10;
    };
```

And the following command uses the equivalence checker to determine if there are any n inputs that can cause the functions to yield difference results:

```
Cryptol> :sat test_iterate_pipe
```

The strategy used to pipeline iterate, in which a sequence comprehension produces a finite number of infinite streams, will be used in Section 4.2 on page 47 to pipeline AES-128 and AES-256.

3.5.3 Pipelining via the *reg* Pragma

In order to pipeline the above examples, we had to lift each circuit into the stream model. This is because we need to have access to a stream that is mapped over time in order to delay it.

In this section we introduce a new pragma that allows the user to pipeline combinational code without lifting code into the stream model, and show how it can be applied to the examples above. This allows the user to pipeline code without changing the structure, yielding a pipelined implementation that more closely resembles the original specification.

When used as intended, this reg pragma causes a combinational circuit to be divided into smaller combinational circuits with registers between. Each application of reg generates a Delay-Undelay pair in the SPIR AST, so the net delay through the circuit is exactly 0. This allows us and the compiler to treat the circuit as combinational and without any notion of time. During the translation from SPIR

to LLSPIR, the circuit becomes sequential as the compiler uses specific rewrite rules to move the Delays and Undelays around while keeping the circuit synchronized with respect to time.

Unlike when the user pipelines by appending [undefined], the compiler is aware of the latency that the reg pragma introduces.The compiler will report the correct latency of the circuit, and when we lift the circuit into the stream model, we do not have to drop the undefined outputs from the beginning of the stream; the first element will be the first valid output.

Using the reg pragma, we can pipeline g and h as combinational circuits without changing the definition of add_g_h_spec:

```
g x = reg(reg(x+5) * 6);
h y = reg(reg(y*7) + 1);
```

Using these definitions of g and h, when add_g_h_spec is lifted into the stream model it will be identical the add_g_h_4 circuit that was manually pipelined above.

We can use the reg pragma to pipeline the iterate function so that when it is lifted it produces the same circuit as iterate_pipe:

```
iterate : b -> [(k+1)]b
iterate x = outs
     where outs = [x] #    [|reg(f prev)
                           || i <- [1..k]
                           || prev <- outs
                           |];
iterate_pipe' : b -> b;iterate_pipe' x = iterate x ! 0;
```

4 Implementing AES

This section documents the process of refining the AES reference specification in Appendix A on page 67 into synthesizable implementations of AES-128 and AES-256.

STEP 1 Remove constructs that are unsupported in the FPGA compiler, replace many of the functions with more efficient versions, and specialize the algorithm to two implementations, one for AES-128 and one for AES-256.

STEP 2 Define a second pair of implementations by defining a function that performs one round of key expansion and encryption for each algorithm and using this function to combine the KeyExpansion and Cipher functions into one top-level function.

STEP 3 Adjust the implementations to meet space and time requirements. We provide two specific implementations of each algorithm: a *small* implementation that uses the seq pragma to reuse the same key expansion and encryption circuitry over multiple clock cycles and a *fast* implementation that is pipelined via the par pragma.

STEP 4 Improve the AES-128 implementation by replacing the round of encryption with an equivalent and more efficient T-Box implementation.

STEP 5 Use the reg pragma to pipeline the AES-128 implementations

The resulting implementations are checked for equivalence with the original specification in symbolic, LLSPIR, and FPGA modes. We also present performance results with and without Block RAMs in both LLSPIR mode and TSIM mode.

4.1 Implementation 1: Make it synthesizable

In this section we replace many of the functions from the reference specification with much more efficient versions and specialize the encryption algorithm to two implementations, one for AES-128 and one for AES-256. Specifically, we perform the following changes:

- Replace gTimes with a more efficient implementation from the Rijndael spec, then rewrite using stream recursion;
- Eliminate gPower, replacing with specialized gPower2
- Eliminate gTimes, replacing with specialized gTimes2 and gTimes3
- Replace mixColumn with much more efficient implementation, through inlining and static evaluation; and
- Specialize KeyExpansion to AES-128 and AES-256.

These changes result in an AES implementation with very reasonable performance (see "Performance" on page 62).

4.1.1 Unsupported Constructs

The following are not supported by the Cryptol FPGA compiler and therefore should be removed when producing an implementation from the spec:

- Recursive functions, which can usually be replaced by stream recursion
- Functions that return non-closed functions

There are no recursive functions in the specification. However, the Cipher Duo record contains higher-order functions. We can remove the use of higher-order functions and inline into Cipher the functions that use Duo, yielding the following definition:

```
Cipher : {nk} (fin nk,8 >= width nk,nk >= 1) => (Key(nk),Block) -> Block;
Cipher(key,plaintext) = stateToBlock(finalRound) where {
    in = blockToState(plaintext);
    roundKeys = KeyExpansion(keyToWords(key));
    initialRound = AddRoundKey (in, roundKeys @ 0);
    medialRounds = [initialRound] #
        [| AddRoundKey(MixColumns(ShiftRows(SubBytes(state))), roundKey)
        || state <- medialRounds
        || roundKey <- roundKeys @@ [1 .. width(roundKeys) - 2] |];
    finalRound = AddRoundKey(ShiftRows(SubBytes(medialRounds ! 0)), roundKeys
! 0);
};
```

We will specialize this function to Cipher_128 and Cipher_256 below.

One can inline the use of gMatrixVectorProduct, gMatrixProduct, gDotProduct, and gSum into the definition of mixColumn, yielding a simpler and much more efficient implementation. Note that the value of aMatrix is known statically as:

```
[[0x2 0x3 0x1 0x1] [0x1 0x2 0x3 0x1] [0x1 0x1 0x2 0x3] [0x3 0x1 0x1 0x2]]
```

Given that the argument to mixColumn is [y0 y1 y2 y3], then the matrix product of the above aMatrix and this column argument is simply:

```
[ (gTimes(2,y0) ^ gTimes(3,y1) ^ gTimes(1,y2) ^ gTimes(1,y3))
  (gTimes(1,y0) ^ gTimes(2,y1) ^ gTimes(3,y2) ^ gTimes(1,y3))
  (gTimes(1,y0) ^ gTimes(1,y1) ^ gTimes(2,y2) ^ gTimes(3,y3))
  (gTimes(3,y0) ^ gTimes(1,y1) ^ gTimes(1,y2) ^ gTimes(2,y3))
```

In every call to gTimes, the first argument is either 1, 2 or 3. We can replace gTimes(1,x) with x, gTimes(2,x) with gTimes2(x) and gTimes(3,x) with gTimes3(x), where gTimes2 and gTimes3 are defined as follows:

```
gTimes2 : [8] -> [8];
gTimes2 x = (x << 1) ^ (if x ! 0 then <| x^4 + x^3 + x + 1 |> else zero);

gTimes3 : [8] -> [8];
gTimes3(b) = gTimes2(b) ^ b;
```

Thus, mixColumn can be re-written as follows:

```
mixColumn_prime : [4][8] -> [4][8];
mixColumn_prime([y0 y1 y2 y3]) = [ (gTimes2(y0) ^ gTimes3(y1) ^ y2 ^ y3)
                                   (y0 ^ gTimes2(y1) ^ gTimes3(y2) ^ y3)
                                   (y0 ^ y1 ^ gTimes2(y2) ^ gTimes3(y3))
                                   (gTimes3(y0) ^ y1 ^ y2 ^ gTimes2(y3))
                                 ];
```

The original gTimes is now used only by gPower. However, gPower is called only by Rcon and always with a constant first argument, 2. Therefore, we could rewrite gPower as gPower2:

```
gPower2 : {a} ([a]) -> [8];
gPower2(i) = ps @ i
  where ps = [1] # [| gTimes2(p) ||  <- ps |];
```

However, the latency of this implementation is 2^a, where a is the width of the input. This is because Cryptol implements synchronous circuits whose latency must be known statically; therefore, the latency of this circuit is equal to the worst-case latency. We can be more efficient by implementing it as a static lookup table. The following expression generates the sequence for the table:

```
[| gPower(2,x) || x <- [0..15] |]
```

gPower takes 8-bit inputs and therefore we would need a 256-element lookup table to represent its entire range. However, the argument to Rcon is a 4-bit round number, so only the first 16 elements of the table will ever be accessed. (In fact, because the round number never goes above 11, we could shorten the table even more.) Thus, we write gPower2_table as follows:

```
gPower2_table : [16][8];
gPower2_table = [0x01 0x02 0x04 0x08 0x10 0x20 0x40 0x80
                 0x1b 0x36 0x6c 0xd8 0xab 0x4d 0x9a 0x2f];
```

Some utility functions must be rewritten to use mixColumn_prime and gPower2_table*:

*We also stop using mapBytes and mapColumns because these higher-order functions are not supported in the C backend.

```
Rcon_prime : [4] -> [32];
Rcon_prime(i) = zero # (gPower2_table @ (i-1));

MixColumns_prime : [4][4][8] -> [4][4][8];
MixColumns_prime(state) = transpose(   [| mixColumn_prime column
                                       || column <- transpose(state)
                                       |]
                                                              );
```

For each replaced function above, we can use the equivalence checker to verify that the new version is equivalent to the original. For example, the following commands can be used to verify that gTimes2 and gTimes3 are correct with respect to the specification for gTimes:

```
Cryptol> :set symbolic
Cryptol> :eq (\x -> gTimes(2,x)) gTimes2
Cryptol> :eq (\x -> gTimes(3,x)) gTimes3
```

FIGURE 4-1 *The* KeyExpansion_128 *and* nextWord_128 *functions.*

```
KeyExpansion_128 : [4][32] -> [11][4][4][8];
KeyExpansion_128 keyAsWords = [| transpose s || s <- ss |]
 where {
   ss = groupBy(4,[| reverse(splitBy(4,w)) || w <- ws |]);
   ws = keyAsWords # ([| nextWord_128(i,w,w')
                      || i <- [4 .. 43]
                      || w <- ws
                      || w' <- drop(3,ws)
                      |]);
 };

nextWord_128 : ([8],[32],[32]) -> [32];
nextWord_128(i,w,w') = w ^ temp
 where {
    temp = if i % 4 == 0
    then SubWord(RotWord(w')) ^ Rcon_prime (take (4, i / 4))
    else w';
 };
```

To produce a more efficient implementation of KeyExpansion, we specialize it to a 128-bit key size by making two minor changes. First, we replace nextWord with nextWord_128, which does not have to check if the key size is more than six bytes. Second, we replace the infinite intermediate sequences ws and [Nk ..] with finite sequences.

In the specification, KeyExpansion is implemented using an infinite sequence of words ws that is defined by drawing i from an infinite sequence, [Nk ..]. KeyExpansion then draws a finite number of elements from that sequence using take. The FPGA compiler can produce a more efficient implementation if we draw i from [4..43], so that its contents can be evaluated at compile time, and implement ws as a finite sequence. The new implementation is defined in Figure 4-1 on

page 44. The same technique is used to define key expansion for AES-256 in Figure 4-2 on page 45.

FIGURE 4-2 *The KeyExpansion_256 and nextWord_256 functions.*

```
KeyExpansion_256 : [8][32] -> [15][4][4][8];
KeyExpansion_256 keyAsWords = [| transpose s || s <- ss |]
      where {
          ss = groupBy(4,[| reverse(splitBy(4,w)) || w <- ws |]);
          ws = keyAsWords # [| nextWord_256(i,w,w')
                             || i <- [8 .. 59]
                             || w <- ws
                             || w' <- drop(7,ws)
                             |];
      };

 nextWord_256 : ([8],[32],[32]) -> [32];
nextWord_256(i,w,w') = w ^ temp
      where {
          temp =  if i % 8 == 0
                  then SubWord(RotWord w') ^ Rcon_prime(take (4, i / 8))
                  else     if (i % 8 == 4)
                           then SubWord(w')
                           else w';
      };
```

Finally, Cipher_128 and Cipher_256 are written to use the new definitions (see Figure 4-3 on page 46), and we lift these definitions into the stream model as follows:

```
Cipher_128_stream : [inf]([128],[128]) -> [inf][128];
Cipher_128_stream ins = [| Cipher_128 in || in <- ins |];

Cipher_256_stream : [inf]([256],[128]) -> [inf][128];
Cipher_256_stream ins = [| Cipher_256 in || in <- ins |];
```

FIGURE 4-3 *The Cipher functions.*

```
Cipher_128 : ([128],[128]) -> [128];
Cipher_128(key,plaintext) = stateToBlock(finalRound) where {
    in = blockToState(plaintext);
    roundKeys =  KeyExpansion_128(keyToWords(key));
    initialRound = AddRoundKey (in, roundKeys @ 0);
    medialRounds = [initialRound] #
        [| AddRoundKey(MixColumns_prime(ShiftRows(SubBytes(state))),roundKey)
        || state <- medialRounds
        || roundKey <- roundKeys @@ [1 .. width(roundKeys) - 2]
        |];
    finalRound = AddRoundKey(ShiftRows(SubBytes(medialRounds ! 0)), roundKeys
! 0);
};

Cipher_256 : ([256],[128]) -> [128];
Cipher_256(key,plaintext) = stateToBlock(finalRound) where {
    in = blockToState(plaintext);
    roundKeys =  KeyExpansion_256(keyToWords(key));
    initialRound = AddRoundKey (in, roundKeys @ 0);
    medialRounds = [initialRound] #
        [| AddRoundKey(MixColumns_prime(ShiftRows(SubBytes(state))),roundKey)
        || state <- medialRounds
        || roundKey <- roundKeys @@ [1 .. width(roundKeys) - 2]
        |];
    finalRound = AddRoundKey(ShiftRows(SubBytes(medialRounds ! 0)), roundKeys
! 0);
};
```

4.2 Implementation 2

In this section we design functions that perform a single round of AES-128 and AES-256, respectively, and combine the KeyExpansion and Cipher functions into one. This implementation of AES is easier to apply the seq pragma to and to pipeline than the previous implementation.

Because the KeyExpansion function will be eliminated, some of the operations that it performed are moved into a new AddRoundKey_prime function:

```
AddRoundKey_prime : ([4][4][8],[4][32]) -> [4][4][8];
AddRoundKey_prime(state, keyAsWords) = state ^ roundKey
    where roundKey = transpose[| reverse (splitBy (4, word))
                                || word <- keyAsWords
                                                        |];
```

Rather than use nextWord to produce one word at a time, we can write a function that produces 4 words at a time. This function is specialized to AES-128 in Figure 4-4 on page 47, and is equivalent to four sequential applications of nextWord.

FIGURE 4-4 *The* nextKey *functions.*

```
nextKey_128 : ([4],[4][32]) -> [4][32];
nextKey_128(rnd, [w0 w1 w2 w3]) = [w0' w1' w2' w3'] where {
    w0' = SubWord(RotWord w3) ^ (Rcon_prime rnd) ^ w0;
    w1' = w0' ^ w1;
    w2' = w1' ^ w2;
    w3' = w2' ^ w3;
};

nextKey_256 : ([4],[4][32],[4][32]) -> [4][32];
nextKey_256(rnd, [w0 w1 w2 w3], [w4 w5 w6 w7]) = [w8 w9 w10 w11] where {
    w8  =   if rnd % 2 == 0
            then w0 ^ RotWord(SubWord w7) ^ Rcon_prime(rnd/2)
            else w0 ^ SubWord(w7);
    w9  = w1 ^ w8;
    w10  = w2 ^ w9;
    w11  = w3 ^ w10;
};
```

FIGURE 4-5 *One round of AES.*

```
oneRound128 : ([4], ([4][4][8],[4][32])) -> ([4][4][8],[4][32]);
oneRound128 (round, (state, key)) = (next_state, next_key)  where {
    state_prime =   if round == 1 then state
                    else if round == 11 then ShiftRows(SubBytes(state))
                    else MixColumns_prime(ShiftRows(SubBytes(state)));
    next_state = AddRoundKey_prime(state_prime, key);
};

oneRound256 : ([4], ([4][4][8],[4][32],[4][32])) -> ([4][4][8],[4][32],[4][32]);
oneRound256 (round, (state, key0, key1)) = (next_state, next_key0, next_key1)
where {
    state_prime =   if round == 1 then state
                    else if round == 15 then ShiftRows(SubBytes(state))
                    else MixColumns_prime(ShiftRows(SubBytes(state)));
    next_state = AddRoundKey_prime(state_prime, key0);
    next_key0 = key1;
    next_key1 = nextKey_256(round+1, key0, key1);
};
```

It is important to note that the SubWord and RotWord operations can be exchanged. This can be checked as follows:

```
Cryptol> :eq (\x -> SubWord (RotWord x)) (\x -> RotWord (SubWord x))
```

When pipelining AES in the following sections, exchanging these operations may produce a faster implementation.

The nextKey function for AES-256 is more complicated than the one for AES-128. Every other 128-bit key in AES-256 is calculated differently; we need to check if the round number is odd or even to determine which operation to perform. Also, each 128-bit key depends on the previous eight words (256 total bits), rather than just the previous four words. The definition of nextKey_256 is provided in Figure 4-4 on page 47.

To verify that the nextKey functions are correct, we can use them to implement key expansion functions and then equivalence check these against the original key expansion implementations. For example, following is an implementation of AES-256 key expansion using nextKey_256:

```
KeyExpansion_256_prime : [8][32] -> [15][4][4][8];
KeyExpansion_256_prime keyAsWords = [| transpose [| reverse (splitBy (4, w))
                                        || w <- words |]
                                    || words <- keys256 keyAsWords |];

keys256 : [8][32] -> [15][4][32];
keys256 keyAsWords = [| key0 || (key0,key1) <- keys |]
  where {
    init = (keyAsWords @@ [0..3], keyAsWords @@ [4..7]);
    keys = [init] #
           [| (key1, nextKey_256(round, key0, key1))
           || round <- [2..15]
           || (key0, key1) <- keys
           |];
  };
```

Then, we can check this for equivalence with the original KeyExpansion_256 function:

```
Cryptol> :set symbolic
Cryptol> :eq  KeyExpansion_256_prime  KeyExpansion_256
```

We can now implement oneRound128 and oneRound256, functions that perform a single round of encryption and key expansion. They take in the previous state and the key for this round, and produce the key for the next round and a new state that uses that key from this round. These functions are provided in Figure 4-5 on page 48.

Our new Cipher functions simply apply the appropriate oneRound function for each of the 11 or 15 rounds. They are defined in Figure 4-6 on page 50.

FIGURE 4-6 *The improved Cipher_prime functions.*

```
Cipher_128_prime : ([128],[128]) -> [128];
Cipher_128_prime (key, pt) = stateToBlock final_state
  where {
    init = (blockToState pt, keyToWords key);

    rounds = [init] #
             [| oneRound128 (round, state_and_key)
             || state_and_key <- rounds
             || round <- [1..11]
             |];
    (final_state, dont_care) = rounds ! 0
  };

Cipher_256_prime : ([256],[128]) -> [128];
Cipher_256_prime (key, pt) = stateToBlock final_state
  where {
    init = (blockToState pt, words @@ [0..3], words @@ [4..7])
      where words = keyToWords key;

    rounds = [init] #
             [| oneRound256 (round, state_and_keys)
             || state_and_keys <- rounds
             || round <- [1..15]
             |];
    (final_state, dont_care0, dont_care1) = rounds ! 0
  };
```

To prevent the Cipher functions above from being laid out over time, we lift them into the stream world:

```
Cipher_128_prime_stream : [inf]([128],[128]) -> [inf][128];
Cipher_128_prime_stream ins = [| Cipher_128_prime in || in <- ins |];


Cipher_256_prime_stream : [inf]([256],[128]) -> [inf][128];
Cipher_256_prime_stream ins = [| Cipher_256_prime in || in <- ins |];
```

These functions should be used whenever translating to a hardware implementation.

4.2.1 The *seq* Pragma: Minimizing Area

In this section we optimize the implementation from Section 4.2 on page 47 for area by reusing one round over multiple clock cycles. Cipher_128_prime and Cipher_256_prime were written with this goal in mind, so we can reuse the oneRound128 and oneRound256 functions and all we have to do is insert the seq pragma as shown in Figure 4-7. The seq pragma has the same effect here as it did in Section 3.4.

FIGURE 4-7 *Rewriting the Cipher function using the seq pragma.*

```
Cipher_128_seq : ([128],[128]) -> [128];
Cipher_128_seq (key, pt) = stateToBlock final_state
  where {
    init_state = blockToState pt;
    init_key = keyToWords key;

    rounds = [(init_state, init_key)] #
             seq [| oneRound128 (round, state_and_key)
                 || state_and_key <- rounds
                 || round <- [1..11]
                 |];
    (final_state, bla) = rounds ! 0
  };

Cipher_256_seq : ([256],[128]) -> [128];
Cipher_256_seq (key, pt) = stateToBlock final_state
  where {
    init = (blockToState pt, words @@ [0..3], words @@ [4..7])
      where words = keyToWords key;

    rounds = [init] #
             seq [| oneRound256 (round, state_and_keys)
                 || state_and_keys <- rounds
                 || round <- [1..15]
                 |];
    (final_state, dont_care0, dont_care1) = rounds ! 0
  };
```

To implement these in hardware, they should be lifted into the stream model:

```
Cipher_128_seq_stream : [inf]([128],[128]) -> [inf][128];
Cipher_128_seq_stream ins = [| Cipher_128_seq in || in <- ins |];

Cipher_256_seq_stream : [inf]([256],[128]) -> [inf][128];
Cipher_256_seq_stream ins = [| Cipher_256_seq in || in <- ins |];
```

4.2.2 The *par* Pragma: Increasing Throughput by Unrolling Loops

In this section we manually derive pipelined implementations of AES-128 and AES-256 from the ones in Section 4.2 on page 47. It is intended mostly to show how one can manually pipeline other algorithms, rather than actually provide a good pipelined implementation of AES. The other pipelined implementations in this paper are easier to design and each yield equivalent or better performance than this one.

FIGURE 4-8 *Unrolling the rounds: the pipelined Cipher functions.*

```
Cipher_128_pipe : [inf]([128],[128]) -> [inf][128];
Cipher_128_pipe ins = outs
  where {
    inits = [| (blockToState pt, keyToWords key)
            || (key, pt) <- ins |];

    roundss = [inits] #
                [| add_delay(round, [| oneRound128 (round, state_and_key)
                                    || state_and_key <- rounds |])
                || rounds <- roundss
                || round <- [1..11]
                |];
    outs = [| stateToBlock final_state
           || (final_state, dont_care) <- roundss ! 0
           |];
  };
Cipher_256_pipe : [inf]([256],[128]) -> [inf][128];
Cipher_256_pipe ins = outs  where {
    inits = [| (blockToState pt, words @@ [0..3], words @@ [4..7])
               where words = keyToWords key
            || (key, pt) <- ins |];

    roundss = [inits] #
                [| add_delay(round, [| oneRound256 (round, state_and_keys)
                                    || state_and_keys <- rounds |])
                || rounds <- roundss
                || round <- [1..15]
                |];
    outs = [| stateToBlock final_state
           || (final_state, dont_care0, dont_care1) <- roundss ! 0
           |];
  };
```

In order to insert registers into it, the rounds loop must be lifted into the stream model. We can only insert registers by prepending [undefined] to *infinite* streams, or streams mapped across time. The following definition of AES-128 is defined in the stream model:

```
Cipher_128_stream : [inf]([128],[128]) -> [inf][128];
Cipher_128_stream ins = outs where {
     inits = [| (blockToState pt, keyToWords key) || (key, pt) <- ins |];

     roundss = [inits] #
          [| [| oneRound128 (round, state_and_key)  || state_and_key <- rounds |]
          || rounds <- roundss
          || round <- [1..11]
          |];
     outs = [| stateToBlock final_stat || (final_state, dont_care) <- roundss ! 0 |];
};
```

Once we have formulated the algorithm this way, pipelining it is very easy. We simply insert a register in front of each round in the loop:

```
Cipher_128_pipe : [inf]([128],[128]) -> [inf][128];
Cipher_128_pipe ins = outs where {
    inits = [| (blockToState pt, keyToWords key) || (key, pt) <- ins |];

    roundss = [inits] #
        [| [undefined] #  [| oneRound128 (round, state_and_key)
                          || state_and_key <- rounds
                          |]
        || rounds <- roundss
        || round <- [1..11]
        |];
    outs =   [| stateToBlock final_state
             || (final_state, dont_care) <- roundss ! 0
             |];
};
```

We can define a much more flexible implementation that allows the number of stages to be anywhere between 0 and the number of rounds, inclusive. First, we define a predicate function, check_stage, which takes in a round number and returns True if a register should be inserted after that round, and add_delay, which conditionally inserts a register after a round. To insert a pipeline stage between every round, simply define check_stage i = True. add_delay and check_stage are defined as follows:

```
add_delay (i, x) = if check_stage i then ([undefined] # x) else x;

check_stage : {a} (fin a, a >= 4) => [a] -> Bit;
check_stage i = True; // (i%3 == 1);
```

We can now replace registers with a call to add_delay in our pipelined implementation. The definition and a high-level graph of the final pipelined implementations are provided in Figure 4-8 on page 52.

The number of stages in our pipelined implementation can be determined by the following function:

```
num_stages : [8] -> [8];
num_stages num_rounds = count @ num_rounds
    where count = [0] #  [| if check_stage i then c+1 else c
                         || c <- count
                         || i <- [1..]
                                                |];
```

To pipeline within each round, we would need to lift the rounds function itself into the stream model. Alternatively, we can use the reg pragma to pipeline a combinational circuit without lifting it into the stream model (see "Pipelined Using the reg Pragma" on page 53 and "T-Box Implementation" on page 55).

4.2.3 Pipelined Using the *reg* Pragma

In this section we use the reg pragma to pipeline the implementation from Section 4.2 on page 47. This code compiles to the exact same circuit as the one in "Pipelining the T-Box Implementation Using the reg Pragma" on page 57, but it is

much simpler. All we do is apply the reg pragma to each round. The function is defined in Figure 4-9 on page 54.

We use the same method as before to conditionally insert registers after each stage:

```
add_reg (i, x) = if check_stage i then (reg x) else x;

check_stage : {a} (fin a, a >= 4) => [a] -> Bit;
check_stage i = True; // (i%3 == 1);
```

FIGURE 4-9 *Rewriting Cipher_128 using the reg pragma*

```
Cipher_128_reg : ([128],[128]) -> [128];
Cipher_128_reg (key, pt) = stateToBlock final_state
  where {
    init_state = blockToState pt;
    init_key = keyToWords key;

    rounds = [(init_state, init_key)] #
                [| add_reg(round, oneRound128 (round, state_and_key))
                || state_and_key <- rounds
                || round <- [1..11]
                |];
    (final_state, bla) = rounds ! 0
  };
```

To implement this in hardware, it should be lifted into the stream model:

```
Cipher_128_reg_stream : [inf]([128],[128]) -> [inf][128];
Cipher_128_reg_stream ins = [| Cipher_128_reg in || in <- ins |];
```

4.3 T-Box Implementation

In this section we replace oneRound128 with an equivalent T-Box implementation that is more efficient. In the next section we will use the reg pragma to produce an very high throughput pipeline from this implementation.

One round is defined in Figure 4-10 on page 56; it is a drop-in replacement for oneRound128. The implementation uses the following utility functions:

```
T0, T1, T2, T3 : [8] -> [4][8];
T0(a) = T0_table @ a;
T1(a) = T0(a) >>> 1;
T2(a) = T0(a) >>> 2;
T3(a) = T0(a) >>> 3;

T0_table = const [| T0_func(a) || a <- [0..255] |];

T0_func : [8] -> [4][8];
T0_func(a) = [(gTimes2 s) s s (gTimes3 s)] where s = SBox @ a;
```

The top-level Cipher function is the same as before, but we use oneRound128_Tbox instead of oneRound128. It is defined in Figure 4-11 on page 56.

This T-Box implementation does not realize a very impressive increase in performance. This is because key expansion is the bottleneck in the algorithm. To obtain a high clockrate from this T-Box implementation, we can pipeline both the key expansion and encryption rounds using the reg pragma (see Section 4.3, "T-Box Implementation," on page 55).

These functions should be used whenever translating to a hardware implementation:

```
Cipher_128_Tbox_stream : [inf]([128],[128]) -> [inf][128];
Cipher_128_Tbox_stream ins = [| Cipher_128_Tbox in || in <- ins |];
```

FIGURE 4-10 *One Round of AES-128, using T-Box.*

```
oneRound128_Tbox : ([4], ([4][4][8],[4][32])) -> ([4][4][8],[4][32]);
oneRound128_Tbox (round, (state, key)) = (next_state, next_key)
  where {
    state_prime = if round == 1 then state
                  else transpose d;
    next_state = AddRoundKey_prime(state_prime, key);
    next_key = nextKey_128(round, key);

    d = [| if (round == 11)
           then [(t0@1) (t1@2) (t2@3) (t3@0)]
           else t0 ^ t1 ^ t2 ^ t3
         where {
           t0 = T0 (state @ 0 @ (j+0));
           t1 = T1 (state @ 1 @ (j+1));
           t2 = T2 (state @ 2 @ (j+2));
           t3 = T3 (state @ 3 @ (j+3));
         }
      || j <- [0 .. 3]
      |];
  };
```

FIGURE 4-11 *AES-128, using T-Box*

```
Cipher_128_Tbox : ([128],[128]) -> [128];
Cipher_128_Tbox (key, pt) = stateToBlock final_state
  where {
    init = (blockToState pt, keyToWords key);

    rounds = [init] #
             [| oneRound128_Tbox (round, state_and_key)
             || state_and_key <- rounds
             || round <- [1..11]
             |];
    (final_state, dont_care) = rounds ! 0
  };
```

4.3.1 Minimizing the Space Requirements of the Tbox Implementation

In this section we use the seq pragma to optimize the T-Box implementation for area, reusing the round each cycle. We use the same technique as in Section 3.4 and Section 4.2. The implementation reuses the oneRound128_Tbox function from the previous section, and is defined in Figure 4-12 on page 57.

These functions should be used whenever translating to a hardware implementation:

```
Cipher_128_Tbox_seq_stream : [inf]([128],[128]) -> [inf][128];
Cipher_128_Tbox_seq_stream ins = [| Cipher_128_Tbox_seq in || in <- ins |];
```

FIGURE 4-12 *AES-128, using T-Box, reused over time*

```
Cipher_128_Tbox_seq : ([128],[128]) -> [128];
Cipher_128_Tbox_seq (key, pt) = stateToBlock final_state
  where {
    init = (blockToState pt, keyToWords key);

    rounds = [init] #
             seq [| oneRound128_Tbox (round, state_and_key)
                 || state_and_key <- rounds
                 || round <- [1..11]
                 |];
    (final_state, dont_care) = rounds ! 0
  };
```

4.3.2 Pipelining the T-Box Implementation Using the *reg* Pragma

In this section we use the reg pragma to pipeline the AES-128 T-Box implementation. We have decided to pipeline each round to 5 stages and target a clockrate of about 400 MHz (2.5 ns).

We should attempt to implement the same number of stages for both nextKey and oneRound, because they execute in parallel. If we do not use the same number of stages in each function, the compiler will simply insert registers to keep the circuits synchronized, but the registers will not be optimally placed. Also, we should use Virtex 4 Block RAMs with 2-cycle latency; they have a higher clockrate, and the extra latency simply allows us to do more in parallel with the Block RAM. Therefore, to obtain 5 stages per round we should insert 3 more registers into both nextKey and oneRound.

Each Block RAM behaves as a register, so the SBox and Tbox operations already define some of the pipeline stages. We are defining a pipeline with many small stages, so the latency of Block RAMs will dominate. Therefore, the input to a Block RAM should never be calculated in the same stage as the Block RAM because this would increase the delay of that stage. For example, we should place a register between the RotWord and SubWord operations so that they are performed in separate stages. Alternatively, we can exchange these operations as this may result in a more balanced pipeline.

Because compiling to LLSPIR is relatively quick and provides us with an estimated clockrate, one can experiment with many different combinations and placements of reg in search of the fastest possible clockrate.

FIGURE 4-13 *Pipelined Key Expansion for AES-128*

```
nextKey_128_reg : ([4],[4][32]) -> [4][32];
nextKey_128_reg(rnd, [w0 w1 w2 w3]) = [w0' w1' w2' w3']
   where {
     w0' = RotWord(SubWord w3) ^ (Rcon_prime rnd) ^ w0;
     w1' = reg(w0' ^ w1);
     w2' = (w1' ^ w2);
     w3' = reg(w2' ^ w3);
 };
```

FIGURE 4-14 *Heavily Pipelined Round of AES-128, using T-Box*

```
oneRound128_Tbox_reg : ([4], ([4][4][8],[4][32])) -> ([4][4][8],[4][32]);
oneRound128_Tbox_reg (round, (state, key)) = (next_state, next_key)
  where {
    state_prime = if round == 1 then state
            else transpose d;
    next_state = reg(AddRoundKey_prime(state_prime, key));
    next_key = nextKey_128_reg(round, key);

   d = [| reg(if (round == 11)
             then [(t0@1) (t1@2) (t2@3) (t3@0)]
             else reg(t0 ^ t1) ^ reg(t2 ^ t3))
          where {
            t0 = T0 (state @ 0 @ (j+0));
            t1 = T1 (state @ 1 @ (j+1));
            t2 = T2 (state @ 2 @ (j+2));
            t3 = T3 (state @ 3 @ (j+3));
          }
      || j <- [0 .. 3]
      |];
 };
```

The nextKey circuit performs the following, sequentially:

- The following, in parallel:
  - SubWord and RotWord
  - Rcon and xor with w0
- Xor the previous two results
- Xor with w1
- Xor with w2
- Xor with w3

Because it is implemented using a Virtex 4 Block RAM, the SubWord operation will take two cycles. We can check the propagation delay of the other operations by executing the following commands in LLSPIR mode:

```
Cryptol> :translate (\(x, rnd) -> Rcon_prime rnd ^ x)
Cryptol> :translate (\(x,y,z) -> (x:[32]) ^ y ^ z)
Cryptol> :translate (\(x,y,z,w) -> (x:[32]) ^ y ^ z ^ w)
Cryptol> :translate (\(x,y,z) -> (RotWord x) ^ y ^ z)
```

From this we find that step 1B can fit within 2 cycles and therefore can execute in parallel with SubWord. We also discover that two xor operations can fit within one cycle, RotWord and two xor operations can, but three xor operations cannot. Therefore, we should implement the nextKey_prime stages as follows:

- SubWord in parallel with Rcon_prime rnd ^ w0 (this takes 2 cycles)
- RotWord and two xor operations
- Two more xor operations

This pipeline is implemented in Figure 4-13 on page 58.

Pipelining one round of encryption is fairly straightforward. We define the following stages:

- Four parallel T-Box lookups
- t0 ^ t1 in parallel with t2 ^ t3, then xor the results
- AddRoundKey_prime

The function in Figure 4-14 on page 58 implements this pipeline.

FIGURE 4-15 *Heavily Pipelined AES-128, using T-Box*

```
Cipher_128_Tbox_reg : ([128],[128]) -> [128];
Cipher_128_Tbox_reg (key, pt) = stateToBlock final_state
  where {
    init = (blockToState pt, keyToWords key);

    rounds = [init] #
             [| oneRound128_Tbox_reg (round, state_and_key)
             || state_and_key <- rounds
             || round <- [1..11]
             |];
    (final_state, dont_care) = rounds ! 0
  };
```

Our top-level function is defined in Figure 4-15 on page 59; it is identical to the one in the previous section, except that it uses oneRound128_Tbox_reg for the oneRound function.

In TSIM mode (after place-and-route), this implementation yields a circuit with a rate of one element per cycle that can be clocked between 350 MHz (44.8 Gbps) and 450 MHz (57.6 Gbps), depending on the FPGA part.

To implement this in hardware, it should be lifted into the stream model:

```
Cipher_128_Tbox_reg_stream : [inf]([128],[128]) -> [inf][128];
Cipher_128_Tbox_reg_stream ins = [| Cipher_128_Tbox_reg in || in <- ins |];
```

4.4 Testing and Verification

In this section we verify that the implementations in Section 4.1, "Implementation 1: Make it synthesizable," on page 42 and Section 4.2, "Implementation 2," on page 47} are equivalent to the reference specifications for AES-128 and AES-256.

First, we generate formal models of the reference specification in symbolic mode:

```
Cryptol> :l AES_Revisited.tex

Cryptol> :set symbolic
Cryptol> :fm (Cipher : ([128],[128]) -> [128]) "aes128-spec.fm"
Cryptol> :fm (Cipher : ([256],[128]) -> [128]) "aes256-spec.fm"
```

Then, we generate formal models for the first implementation in symbolic mode and check them against the reference specification. Because the implementation is much more efficient than the spec, we will check all further implementations against the first implementation, rather than against the reference specification.

```
Cryptol> :l aes-imp1.tex

Cryptol> :set symbolic
Cryptol> :fm Cipher_128 "aes128-imp1.fm"
Cryptol> :fm Cipher_256 "aes256-imp1.fm"

Cryptol> :eq "aes128-imp1.fm" "aes128-spec.fm"
Cryptol> :eq "aes256-imp1.fm" "aes256-spec.fm"
```

We also check that the implementation is correct in LLSPIR, FSIM, and TSIM modes by issuing the following commands in each of these modes:

```
Cryptol> :eq Cipher_128 "aes128-imp1.fm"
Cryptol> :eq Cipher_256 "aes256-imp1.fm"
```

And we check all other implementations against the first one in symbolic, LLSPIR, FSIM, and TSIM modes:

```
Cryptol> :eq Cipher_128_prime "aes128-imp1.fm"
Cryptol> :eq Cipher_256_prime "aes256-imp1.fm"

Cryptol> :eq Cipher_128_seq "aes128-imp1.fm"
Cryptol> :eq Cipher_256_seq "aes256-imp1.fm"
```

To test the pipelined implementation, we must write a wrapper that tests the circuit for a finite number of clock cycles:

```
do_Cipher_128_pipe : ([128],[128]) -> [128];
do_Cipher_128_pipe x = Cipher_128_pipe ([x] # zero) @ (num_stages 11);

do_Cipher_256_pipe : ([256],[128]) -> [128];
do_Cipher_256_pipe x = Cipher_256_pipe ([x] # zero) @ (num_stages 15);
```

And then use the `:eq` command on this wrapper:

```
Cryptol> :eq do_Cipher_128_pipe "aes128-imp1.fm"
Cryptol> :eq do_Cipher_256_pipe "aes256-imp1.fm"
```

We can also verify that the pipelines for AES-128 and AES-256 are correct for some finite number of clock cycles by defining the following queries:

```
check_Cipher_128_pipe : ([128], [128]) -> Bit;
check_Cipher_128_pipe input = bools ! 0
  where {
    stages = num_stages 11;
    bools = [False] # [| prev | (ct0 != ct)
                  || prev <- bools
                  || ct <- [ct1 ct2 ct3 ct4 ct5] |];
    ct0 = Cipher_128_pipe ([input] # undefined) @ stages;
    ct1 = Cipher_128_pipe ((take(1,undefined)) # [input] # undefined) @ (stages+1);
    ct2 = Cipher_128_pipe ((take(2,undefined)) # [input] # undefined) @ (stages+2);
    ct3 = Cipher_128_pipe ((take(3,undefined)) # [input] # undefined) @ (stages+3);
    ct4 = Cipher_128_pipe ((take(4,undefined)) # [input] # undefined) @ (stages+4);
    ct5 = Cipher_128_pipe ((take(5,undefined)) # [input] # undefined) @ (stages+5);
  };
check_Cipher_256_pipe : ([256], [128]) -> Bit;
check_Cipher_256_pipe input = bools ! 0
  where {
    stages = num_stages 15;
    bools = [False] # [| prev | (ct0 != ct)
                  || prev <- bools
                  || ct <- [ct1 ct2 ct3 ct4 ct5] |];
    ct0 = Cipher_256_pipe ([input] # undefined) @ stages;
    ct1 = Cipher_256_pipe ((take(1,undefined)) # [input] # undefined) @ (stages+1);
    ct2 = Cipher_256_pipe ((take(2,undefined)) # [input] # undefined) @ (stages+2);
    ct3 = Cipher_256_pipe ((take(3,undefined)) # [input] # undefined) @ (stages+3);
    ct4 = Cipher_256_pipe ((take(4,undefined)) # [input] # undefined) @ (stages+4);
    ct5 = Cipher_256_pipe ((take(5,undefined)) # [input] # undefined) @ (stages+5);
  };
```

And posing these queries to the sat solver as follows:

```
Cryptol> :sat check_Cipher_128_pipe
Cryptol> :sat check_Cipher_256_pipe
```

4.5 Performance

This section summarizes the performance obtained by the AES implementations presented previously in this section. We present numerical results for logic utilization, latency, clockrate, and throughput.

The following implementations have been provided in this tutorial:

TABLE 4-1 *Summary of AES implmentations developed in this tutorial.*

tag	source code	characteristics	reference
1	aes-imp1.tex	Unrolled. When using Block RAMs, automatically pipelined to one pipeline stage per round.	page 42
2	aes-imp2.tex	Using a oneRound function, combined key expansion and cipher functions; unrolled, and pipelined when using Block RAMs.	page 47
2A	aes-seq.tex	Using the seq pragma to reuse one round	page 50
2B	aes-pipe.tex	Manually pipelined	page 51
2C	aes-reg.tex	Pipelined using the reg pragm	page 53
3	aes-tbox.tex	Replacing each round of encryption from aes-imp2.tex with a T-Box implementation (AES-128 only); unrolled, and pipelined when using Block RAMs	page 55
3A	aes-tbox-seq.tex	Using the seq pragma to reuse one round	page 56
3B	aes-tbox-reg.tex	Pipelining the T-Box implementation using the reg pragma	page 57

The performance results for some of these implementations are provided in Table on page 62.

Inserting Block RAMs into an unrolled implementation effectively pipelines the design to be one stage per round. Therefore, there is no need to report results for the manually pipelined implementations 2b, since they crash without Block RAMs and are identical to implementations 1 and 2 when using Block RAMs. Also, while implementations 1 and 2 are structured differently in Cryptol, they compile to extremely similar circuits with almost identical performance. Therefore, we only provide performance results for implementations 1, 2a, 3, 3a, and 3b.

TABLE 4-2 *Space/Time Characteristics for the AES Implementations*

		Resource utilization			Performance					
					Latency (clock cycles)	Rate (clock cycles)	Clockrate (Mhz)		Throughput (Gpbs)	
Tag	Keysize	LUTs	Flip-flops	BRAMs			FSIM	TSIM	FSIM	TSIM
1	128	52158	0	0	1	1	26.0		3.3	
		4530	1280	100	11	1	203.0	144.4	26.0	18.5
	256	50343	0	0	1	1	20.0		2.6	
		6132	3456	138	15	1	208.1	150.0	26.6	19.2
2A	128	10565	498	0	11	11	109.2	100.3	1.3	1.2
		1405	457	10	12	11	168.5	110.2	2.0	1.3
	256	10919	670	0	15	15	110.6	100.2	0.9	0.9
		1565	457	10	16	15	168.5	150.9	1.4	1.3
3	128	4096	1280	100	11	1	213.2	156.0	27.3	20.0
3A	128	1134	461	10	12	11	179.2	143.5	2.09	1.67
3B	128	6336	12576	100	53	1	385.5	370.0	49.3	47.4

When not using Block RAMs in an unrolled AES, the Xilinx tools sometimes run out of memory during synthesis, and always crash before finishing place-and-route. Thus, the clockrate and throughput numbers are missing from Table .

Both FSIM (synthesis only) and TSIM (synthesis with place-and-route) results are presented.

We obtain the latency and IO rate of the implementations by translating their stream versions in LLSPIR mode:

```
Cryptol> :set LLSPIR
Cryptol> :set +v
Cryptol> :l aes-imp1.tex
Cryptol> :translate Cipher_128_stream
Cryptol> :translate Cipher_256_stream
Cryptol> :l aes-imp2.tex
Cryptol> :translate Cipher_128_prime_stream
Cryptol> :translate Cipher_256_prime_stream
Cryptol> :l aes-seq.tex
Cryptol> :translate Cipher_128_seq_stream
Cryptol> :translate Cipher_256_seq_stream
Cryptol> :l aes-pipe.tex
Cryptol> :translate Cipher_128_pipe
Cryptol> :translate Cipher_256_pipe
Cryptol> :l aes-tbox.tex
Cryptol> :translate Cipher_128_Tbox_stream
Cryptol> :l aes-tbox-seq.tex
Cryptol> :translate Cipher_128_Tbox_seq_stream
```

Both latency and IO rate are reported in number of clock cycles.

All performance results are obtained by synthesizing for a specific FPGA part. This part has 63,168 slices, each with two LUTs and two flip-flops, for a total of 126,336 LUTs and 126,336 flip-flops, and 552 Block RAMs.

```
Cryptol> :set fpga_part=xc4vfx140-10ff1517
Cryptol> :set fpga_part_stepping=4
```

This choice greatly influences the results. Some other FPGA parts yield smaller and/or faster implementations. For example, most or all of these implementations are faster under the following FPGA settings:

```
Cryptol> :set fpga_part=xc4vlx60-12ff668
Cryptol> :set fpga_part_stepping=2
```

The effort level (fpga_optlevel) is at its default, minimal setting. A higher effort level could yield better performance in TSIM mode.

We test most implementations once using no Block RAMs and once using one-cycle latency Block RAMs, which can be set using the following commands, respectively:

```
Cryptol> :set fpga_blockram=none
Cryptol> :set fpga_blockram=behavioural
```

The T-Box implementation is an exception. We test this using one-cycle latency Block RAMs for the non-pipelined versions (implementations 3 and 3a) and two-cycle latency Block RAMs for the pipelined version (implementation 3b).

4.6 In Conclusion

Cryptol is a domain-specific language that is ideal for representing data-flow algorithms, particularly cryptographic algorithms. Cryptol's types and language constructs also provide an appropriate abstraction of hardware. Along with the interpreter and supporting tools, the Cryptol toolchain provides a framework for producing verified, high-speed FPGA implementations.

We have introduced the types and constructs of the Cryptol language, as well as the Cryptol interpreter, and in Section 3, "Cryptol & Hardware Design," on page 21 provided several examples of how to use the language and toolchain to specify, implement, refine, and verify hardware circuits for an FPGA. We then used these techniques to produce and refine several implementations of AES, including implementations with very little resource utilization and pipelined implementations with very high throughput.

The LLSPIR compiler allows us to view the results of optimizations, including the latency, rate, estimated area utilization, and estimated clockrate, without performing lengthy synthesis. This greatly reduces the time to produce new refinements.

Once the user has formulated a circuit the right way - as an appropriate sequence comprehension - the seq, par, and reg pragmas provide simple, effective ways of making space-time trade-offs and implementing pipelines without sacrificing the high-level, readable format of the source code.

In the future, the Cryptol team plans to increase the accuracy of profiling in LLSPIR, support more hardware architectures, and implement automated and guided pipelining.

A The AES Reference Specification

```
// -------------------------------------------------------------
// reference spec for AES algorithm
// useful only in bit model
// polynomial arithmetic is not supported in symbolic mode, SPIR, LLSPIR, VHDL, etc...
// thus, cannot perform equivalence checking with this spec
// -------------------------------------------------------------

include "../rtlib/polynomial.cry";

// -----------------------------------------------------
// Type synonyms
// -----------------------------------------------------

type ByteWidth = 8;
type WordWidth = 32;     // 4 bytes
type BlockWidth = 128;   // 4 words
type Byte = [ByteWidth];
type Word = [WordWidth];
type Block = [BlockWidth];
type State = [4][4]Byte;   // 4x4x8 = 128 (block size)

type Column = [4]Byte;

// nk is 4, 6 or 8
// so Key is 128, 192, or 256
type Key(nk) = [nk * WordWidth];

type KeyAsWords(nk) = [nk]Word;

// nr is 10, 12, or 14
// there are 11, 13, or 15 round keys
type RoundKeys(nr) = [nr + 1]State;

type Map = State -> State;

// a bijection that holds a map and its inverse
```

```
type Permutation = {en : Map; de : Map};

// the top-level AES functions
AES128 : {encrypt : (Key(4),Block) -> Block; decrypt : (Key(4),Block) -> Block};
AES128 = {encrypt = Cipher; decrypt = InvCipher};
AES192 : {encrypt : (Key(6),Block) -> Block; decrypt : (Key(6),Block) -> Block};
AES192 = {encrypt = Cipher; decrypt = InvCipher};
AES256 : {encrypt : (Key(8),Block) -> Block; decrypt : (Key(8),Block) -> Block};
AES256 = {encrypt = Cipher; decrypt = InvCipher};

// ------------------------------------------------------
// Utility functions
// ------------------------------------------------------

blockToState : Block -> State;
blockToState(x) = transpose(split(reverse(split(x))));

stateToBlock : State -> Block;
stateToBlock(x) = join(reverse(join(transpose(x))));

keyToWords : {nk} (fin nk) => Key(nk) -> KeyAsWords(nk);
keyToWords(key) = reverse(split(key));

// ------------------------------------------------------
// polynomial helpers

mixColumn : Column -> Column;
mixColumn(column) = matrixVectorProduct(transpose(columns),column)
  where columns = [| [ 0x02 0x01 0x01 0x03 ] >>> i || i <- [0 .. 3] |];

invMixColumn : Column -> Column;
invMixColumn(column) = matrixVectorProduct(transpose(columns),column)
  where columns = [| [ 0x0e 0x09 0x0d 0x0b ] >>> i || i <- [0 .. 3] |];

matrixVectorProduct : (State,Column) -> Column;
matrixVectorProduct(matrix,column) = join(mmult(matrix,split(column)));

mmult : {a b c} (fin b) => ([a][b]Byte,[b][c]Byte) -> [a][c]Byte;
mmult(xss,yss) = [| [| dot(row,col) || col <- transpose(yss) |] || row <- xss |];

dot : {a} (fin a) => ([a]Byte,[a]Byte) -> Byte;
dot(xs,ys) = sum([| gtimes(x,y) || x <- xs || y <- ys |]);

// parity
sum : {a b} (fin a) => [a][b] -> [b];
sum(xs) = (sums xs) ! 0 ;

sums : {a b} [a][b] -> [a+1][b];
sums(xs) = ys
  where ys = [0] # [| x ^ y || x <- xs || y <- ys |];

// multiplication in galois field
gtimes : (Byte,Byte) -> Byte;
gtimes(x,y) = take (8, pmod(pmult(x,y), <| x^8 + x^4 + x^3 + x + 1 |>));
```

```
gpower : {a} (Byte,[a]) -> Byte;
gpower(x,i) = ps @ i
  where ps = [1] # [| gtimes(x,p) || p <- ps |];

// ------------------------------------------------------
// AES algorithm specification
// ------------------------------------------------------

// ----------
// Cypher

Cipher : {nk} (fin nk,8 >= width nk,nk >= 1) => (Key(nk),Block) -> Block;
Cipher(key,plaintext) = stateToBlock(out)
  where {
    in = blockToState(plaintext);
    roundKeys = KeyExpansion(keyToWords(key));
    maps = [| (bijection : Permutation).en || bijection <- bijections(roundKeys) |];
    out = applyMany(maps,in);
  };

InvCipher : {nk} (fin nk,8 >= width nk,nk >= 1) => (Key(nk),Block) -> Block;
InvCipher(key,ciphertext) = stateToBlock(out)
  where {
    in = blockToState(ciphertext);
    roundKeys = KeyExpansion(keyToWords(key));
    maps = reverse([| (bijection : Permutation).de || bijection <- bijections(roundKeys) |]);
    out = applyMany(maps,in);
  };

bijections : {nr} (fin nr,nr >= 1) => RoundKeys(nr) -> [nr + 1]Permutation;
bijections(roundKeys)
  = [ (initialRoundPermutation(roundKeys @ 0)) ]
    # [| (medialRoundPermutation(roundKey)) || roundKey <- roundKeys @@ [1 ..
width(roundKeys) - 2] |]
    # [ (finalRoundPermutation(roundKeys ! 0)) ];

initialRoundPermutation : State -> Permutation;
initialRoundPermutation(roundKey) = {
  en(state) = AddRoundKey(state,roundKey);
  de(state) = AddRoundKey(state,roundKey)
};

medialRoundPermutation : State -> Permutation;
medialRoundPermutation(roundKey) = {
    en(state) = AddRoundKey(MixColumns(ShiftRows(SubBytes(state))),roundKey);
    de(state) = InvSubBytes(InvShiftRows(InvMixColumns(AddRoundKey(state,round-
Key))))
};

finalRoundPermutation : State -> Permutation;
finalRoundPermutation(roundKey) = {
  en(state) = AddRoundKey(ShiftRows(SubBytes(state)),roundKey);
  de(state) = InvSubBytes(InvShiftRows(AddRoundKey(state,roundKey)))
```

```
};

// Transformations
// --------

// ShiftRows()
ShiftRows : Map;
ShiftRows state = [| row <<< i || row <- state || i <- [ 0 .. 3 ] |];

// SubBytes()
SubBytes : Map;
SubBytes(state) = mapBytes(f,state)
  where f(b) = sbox @ b;

sbox : [256]Byte;
sbox = [| affine(inverse(x)) || x <- [0 .. 255] |];

affine : Byte -> Byte;
affine(xs) = join(mmultBit(affMat,split(xs))) ^ 0x63;

affMat : {a} (a >= 8) => [8][a];
affMat = [ 0xf1 ] # [| x <<< 1 || x <- affMat || i <- [1 .. 7] |];

mmultBit : {a b} ([a][8],[8][b]) -> [a][b];
mmultBit(xss,yss) = [| [| dotBit(row,col) || col <- transpose yss |] || row <- xss |];

dotBit : (Byte,Byte) -> Bit;
dotBit(as,bs) = parity([| a & b || a <- as || b <- bs |]);

inverse : Byte -> Byte;
inverse(x) = if x == 0 then 0 else find1(zs,0)
  where {
    zs = [| gtimes(x,y) || y <- [0 .. 255] |];
    find1(zs,i) = if zs @ i == 1 then i else find1(zs,i + 1);
  };


// MixColumns()
MixColumns : Map;
MixColumns(state) = mapColumns(mixColumn,state);

// AddRoundKey()
AddRoundKey : (State,State) -> State;
AddRoundKey(roundKey,state) = roundKey ^ state;

// Inverse Transformations
// ----------------

// InvShiftRows()
InvShiftRows : Map;
InvShiftRows(state) = [| row >>> i || row <- state || i <- [ 0 .. 3 ] |];

// InvSubBytes()
InvSubBytes : Map;
```

```
InvSubBytes(state) = [| [| (sibox @ b) || b <- row |] || row <- state |];

sibox : [256]Byte;
sibox = [| inverse(invAffine x) || x <- [0 .. 255] |];

invAffine : Byte -> Byte;
invAffine(xs) = join(mmultBit(invAffMat,split(xs ^ 0x63)));

invAffMat : {a} (a >= 8) => [8][a];
invAffMat = [ 0xa4 ] # [| x <<< 1 || x <- invAffMat || i <- [1 .. 7] |];

// InvMixColumns()
InvMixColumns : Map;
InvMixColumns(state) = mapColumns(invMixColumn,state);

// Key Expansion
// -------------

KeyExpansion : {nk} (fin nk,8 >= width nk,nk >= 1) => KeyAsWords(nk) -> RoundKeys((nk
+ 6));
KeyExpansion(keyAsWords) = [| (transpose s) || s <- ss |]
  where {
    Nk = width(keyAsWords);
    ss = groupBy(4,[| (reverse(splitBy(4,w))) || w <- take(4 * (Nk + 7),ws) |]);
    ws = keyAsWords # [| nextWord(Nk,i,w,w')
                      || i <- [Nk .. ]
                      || w <- ws
                      || w' <- drop(Nk - 1,ws)
                      |];
    nextWord : ([8],[8],Word,Word) -> Word;
    nextWord(Nk,i,w,w') = w ^ temp
    where {
       temp = if i % Nk == 0
       then SubWord(RotWord(w')) ^ Rcon(i / Nk)
       else if (Nk > 6) & (i % Nk == 4)
       then SubWord(w')
       else w';
    };
  };

SubWord : Word -> Word;
SubWord(w) = join([| (sbox @ b) || b <- split(w) |]);

RotWord : Word -> Word;
RotWord(w) = w <<< 8;

Rcon : {a} (a >= 1) => [a] -> Word;
Rcon(i) = zero # gpower(<| x |>,i - 1);


// Auxiliary Definitions
// ---------------------

applyMany : {a b} (fin a) => ([a](b -> b),b) -> b;
```

```
applyMany(fs,x) = rs ! 0
  where rs = [x] # [| f(r) || f <- fs || r <- rs |];

mapBytes : (Byte -> Byte,State) -> State;
mapBytes(f,state) = [| [| f(byte) || byte <- row |] || row <- state |];

mapColumns : (Column -> Column,State) -> State;
mapColumns(f,state) = transpose([| f(column) || column <- transpose(state) |]);

// ---------------------------------------------------------------------------

test_1 : Bit;
test_1 = ASSERT (ct == answer, “Reference spec, test 1”, True)
  where {
    ct = AES128.encrypt(0x000102030405060708090a0b0c0d0e0f,
0x00112233445566778899aabbccddeeff);
    answer = 0x69c4e0d86a7b0430d8cdb78070b4c55a;
  };

test_2 : Bit;
test_2 = ASSERT (pt == answer, “Reference spec, test 2”, True)
  where {
    pt = AES128.decrypt(0x000102030405060708090a0b0c0d0e0f,
0x69c4e0d86a7b0430d8cdb78070b4c55a);
    answer = 0x00112233445566778899aabbccddeeff;
  };

runtests = xs ! 0
  where {
    xs = [True] # [| prev & t || t <- tests || prev <- xs |];
    tests = [test_1 test_2];
  };

// ---------------------------------------------------------------------------
```

B SPIR and LLSPIR Graphs

This appendix discusses the graphs (.dot files) produced in SPIR and LLSPIR modes. It provides an explanation of the edge labels, the meaning of colors, and a glossary of components that may appear in a graph.

Edges, Nodes and Labels

A SPIR/LLSPIR graph consists of *edges* and *nodes*. Edges correspond to Cryptol values, and nodes correspond to operators and internal or hardware primitives that operate on these values.

Two numerical labels appear on each edge. The first is the width of the bit vector, and the second is the argument number of the node it is input to. The argument numbers are unnecessary for commutative operators, and oftentimes the widths are irrelevant. All unnecessary labels have been removed from the graphs in this document.

Colors and Shapes

Each node is of a certain color and shape. The following colors and shapes have meaning in SPIR and LLSPIR graphs:

- **black parallelogram**. Most clocked circuits, including delays, some internal macros (such as coercions between space and time) that are removed when translating from SPIR to LLSPIR, and components inserted during translation to LLSPIR such as Block ROMs.
- **blue text**. Basic combinatorial circuits such as switches, projects, arithmetic operators and shift operators.
- **brown parallelogram**. LoopPulse and DelayRestart (see "Common Components" on page 74).
- **orange hexagon**. Non-synthesizable time shifts, such as undelays.
- **green plaintext**. A BackEdge (see "Common Components" on page 74).

Common Components

The following components may appear in SPIR and LLSPIR graphs. There are many other primitives, such as arithmetic operators, which are self explanatory and not listed here.

- **Append**. Append two or more arguments, just like Cryptol's # operator. Append is also used to represent each tuple as a single bitvector of appended bitvectors.
- **BackEdge**. Inserted when output is fed back into a circuit as input, creating a cycle. Used by the compiler to detect illegal cycles. Has no semantic meaning and is simply removed during translation to VHDL.
- **Delay**. A clocked circuit whose output occurs one cycle after the input.
- **DelayRestart**. Upon restart, DelayRestart n outputs zeros for n cycles, followed by a high bit, followed by zeros.
- **HoldHigh**. Upon restart, HoldHigh n holds its output high for n cycles.
- **Project**. Extract some range of bits from a bitvector, or project one element from a tuple. Project n starts at bit number n.
- **LoopPulse**. LoopPulse n outputs a high bit every n cycles.
- **Restart**. A 1-bit signal used to restart parts of a circuit. The compiler inserts it automatically; it is often used to control the initial state of a circuit, such as selecting the first two values of the fibonacci sequence in Section 3.2, "Delays and Undelays," on page 24.
- **ROM**. Read-only lookup table implemented using a Block RAM. ROMs are inserted during translation to LLSPIR.
- **Switch**. A combinatorial circuit that synthesizes to a mux in hardware. The third argument is a bit that selects which of the first two arguments to output. These are elaborated from if expressions in Cryptol, or may be introduced automatically by the compiler.
- **Undelay**. A clocked circuit whose output occurs one cycle before the input. Not synthesizable.

C Sample Utility Functions

This section presents some simple utility functions, including some basic combinatorial and sequential circuits and common functions from functional programming. These utilities are defined in fpga-utils.tex and can be loaded into Cryptol like any .cry file.

FIGURE C-1 *Replace an element in a sequence (indexed from the left).*

In sequence s, replace element at index n with x

```
setL : {b c} (fin b, b >= 1) => ([width (b-1)],[b]c,c) -> [b]c;
setL (n, s, x) = (s & mask1) | mask2
  where {
    mask1 = ([zero] # ~zero) >>> n;
    mask2 = ([x] # zero) >>> n;
  };
```

FIGURE C-2 *Replace an element in a sequence (this time indexed from the right).*

Same thing, but take n relative to the most significant bit (the right-hand side)

```
setR : {b c} (fin b, b >= 1) => ([width (b-1)],[b]c,c) -> [b]c;
setR (n, s, x) = (s & mask1) | mask2
  where {
     mask1 = (~zero # [zero]) <<< n;
     mask2 = (zero # [x]) <<< n;
  };
```

FIGURE C-3 *Shift Registers*

```
shift_reg : {n x} (fin n, n >= 1) => [inf][x] -> [inf][n][x];
shift_reg input_seq = buffer where
  buffer = [undefined] #
           [| setL (0, z >> 1, x)
           || z <- buffer
           || x <- input_seq
           |];

shift_reg' : {n x} (fin n, n >= 1) => [inf][x] -> [inf][n][x];
shift_reg' input_seq = buffer where
  buffer = [undefined] #
           [| setR (0, z << 1, x)
           || z <- buffer
           || x <- input_seq
           |];
```

FIGURE C-4 *Sum inputs*

The following function outputs the sum of all previous inputs, and can be mapped across time as a sequential circuit when a *is infinite.*

```
sum : {a b} [a][b] -> [a+1][b];
sum xs = sums
     where sums = [0] # [| x + s || x <- xs || s <- sums |];
```

FIGURE C-5 *Generate a stream of alternating bits*

An infinite stream of alternating bits can be useful.

```
toggles : [inf]Bit;
toggles = [True] # [| ~toggle || toggle <- toggles |];
```

FIGURE C-6 *A map function*

map *applies a function to each element of a sequence. This definition is identical to* lift *from Section 1.2, "Using Cryptol for Hardware Design," on page 7, except the width is finite:*

```
map : {n a b} (fin n) => (a -> b, [n]a) -> [n]b;
map (f, ins) = [| f x || x <- ins |];
```

FIGURE C-7 *zip and unzip functions*

These functions convert between sequences of tuples and tuples of sequences.

```
zip : {a b c} ([a]b,[a]c) -> [a](b,c);
zip (as, bs) = [*\| (a, b) || a <- as || b <- bs |];
```

```
unzip : {a b c} [a](b,c) -> ([a]b,[a]c);
unzip as_bs = (as, bs)   where {
    as = [| a || (a, b) <- as_bs |];
    bs = [| b || (a, b) <- as_bs |];
};
```

FIGURE C-8 *A loadable register*

This register function is similar to the one defined in "Delays and Undelays" on page 24, except that the output only changes when the load *bit is set. Otherwise, the input is ignored and the output stays the same as in the previous cycle.*

```
lreg : {a} [inf](Bit, a) -> [inf]a;
lreg ins = xs where
    xs = [undefined] #   [| if load then in else prev
                         || (load, in) <- ins
                         || prev <- xs
                         |];
```

FIGURE C-9 *A counter.*

This counter produces an infinite stream of incrementing values. The count loops back to zero every 2^a *clock cycles. It can also be reset to an initial value by setting its load bit to* True*. Note that the counter is undefined until the first time it is reset to an initial value.*

```
cntr : {a} (a >= 1) => [inf](Bit, [a]) -> [inf][a];
cntr ins = xs where
    xs = [undefined] #   [| if load then init else prev+1
                         || (load, init) <- ins
                         || prev <- xs
                         |];
```

Glossary of Terms

AES	**A**dvanced **E**ncryption **S**tandard.
AST	**A**bstract **S**yntax **T**ree.
bit	Cryptol mode for interpretation of IR
Block RAM	**Block R**andom **A**ccess **M**emory. A clocked FPGA resource.
circuit size	A measure of the number of logic units on the FPGA. It includes the number of LUTs, slices, flip-flops, and Block RAMs.
clockrate	The frequency of the clock on the FPGA, measured in Hertz (Hz).
combinational	See *combinatorial circuit*.
combinatorial circuit	An unclocked circuit whose output is a pure function of its inputs.
flip-flop	A clocked FPGA resource. Also known as a 1-bit *register* or *latch*.
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
FSIM	**F**unctional **SIM**ulation. A Cryptol mode for synthesis without place and route
IR	**I**ntermediate **R**epresentation
latch	A clocked hardware component that holds the input for one cycle, also known as a *register* or a *flip-flop*.
latency	The amount of time between when an input is fed to the circuit and when a corresponding output is produced, measured in integral number of clock cycles.
LLSPIR	**L**ow **L**evel **S**ignal **P**rocessor **I**ntermediate **R**epresentation
LUT	**L**ook**U**p **T**able. A combinational FPGA resource, typically 3- or 4-bit input.
NSIM	Verilog **N**etlist **SIM**ulation. A Cryptol mode for equivalence checking after synthesis.
output rate	A measure of how often a circuit can accept input and produce output, in inverse clock cycles (e.g. one output every 3 cycles).

propagation delay	The amount of time between when an input is fed to the circuit and when a corresponding output is produced, measured in real-valued seconds (wall clock time).
register	A clocked hardware component that holds the input for one cycle. Also known as *latch* or *flip-flop.*
sequential circuit	A clocked circuit that may carry state over time
SPIR	**S**ignal **P**rocessor **I**ntermediate **R**epresentation
slice	FPGA resource that contains a fixed number of LUTs and flip-flops (2 each on many boards)
symbolic	A Cryptol mode for equivalence checking via symbolic interpretation of the IR.
throughput	The amount of information that is output from the circuit per unit of time. It is the clockrate multiplied by the width of the output and divided by the output rate of the circuit, and is measured in bits/second (bps).
TSIM	**T**iming **SIM**ulation. A Cryptol mode for synthesis with place and route.

Index