Pipelined Regular Expression Monitor Compiler Manual

February 26, 2003

1 Introduction to PREMS

1.1 Overview

The Pipelined Regular Expression Monitor Specification (PREMS) is a high-level specification style designed to facilitate the construction of interface monitors. The PREMS compiler automatically translates the interface specification into a Verilog or VHDL monitor. This manual describes the PREMS language and the usage of the compiler.

1.2 Identifiers and Reserved Words

An identifier in the PREMS language is any sequence of alphabet letters, digits, and underscores, where the first character must be an alphabet letter. Identifiers are case-insensitive in order to facilitate the translation to VHDL, which is also case-insensitive. Examples of legal identifiers are: `grant` (same as `GrAnt`), `split_2`, `data32`. Examples of illegal identifiers are: `_grant`, `2_split`, `32data`.

The following is the list of PREMS reserved words:

```
internal input output in_out define monitor
```

Reserved words cannot be used as identifiers. Their meaning will be explained in detail in the following sections.

1.3 Input Signals

The input signals of the monitor, are all the input and output signals of the block being monitored. These signals must be declared at the beginning of the file, and they can be any of the following three types:

- `input`: this signal is an input to the block being monitored.
- `output`: this signal is an output of the block being monitored.
- `in_out`: this signal is an input and an output to the block being monitored.

The actual implementation of the compiler does not make any distinction between these types but in the future we intend to use these different types to be able to blame the block responsible for causing an error.

The syntax for signal declarations is:

signal-type comma-separated-list-of-identifiers;

Arrays can be declared as *identifier[begin:end]*. A few examples of signal declaration taken from the AMBA slave specification are shown below:

```
input HTRANS[1:0], HREADY, HSEL, HMASTER[3:0];
output HRESP[1:0], HSPLIT[15:0];
```

The signals `HTRANS`, `HREADY`, `HSEL`, and `HMASTER` are the slave inputs and signals `HRESP` and `HSPLIT` are the slave outputs.

In order to access an element from an array, square brackets are used to index the specific element. For example, `HMASTER[2]` refers to the third element of array `HMASTER` which has size four.

1.4 Storage Variables

Storage variables are used in the PREMS language to facilitate the monitor specification. They can be seen as a type of memory. The main difference between a signal and a storage variable is that the later can have values assigned to it according to some conditions in the specification (see Section 1.10). Other than that, they can be used in any place a signal is expected.

The declaration of storage variables has almost the same syntax as signal declaration with the difference that they can be initialized. Initialization values are constants (see Section 1.5), and in case no initial value is set, the default value is 0 (zero). In the PREMS compiler input file, the declaration of storage variables is done in the beginning of the file together with signal declarations.

A few examples from the AMBA master specification are shown below:

```
internal i_grant = 0;
internal i_first = 1;
internal i_wdata[31:0] = 0;
```

The storage variables `i_grant` and `i_first` are used by the monitor to keep track of the grant signals and `i_wdata` is used to guarantee that the master will not change the data on the bus during a write transfer.

1.5 Constants

Constants in the PREMS language are unsigned integers represented in base ten. A few examples usage of constants are shown below:

```
internal i_count[2:0] = 0
i_count != 7
i_count <- i_count + 1
```

In the first example the constants are used for declaring the size of the array and to initialize it to 0. The second line shows a constant being used in a comparison expression (see Section 1.6.3), and in the last line the constant is used in a action expression (see Section 1.10) .

1.6 Primitive Expressions

There are three types of expressions defined in the PREMS language: *primitive expressions* (explained here), *extended regular expressions* (explained on section 1.7), and *action expressions* (explained on section 1.10).

A *primitive expression* is a formula consisting of signals, storage variables, unary operators, comparison operators and bitwise operators. The precedence of all operators are listed in Table 1 from highest precedence to lowest precedence.

Precedence	Operators
Highest	!
	== !=
Lowest	| &

Table 1: Operator precedence.

1.6.1 Signals and Storage Variables

Any monitor input signal or storage variable of size one is a primitive expression. Arrays are not considered primitive expressions but array elements are.

1.6.2 Unary Operators

The only unary operator in the PREMS language is the *negation* operator which is represented by the symbol "`!`". This operator can be used with any primitive expression and it cannot be used with arrays. For example: if `HTRANS[1:0]` is an array of size two then `!HTRANS` is illegal but `!HTRANS[0]` is legal.

1.6.3 Comparison Operators

The *equality* operator "`==`" and the *not equal* operator "`!=`" are the two comparison operators present in the language.

Only input signals, storage variables and constants can be compared. An array can be compared to another array only if both have the same size and the begin and end positions match. Arrays can also be compared to constants if their size is big enough to hold the constant. As an example, the following code shows the declaration of four arrays `A`, `B`, `C`, `D`:

```
input A[0:1], B[0:2], C[1:2], D[0:2];
```

The only valid comparison of two arrays is between `B` and `D`. Even though arrays `A` and `C` have the same size, they cannot be compared because the initial position of `A` (0) is different of initial position of `C` (1) and their end positions (1 and 2) are also different. `B` and `D` can be compared to any constant which has a value less than 8 and, `A` and `C` can be compared to any constant which has a value less than 4. In summary:
`B == C` is legal
`B != A` is illegal because of size difference
`A == C` is illegal because of initial and end positions difference
`A != 3` is legal
`A == 4` is illegal because `A` can only represent values from 0 to 3

1.6.4 Bitwise Operators

The two bitwise operators in the language are the *and* operator "`&`" and the *or* operator "`|`".

The left-hand-side and right-hand-side of the operators can be any primitive expression (arrays are not primitive expressions). The semantics for both operators is the same semantics used for logic circuits. The following is an example form the AMBA AHB slave specification:

```
!HTRANS[0] & !HTRANS[1] & HSEL & HREADY
```

The previous expression corresponds to the control signals for a slave being selected to perform an idle transfer.

1.7 Extended Regular Expressions

A *extended regular expression* is a formula containing primitive expressions and extended regular expression operators. The precedence of all operators are listed in Table 2 from highest precedence to lowest precedence.

Precedence	Operators
Highest	`+`
	`^`
	`*`
	`||`
	`,`
Lowest	`@`

Table 2: Extended regular expression operator precedence.

1.7.1 Primitive Expressions

Any primitive expression is an extended regular expression.

1.7.2 Choice

The *choice* operator "||" is used to describe two possible behaviors. At least one of the two sub-expressions must be in a matching state in order for the monitor not to generate an error.

The following is an example from the AMBA AHB slave:

```
transfer -> idle_transfer   ||
            busy_transfer   ||
            nonseq_transfer ||
            seq_transfer;
```

Each transfer can be one of the four possible types: idle, busy, non-sequential and sequential.

1.7.3 Concatenation

The *concatenation* operator "," concatenates the behavior of two sub-expressions in a time sequence. The monitor watches the left-hand-size and as soon as this sub-expression is over, it starts to watch the right-hand-side. If a mismatch occurs in any of the two sub-expressions the monitor will generate an error.

The following is an example from the AMBA AHB slave:

```
error_response -> (!HREADY & a_error) , (HREADY & a_error);
```

An error response consists of two cycles, the first one with `HREADY` low and the second one with `HREADY` high.

1.7.4 Multiple Concatenation

The *multiple concatenation* operator "^" is used to concatenate the behavior of a sub-expressions a given number of times in a time sequence. The actual implementation of the compiler expands the expression into a sequence of concatenations. For example, the following expression:

```
frame_header -> one^5, zero^3;
```

is expanded in the sequence below by the compiler:

```
frame_header -> one, one, one, one, one, zero, zero, zero;
```

1.7.5 Pipeline

The *pipeline* operator "@" makes the monitor watch the left-hand sub-expression and as soon as the parsing of this expression is done, the monitor starts two new threads, the first will watch the right-hand sub-expression and the second will watch the sub-expression that comes after the pipeline expression. Both threads must not generate an error in order for the monitor to not generate an error.

The following is an example based on the ARM AMBA slave:

```
transfer -> ((a_nonseq & HSEL_1 & HREADY) @ response)*;
```

A transfer consists of an address phase (`a_nonseq & HSEL_1 & HREADY`) followed by a pipelined response phase (`@ response`). As soon as the left-hand-side sub-expression is done, the monitor starts two new threads, the first will watch the `response` sub-expression and the second will watch the `a_nonseq & HSEL_1 & HREADY` sub-expression because of the Kleene star operator. This way we can get the expected pipelined behavior, on every cycle an address phase and a response phase take place.

1.7.6 Kleene Star

The *Kleene star* operator "`*`" is used to represent zero or more repetitions of the behavior described by the associated sub-expression.

The following is an example from the AMBA AHB slave:

```
response ->
    wait_state* ,
    (okay_response || error_response ||
     split_response || retry_response);
```

A slave response may include any number of wait states before the actual response is set.

1.7.7 One-or-more

The *one-or-more* operator "`+`" is used to represent one or more repetitions of the behavior described by the associated sub-expression. The compiler expands the one-or-more expression into a concatenation followed by a Kleene star expression. For example, the following expression:

```
write_sequence -> write+;
```

is expanded into the expression below:

```
write_sequence -> write, write*;
```

1.7.8 Extended Regular Expression Restrictions

In order to be able to automatically build a monitor from the specification, we impose a few restrictions on the extended regular expressions.

First, we require the expression contained within a Kleene star not to accept the empty string. Known constructions can normalize regular expressions to obey this restriction [1], but our implementation does not currently include this step.

Second, we forbid non-deterministic choice: we allow the choice operator, but the choices must be distinguishable within the first clock cycle. In practice, this

restriction is not a problem because protocols are typically designed to make it easy to determine immediately what action is occurring.

Finally, we allow at most one thread at a time to execute in a pipeline stage. For example, the expression `(a@(b,c))*` generates an error when the second repetition arrives at the `b` while the first repetition's pipeline sub-thread is still at the `c`. This restriction corresponds to allowing only one transaction at a time to use the hardware resources devoted to a pipeline stage.

1.8 Define Statement

The *define* statement is used to declare an abbreviation for a primitive expression. These definitions must be done after the signal and storage variable declaration section and before the *monitor* statement section. An example from the AMBA AHB slave specification is shown below:

```
define idle   = !HTRANS[0] & !HTRANS[1];
define busy   =  HTRANS[0] & !HTRANS[1];
define nonseq = !HTRANS[0] &  HTRANS[1];
define seq    =  HTRANS[0] &  HTRANS[1];
```

The identifiers on the left-hand-side of the "`=`" operator are the abbreviation for the four possible transfer types, `idle`, `busy`, `nonseq`uential, and `seq`uential.

1.9 Productions

A production is an abbreviation for an extended regular expression. Its name is an identifier that can be used in other extended regular expressions. In fact, we have already been using productions in the previous examples. In order to be able to obtain a finite state machine, recursive productions are not allowed.

The syntax for a production declaration is:

production-name $\rightarrow$ *extended-regular-expression;*

The example below shows the usage of productions in the specification of an AMBA AHB master that performs only idle transfers:

```
master_bus -> (idle || idle_transfer)*;
idle -> !i_grant;
idle_transfer ->
     (a_idle & !HREADY & i_grant)* ,
     (a_idle & HREADY & i_grant);
```

`master_bus`, `idle`, and `idle_transfer` are the production names.

The first production in the input file is the top-level production of the monitor. If the file contains the specification of more than one monitor, then the *monitor* statement must be used to indicate which production is the top-level production for each monitor. The example below shows the usage of the *monitor* statement in the AMBA AHB master spec:

```
monitor master_bus, grant;
```

The productions `master_bus` and `grant` are the top-level productions of the two monitors described in the AHB master specification.

1.10 Variable Assignment

Storage variables can have values assigned to them in an extended regular expression. The assignment is triggered when the extended regular sub-expression associated with it is matched. The following is an example from the AMBA AHB master:

```
grant -> ((HGRANT_1 & HREADY {i_grant <- 1;})  ||
          (!HGRANT_1 & HREADY {i_grant <- 0;}) ||
          (!HREADY))*;
```

The storage variable `i_grant` is assigned the value 1 every time the sub-expression `HGRANT_1 & HREADY` is matched and it is assigned the value 0 when the sub-expression `!HGRANT_1 & HREADY` is matched. If the sub-expression `!HREADY` is matched, `i_grant` remains unchanged.

The right-hand-side of an assignment is an *action expression*. The action expression can be a constant, a signal/variable, or a signal/variable array. Arithmetic addition "`+`" and subtraction "`-`" are also available.

The same storage variable may have different values assigned to it in the same cycle. The result of simultaneous assignments is implementation specific. In the current implementation, the order of assignments is defined by the parse tree of the regular expression, with assignments done in the order of a pre-order traversal of the parse tree.

1.11 Input File Format

The PREMS compiler input is a file consisting of four sections: *Signal/Variable Declaration*, *Define Declaration*, *Monitor Declaration*, and *Productions*. The order of the sections cannot be changed and the *Define Declaration* and *Monitor Declaration* parts are optional. The *Signal/Variable Declaration* section is where all input signals and storage variables are declared. The *Define Declaration* part contains the declaration of all primitive expression abbreviations. If a file describes more than one monitor than the *Monitor Declaration* statement indicates which production is the top-level production for each monitor. The last part is a list of productions that describes the behavior of the interface being specified. A more detailed explanation of each part is given in the following sections.

2 Running the PREMS Compiler

2.1 Command Line Syntax and Options

The PREMS compiler takes one file as input and it generates one file as output. The input file is the specification of the monitor and the output file is a Verilog or VHDL monitor. The command line syntax is:

premsc [options] <input-file>

where options are:

- `-t <verilog, vhdl>`: Set output language. The default value is `verilog`.
- `-o <output-file>`: Write output to `output-file`. The default value is `stdout`.

2.2 Output File

The compiler generates a Verilog file or a VHDL file as output. If the chosen language is Verilog, the file contains one module called `MONITOR`. If VHDL is the chosen language, the file contains one entity called `MONITOR` and one architecture description called `MONITOR_BEHAVIOUR`.

The input signal declaration follows the same order they were declared in the specification. In addition to these signals there are two more inputs and one output to the monitor. The first additional input is the clock signal and the second is the reset signal. The only output is the OK signal which, when high, indicates that there have not been any violations to the protocol yet. These additional signals are declared after the ones declared in the specification, and they follow the order they were presented here.

References

[1] Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *23rd International Colloquium on Automata, Languages, and Programming*, pages 336–347. Springer, 1996. Lecture Notes in Computer Science Number 1099.