fsm_perl: A Script to Generate RTL Code for State Machines and Synopsys Synthesis Scripts

Clifford E. Cummings

Sunburst Design, Inc.

ABSTRACT

Coding a Verilog RTL model of a state machine requires significant effort to generate an efficient synthesizable implementation. There are a number of different coding styles that can yield different results with varying degrees of efficiency. Because of the effort required to code a Verilog state machine, an engineer typically makes a guess as to which coding style will yield a good implementation and then rarely experiments with other styles after the first model simulates correctly.

This paper details a new and highly abbreviated language for coding a state machine and then describes the use of a Perl script called fsm_perl to turn the abbreviated code into a variety of synthesizable models for synthesis experimentation.

The fsm_perl also generates an accompanying dc_shell script to synthesize and compare the area and timing of each synthesized implementation.

1.0 **Introduction**

Coding a Finite State Machine (FSM) is not a difficult task but does involve a fair amount of typing. Efficient Verilog coding styles are well known but which FSM state-encoding style will give the best results is not obvious. The ability to easily generate different Verilog FSM designs and the accompanying synthesis scripts was the reason fsm_perl was developed. Fsm_perl is a freely available Perl script designed to make Finite State Machine (FSM) coding, experimentation and synthesis easy and efficient. Instructions on how to download fsm_perl from the Sunburst Design web site are included at the end of this paper.

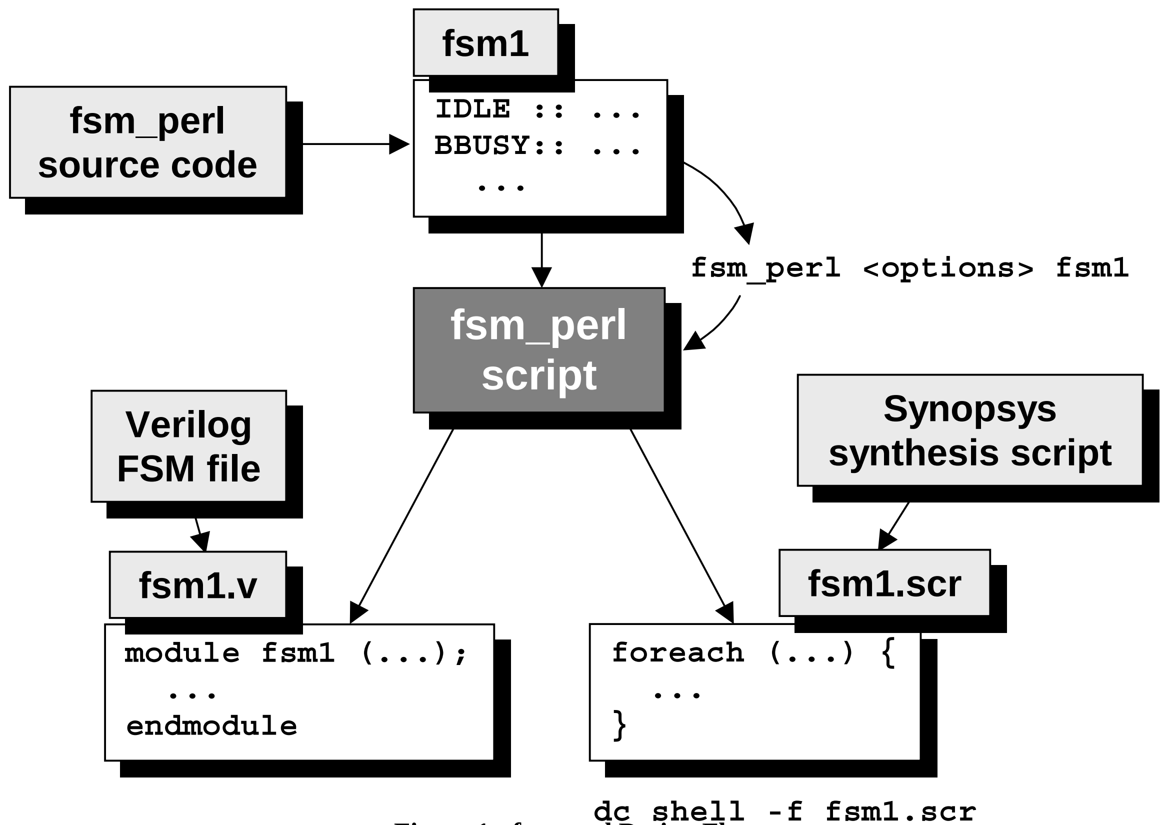


Figure 1 - fsm_perl Design Flow

Figure 1 shows the basic fsm_perl design flow. An fsm_perl source file is coded using any text editor, and then the source file is compiled using the fsm_perl command. Fsm_perl generates two files, the synthesizable Verilog source code file and a Synopsys synthesis script. The Synopsys synthesis script can then be run using dc_shell to read and compile the Verilog FSM code, produce a Verilog gate-level netlist, produce the corresponding SDF timing file and an area/timing report file.

2.0 **Basic fsm_perl Syntax**

The fsm_perl syntax was designed to make FSM coding simple, compact and easily interpreted; indeed, the fsm_perl syntax was intended to be easier to read and maintain than an equivalent Verilog source file for the same FSM. To this end, the basic fsm_perl syntax largely revolves around triplets to describe state diagram transition arcs.

In its most basic form, an fsm_perl source code file consists of state names, followed by one or more triplets consisting of input statements, next state values and Mealy- or Moore-output(s) assignments. Specifying all three fields is not required for every triplet. Legal triplet combinations are detailed in section 5.

3.0 **State Names and Encodings**

Each state in the state machine must appear as a left-side argument to a state-separator operator (a pair of adjacent colons ::). Only one adjacent colon-pair is permitted for each defined state and no white space is permitted between the colons.

State name example:

```
IDLE::  ...
READ::  ...
WAIT::  ...
DONE::  ...
```

The first state listed will be the reset-state (the state that the state machine will go to on reset).

The state names may be optionally followed by a binary state encoding enclosed within parentheses between the state name and the state-separator operator.

State name and encoding example:

```
IDLE (00)::  ...
READ (01)::  ...
WAIT (11)::  ...
DONE (10)::  ...
```

If user-defined state encodings are specified, then all of the states must specify a user-defined state encoding; otherwise, fsm_perl will report a state encoding error.

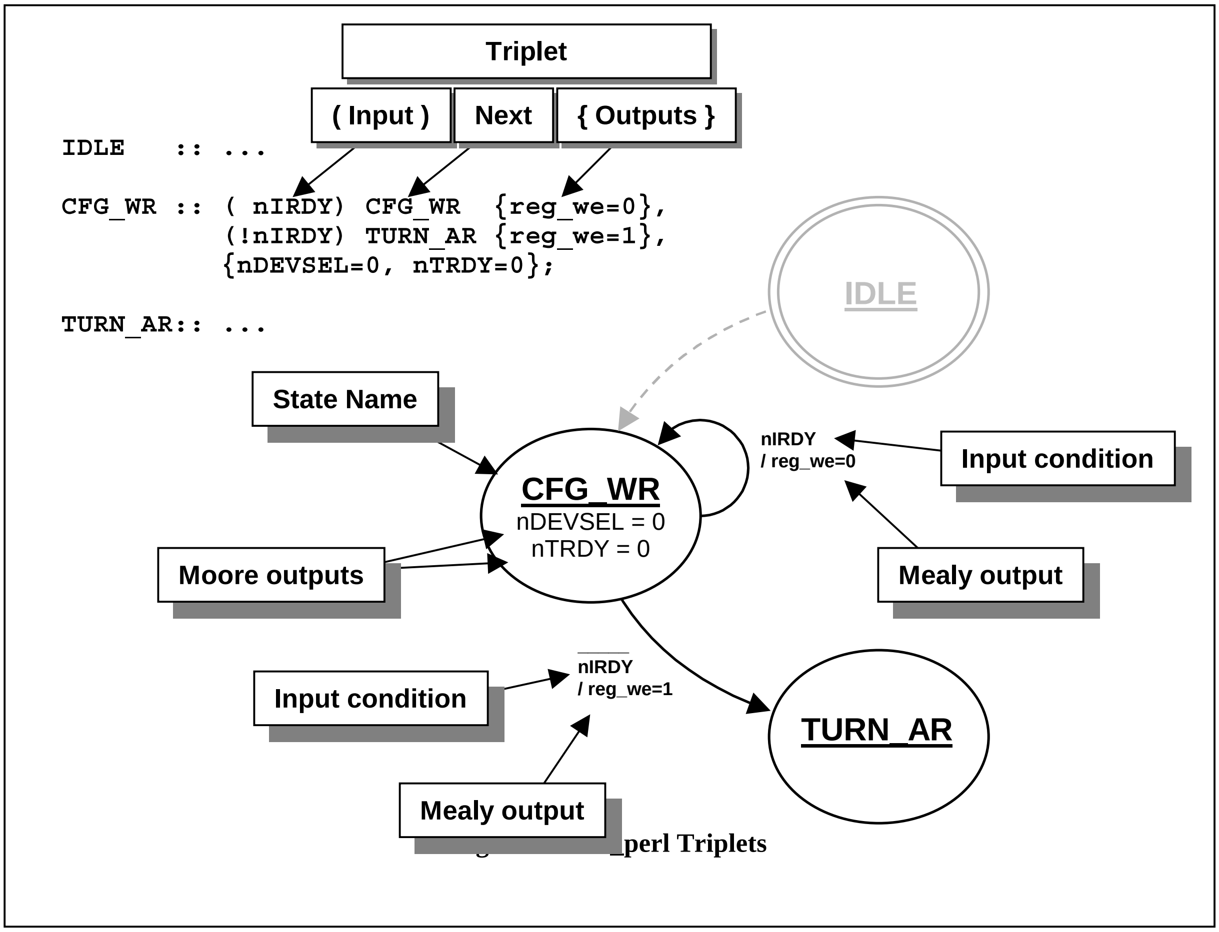
4.0 **Triplets**

Triplets are one or more comma-separated groups consisting of input statements, next state values and Mealy or Moore outputs assignments. Specifying all three fields is not required for every triplet. Legal triplet combinations are detailed in section 5. Triplets must appear as a right-side argument to a state-separator operator (::).

5.0 **Legal Triplet Statements**

5.1 **(Input)**

The input statement is enclosed in parentheses ()'s and is an expression that is used as a Boolean test. The code between the parentheses must be legal Verilog code since this expression will be copied directly into the Verilog code generated by fsm_perl. At this time, fsm_perl does no syntax checking of the Verilog expression between the parentheses; therefore, a Verilog syntax error placed in the fsm_perl source code will be written to the generated Verilog output file and will not be detected until the Verilog output file is compiled.



5.2 **Next State**

The next state statement is not enclosed in either parentheses or curly braces. The next state value must exactly match one of the state names used in the fsm_perl source code.

5.3 **{Outputs}**

Output statements are one or more Mealy outputs enclosed in curly braces {}'s or one or more Moore outputs enclosed in curly braces {}'s.

5.3.1 {Mealy Outputs}

Mealy outputs are enclosed within curly braces {}'s and must follow either an input statement, or an input statement and a next state statement. Mealy outputs are a function of the present state and one or more inputs, so if a next state is specified but an input statement is missing, inclusion of an output statement is illegal and fsm_perl issues a syntax error and halts. An output statement by itself with no input statement and no next state statement is a legal Moore output since Moore outputs are not dependent on either inputs or next state transitions.

Only one Mealy output statement is permitted for each state diagram transition-arc triplet. Multiple Mealy output definitions per transition-arc are coded as comma separated output assignments enclosed within one set of curly braces.

5.3.2 {Moore Outputs}

Moore outputs are enclosed within curly braces {}'s and are not preceded by either an input statement or a next state statement. Only one Moore output statement is permitted for each defined state. An fsm_perl syntax error is reported if more than one Moore output is detected per state definition.

5.4 Polarities and Bus Assignments

Input expressions are copied directly to the generated Verilog output code and input expressions are parsed to detect input vectors. A vector range should be included in the first input vector expression in the fsm_perl source file. Subsequent vector expressions do not require a range specification. Whenever a range specification is found in the input expression, fsm_perl tests the range to determine if a new minimum or maximum range value has been specified and updates the stored identifier range if a new minimum or maximum limit is detected.

Output expressions are parsed to determine both vector ranges (if the output is a non-scalar output) and default assignment value.

Fsm_perl has a source code directive that permits more control over vector declarations and default assignments. The directive is called "//fsm default" and is explained in more detail in section 7.4.

5.5 Comments

Fsm_perl only recognizes the Verilog single-line comment style (//). Everything in the fsm_perl source file from "//" to the end of the line is considered a comment.

6.0 **Simple Example**

fsm_perl syntax

```
IDLE :: (req) BBUSY,
        IDLE;

BBUSY:: (dly && done) BWAIT,
        (!dly && done) BFREE,
        BBUSY,
        {gnt};

BWAIT:: (!dly) BFREE,
        BWAIT,
        {gnt};

BFREE:: (req) BBUSY,
        IDLE;
```

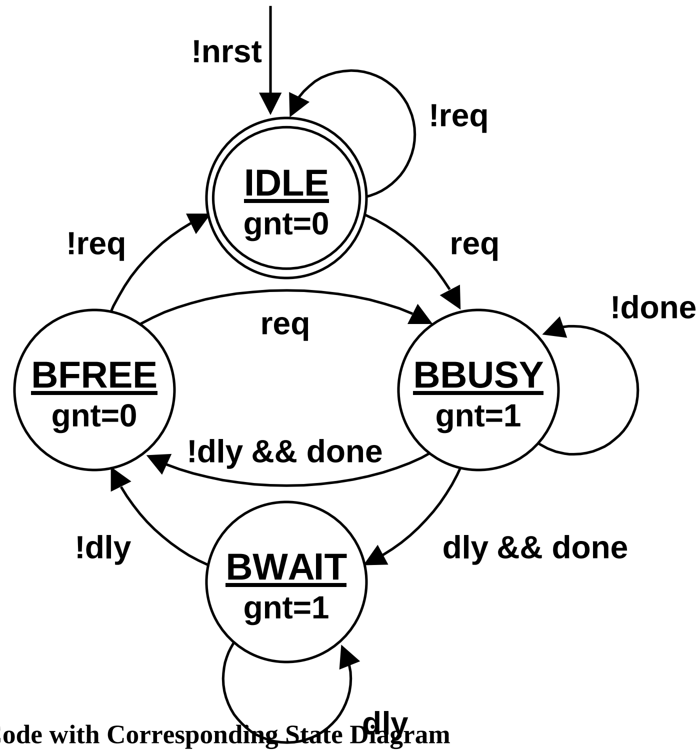


Figure 3 - fsm_perl Code with Corresponding State Diagram

Reset state is listed first

```
IDLE :: (req) BBUSY,
        IDLE;
```

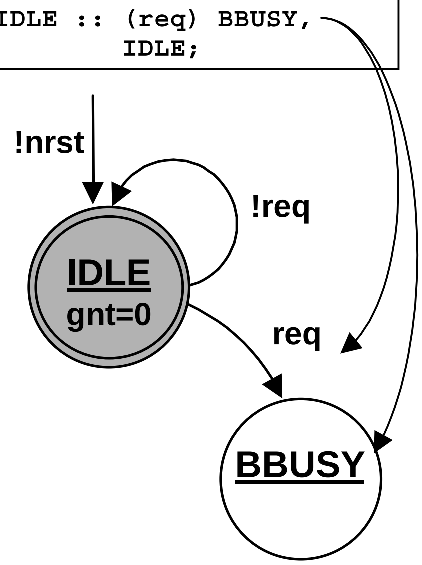


Figure 4 - IDLE State Definition

Figure 3 shows the state diagram for a simple 4-state, Moore state machine.

6.1 **IDLE State**

Figure 4 shows the fsm_perl syntax for the `IDLE` state of the state machine.

Since there are two transition arcs leaving the `IDLE` state, there will be two triplets to describe these arcs. The first triplet:

```
IDLE :: (req) BBUSY,
```

indicates that if `req` is true, a transition to the `BBUSY` state will occur. The second triplet:

```
IDLE;
```

could have been coded as `(!req) IDLE;` but for this example, a default next-state transition triplet was used, indicating that there will be a transition to the **`IDLE`** state unless **`req`** is true, in which case the transition to **`BBUSY`** will occur.

The **`IDLE`** state does not list any outputs. This means that this state will use the default reset output assignments, as determined by the other output assignments in the fsm_perl source file.

```
BBUSY:: (dly && done) BWAIT,
        (!dly && done) BFREE,
        BBUSY,
        {gnt};
```

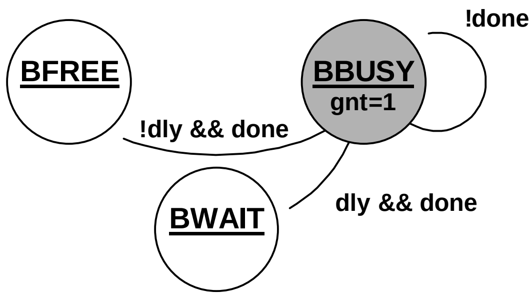


Figure 5 - BBUSY State Definition

6.2 BBUSY State

Figure 5 shows the fsm_perl syntax for the Bus BUSY (**`BBUSY`**) state of the state machine.

The **`BBUSY`** state has one Moore and no Mealy outputs. The output **`gnt`** is the first occurrence of this output in the fsm_perl code; therefore, this output is assumed to have a default-reset value opposite to the assigned value in this state. The default value for **`gnt`** will be a **`0`** after reset, and is assigned to **`1`** for the **`BBUSY`** state.

The **`BBUSY`** state also has three transition arcs leaving this state so there will be three triplets in the fsm_perl code to describe these arcs. The first triplet:

```
(dly && done) BWAIT,
```

indicates that if **`dly`** and **`done`** are both true, a transition to the **`BWAIT`** state will occur. The second triplet:

```
(!dly && done) BFREE,
```

indicates that if **`dly`** is not true and **`done`** is true, a transition to the **`BFREE`** state will occur. The third triplet:

```
BBUSY,
```

could have been coded as **`(!done) BBUSY;`** but for this example, a default next-state transition triplet was used, indicating that there will be a transition to the **`BBUSY`** state unless **`done`** is false, in which case a transition to one of the other two destination states will occur. There is one more triplet without either an input or a next state statement:

```
{gnt};
```

This triplet represents the Moore output for the **`BBUSY`** state, which was described above.

```
BWAIT:: (!dly) BFREE,
        BWAIT,
        {gnt};
```

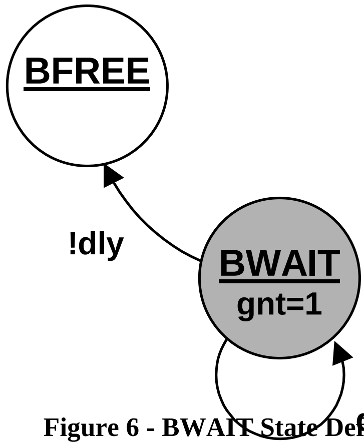


Figure 6 - BWAIT State Definition

6.3 BWAIT State

Figure 6 shows the fsm_perl syntax for the Bus WAIT (`BWAIT`) state of the state machine.

The `BWAIT` state has one Moore and no Mealy outputs. The output:

```
{gnt};
```

is assigned to `1` for the `BWAIT` state.

The `BWAIT` state also has two transition arcs leaving this state so there will be two triplets in the fsm_perl code to describe these arcs. The first triplet:

```
(!dly) BFREE,
```

indicates that if `dly` is not true, a transition to the `BFREE` state will occur. The second triplet:

```
BWAIT,
```

could have been coded as `(dly) BWAIT;` but for this example, a default next-state transition triplet was used, indicating that there will be a transition to the `BWAIT` state unless `dly` is false, in which case the transition to the `BFREE` state will occur.

```
BFREE:: (req) BBUSY,
        IDLE;
```

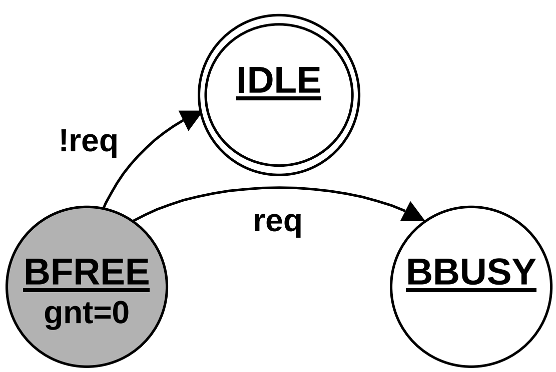


Figure 7 - BFREE State Definition

6.4 BFREE State

Figure 7 shows the fsm_perl syntax for the Bus FREE (`BFREE`) state of the state machine.

The `BFREE` state has two transition arcs leaving this state so there will be two triplets in the fsm_perl code to describe these arcs. The first triplet:

```
(req) BBUSY,
```

indicates that if `req` is true, a transition to the `BBUSY` state will occur. The second triplet:

```
IDLE;
```

could have been coded as **`(!req) IDLE;`** but for this example, a default next-state transition triplet was used, indicating that there will be a transition to the **`IDLE`** state unless **`req`** is true, in which case the transition to the **`BBUSY`** state will occur.

7.0 Fsm_perl Directives

Although fsm_perl is able to extract most of the information from the fsm_perl source code, "fsm" options can be specified to alter the default settings used by fsm_perl.

The fsm_perl user-selected defaults can be changed by adding directives (synthetic comments) to the fsm_perl source code. An fsm synthetic comment must start with "//fsm". No space is permitted between the "//" and "fsm". Legal "//fsm" commands are listed below.

7.1 //fsm clock

By default, fsm_perl generates Verilog code with clock name "clk" and "posedge" polarity. These values can be changed by adding an fsm synthetic comment to the fsm_perl source code of the form:

```
//fsm clock [negative_polarity]new_clock_name

Examples:   //fsm clock !clock      (low-true "clock")
            //fsm clock CLK         (high-true "CLK")
```

Negative edge polarities are specified by preceding the clock signal name with either "!" or "~".

7.2 //fsm period

The //fsm period directive is the only fsm_perl directive that does not affect the generated Verilog code. The "period" directive helps specify clock constraints for the generated Synopsys synthesis script. By default, fsm_perl generates a synthesis script with no clock constraints. A clock constraint can be added to the Synopsys synthesis script by adding an fsm synthetic comment to the fsm_perl source code of the form:

```
//fsm period clock_period

Examples:   //fsm period 20         ("create_clock clk -period 20")

            //fsm clock CK
            //fsm period 10         ("create_clock CK -period 10")
```

7.3 //fsm reset

By default, fsm_perl generates Verilog code with reset name "nrst" and "negedge" polarity. These values can be changed by adding an fsm synthetic comment to the fsm_perl source code of the form:

```
//fsm reset [negative_polarity]new_reset_name

Examples:   //fsm reset !rst_N      (low-true "rst_N")
            //fsm reset reset       (high-true "reset")
```

Negative edge polarities are specified by preceding the clock signal name with either "!" or "~".

7.4 //**fsm default**

Fsm_perl extracts scalar or vector information from the fsm_perl source and makes a default assignment when the outputs are assigned within the combinational always block. The default assignment value is determined by the scalar polarity or vector assignment value of each identifier that is found in the fsm_perl source code. The first time an output identifier is found, fsm_perl assumes that an assignment other than the reset-default is being made. Scalar reset-defaults are set to the opposite polarity of the first scalar assignment and vector reset-defaults outputs are assigned to all 0's. For example:

```
S1 :: S2,
      {y1};
...
```

State **S1** unconditionally transitions to state **S2** on the next clock edge and State **S1** has one scalar Moore output, **y1**, that is set to **1'b1**; therefore, fsm_perl assumes the reset-default setting for the **y1** output must have been **1'b0**. This assumption means that only when the output is assigned to a non-default value, must the output assignment be specified in the fsm_perl source code.

When an fsm_perl source file includes vector assignments and Mealy output assignments, fsm_perl might assume an incorrect reset-default output assignment value. //fsm default can be used to change the reset-default output assignment value. For the above example, if **y1** should be set to **1'b1** by default, the example could include the directive:

```
//fsm default y1=1'b1
...
S1 :: S2,
      {y1};
...
```

7.5 //**fsm state**

By default, fsm_perl generates Verilog code with the state name "state". To select a different state name, use the fsm_perl directive:

```
//fsm state new_state_name

Examples:   //fsm state PS
            //fsm state STATE
```

7.6 //**fsm next**

By default, fsm_perl generates Verilog code with the next state name "next". To select a different next state name, use the fsm_perl directive:

```
//fsm next new_next_name

Examples:   //fsm next NS
            //fsm next NEXT
```

7.7 //**fsm define**

It is sometimes desirable to make input comparisons against a macro that is defined by the Verilog `define compiler directive. When a `define comparison is used, fsm_perl must be notified of the existing definition; otherwise, fsm_perl will assume the `define identifier is a scalar input and include the identifier as both a port identifier and as a declared input. To specify the existence of a macro definition, use the fsm_perl directive:

```
//fsm define macro_name

Examples:   //fsm define `CONFIG_READ
            //fsm define `CONFIG_WRITE
Usage example
triplet:   (!nFRAME && IDSEL && (nC_BE==`CONFIG_WRITE)) CFG_WR,  ...
```

7.8 //fsm filename

By default, fsm_perl uses the fsm_perl source file name as the root of the Verilog output file name. To select a different Verilog output file name, use the fsm_perl directive:

```
//fsm filename new_file_name

Example:    //fsm filename myfile
```

7.9 //fsm module

By default, fsm_perl uses the fsm_perl source file name as the root of the Verilog module name and output file name. If there is a period in the fsm_perl source file name, unless the "//fsm module" directive is included in the fsm_perl source code, an illegal Verilog module name will be generated. To select a different Verilog module name, use the fsm_perl directive:

```
//fsm module new_module_name

Example:    //fsm module mymodule
```

In the absence of a separate "//fsm filename" directive, the "//fsm module" directive also changes the output file name. To generate unique Verilog module and output file names, use both the "//fsm module" and "//fsm filename" directives.

```
Example:    //fsm module mymodule
            //fsm filename myfile
```

8.0 Command Invocation

Fsm_perl is invoked from the UNIX command prompt as follows:

fsm_perl [options] *fsm_perl_source_file*

The fsm_perl source code, Verilog output file and Synopsys synthesis script for the **fsm1** design used in the "Simple Example" (in section 6) are shown in Figure 11 at the end of this paper.

8.1 Fsm_perl Output Files

Fsm_perl generates two output files: a Verilog source file and a Synopsys synthesis script to compile the Verilog source file. See section 9.0 for details about the generated Synopsys synthesis scripts.

8.2 **Fsm_perl Options**

Fsm_perl has command line options that help generate different Verilog coding styles and Synopsys synthesis scripts for synthesis experimentation. Each option generates just one Verilog output file and one synthesis script.

8.2.1 **-e Option**

The -e option generates a Verilog file with binary encoded state variables and standard Synopsys enumeration comments. The -e option also generates a synthesis script that will compile the Verilog design four different ways. The script compiles the design (1) with no special processing, (2) using the FSM compiler gray encoding style setting, (3) using the FSM compiler one_hot encoding style setting, and (4) using the FSM compiler binary encoding style setting.

The Synopsys enumeration comments help the Synopsys FSM compiler to find and process the state variables and state encodings.

For the following command invocation:

    `fsm_perl -e fsm1` (where **`fsm1`** is the fsm_perl source file)

fsm_perl will create two output files named:

    `fsm1_e.v` (the synthesizable Verilog FSM source file)
    `fsm1_e.scr` (the Synopsys synthesis script to compile the **`fsm1_e.v`** file).

The "_e" appendage indicates the "enumerated" coding style. The fsm_perl source code, Verilog output file and Synopsys synthesis script using the "-e" command option are shown in Figure 13 at the end of this paper.

8.2.2 **-1 Option**

The -1 (the number "one") option generates a Verilog file with one-hot encoded state variables and adds "synopsys full_case parallel_case" to the case statement. This is the only place where "full_case parallel_case" is automatically added to the Verilog source code since this coding style is the only coding style where full and parallel directives generally seem to make a positive difference in the quality of the synthesized design.

For the following command invocation:

    `fsm_perl -1 fsm1` (where **`fsm1`** is the fsm_perl source file)

fsm_perl will create two output files named:

    `fsm1_1fp.v` (the synthesizable Verilog FSM source file)
    `fsm1_1fp.scr` (the Synopsys synthesis script to compile the **`fsm1_1fp.v`** file)

The "_1fp" appendage indicates the "one-hot full_case parallel_case" coding style. The fsm_perl source code, Verilog output file and Synopsys synthesis script using the "-1" command option are shown in Figure 12 at the end of this paper.

8.2.3 **-f Option**

The -f option generates a Verilog file with "synopsys full_case" appended to the case statement header code. Using this option is not recommended since the pre-synthesis simulation might not match the post-synthesis implementation, plus there are Verilog coding styles that can accomplish the same or better synthesis optimization without using this potentially dangerous switch; however, the switch is included to permit easy experimentation with "full_case" usage.

For the following command invocation:

    `fsm_perl -f fsm1` (where **`fsm1`** is the fsm_perl source file)

fsm_perl will create two output files named:

    `fsm1_f.v` (the synthesizable Verilog FSM source file)
    `fsm1_f.scr` (the Synopsys synthesis script to compile the **`fsm1_f.v`** file)

The "_f" appendage indicates that " full_case" has been added to the Verilog output file.

8.2.4 **-p Option**

The -p option generates a Verilog file with "synopsys parallel_case" appended to the case statement header code. Using this option is not recommended since the pre-synthesis simulation might not match the post-synthesis implementation, plus there are Verilog coding styles that can accomplish the same or better synthesis optimization without using this potentially dangerous switch; however, the switch is included to permit easy experimentation with "parallel_case" usage.

For the following command invocation:

    `fsm_perl -p fsm1` (where **`fsm1`** is the fsm_perl source file)

fsm_perl will create two output files named:

    `fsm1_p.v` (the synthesizable Verilog FSM source file)
    `fsm1_p.scr` (the Synopsys synthesis script to compile the **`fsm1_p.v`** file)

The "_p" appendage indicates that " parallel_case" has been added to the Verilog output file.

8.2.5 **Multiple Options**

Multiple options can be used at the same time when fsm_perl is invoked. When multiple options are used, the resultant file names will contain appended letters indicating which file options were used to run the fsm_perl script.

9.0 Synthesis Scripts

Not only does fsm_perl generate the Verilog code for an FSM, it also generates the Synopsys synthesis script required to compile the design and report performances.

```
design_list = { fsm1 }
foreach (DESIGN, design_list) {
  rpt_file = DESIGN + ".rpt"
  echo DESIGN + " Synthesis Run" > rpt_file
  read -f verilog DESIGN + ".v"
  current_design = DESIGN
  compile
  create_schematic -size infinite
  write_timing -f sdf-v2.1 -context verilog -o DESIGN + ".sdf"
  write -f verilog -hier -output DESIGN + ".vg"
  report_area   >> rpt_file
  report_timing >> rpt_file
}
```

Figure 8 - fsm1.scr File

Figure 8 shows the synthesis script that is typically generated when using fsm_perl. The synthesis script reads the Verilog source file, compiles the design, writes out the SDF timing file, writes out the Verilog gate-level netlist, then reports area usage and worst case timing to a report file.

```
design_list = { fsm1_e }
foreach (DESIGN, design_list) {
  rpt_file = DESIGN + ".rpt"
  echo DESIGN + " Synthesis Run" > rpt_file
  read -f verilog DESIGN + ".v"
  current_design = DESIGN
  compile
...
  echo DESIGN + " Synopsys Gray-Code Synthesis Run" >> rpt_file
  extract
  set_fsm_encoding_style gray
  compile
  create_schematic -size infinite
  write_timing -f sdf-v2.1 -context verilog -o DESIGN + "_xg.sdf"
  write -f verilog -hier -output DESIGN + "_xg.vg"
...
  echo DESIGN + " Synopsys One-Hot Synthesis Run" >> rpt_file
  extract
  set_fsm_encoding_style onehot
  compile
...
  echo DESIGN + " Synopsys Binary Synthesis Run" >> rpt_file
  extract
  set_fsm_encoding_style binary
  compile
...
}
```

Figure 9 - Synopsys FSM Compiler-Options Script (fsm1_e.scr)

If the FSM file is compiled with the "-e" command option, fsm_perl will generate a synthesis script to compile the design four different ways: (1) with no special processing, (2) using the FSM compiler gray encoding style setting, (3) using the FSM compiler one_hot encoding style setting, and (4) using the FSM compiler binary encoding style setting. Figure 9 shows a section of the synthesis script that is typically generated using the "-e" option.

If the fsm_perl source file contains the "//fsm period" option, fsm_perl will generate a synthesis script to compile the design with clock constraints. Both "//fsm period" and "//fsm clock" affect the "create_clock" command that is put into the synthesis script, as shown in figure 10.

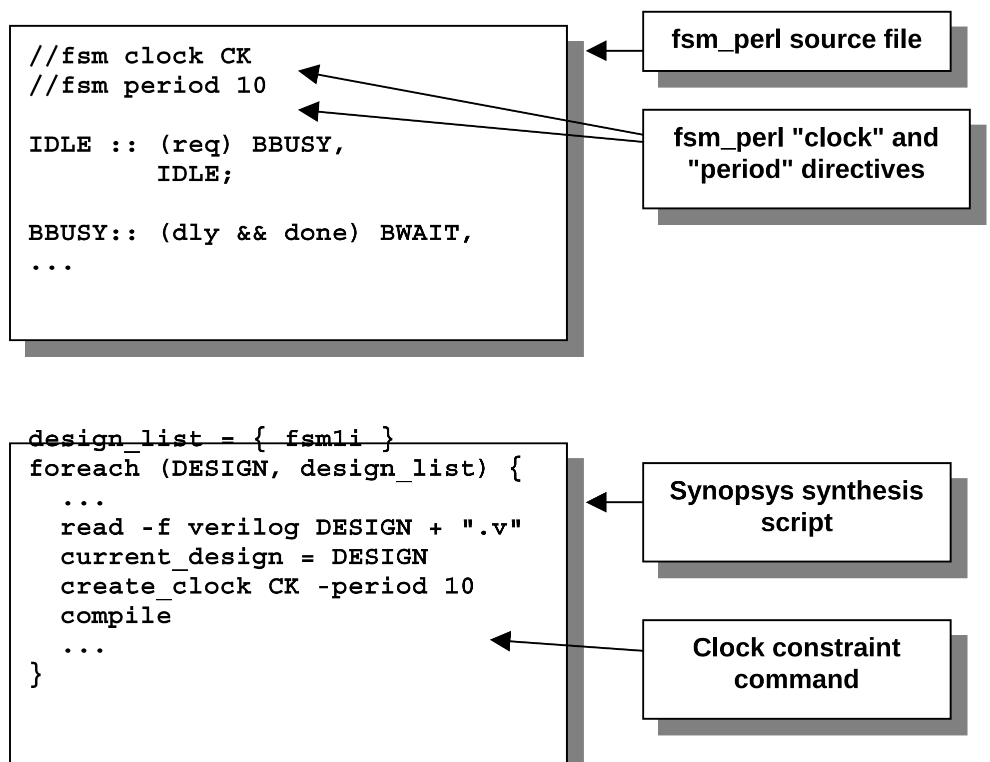


Figure 10 - Adding Clock Constraints

10.0 **Download fsm_perl**

This paper and the fsm_perl script are available for download at the Sunburst Design web site:

`www.sunburst-design.com`

The fsm_perl source code contains the GNU copyright header that permits free distribution of the fsm_perl code as long as the copyright header is included and remains unchanged.

Both fsm_perl and the paper can be freely downloaded from the Sunburst Design web site. Enhancement requests can be sent to **`cliffc@sunburst-design.com`**. The subject line should contain "**`fsm_perl enhancement request`**".

10.1 Fsm_perl Development & Enhancements

Fsm_perl was first created with the intent of simplifying the task of generating Verilog source code for simulation and synthesis. Later, other capabilities were added to generate state machines using different FSM state-encoding styles and different Verilog coding styles. The next step was to permit the creation of multiple Verilog files with different coding styles and an accompanying Synopsys synthesis script to permit easy experimentation with Verilog styles, Synopsys switches and to report the various area and timing results. The latter capability greatly accelerates the selection of an optimal coding style and synthesis strategy.

One enhancement in progress is the generation of all fsm_perl Verilog output files and a synthesis script to compile and report results from all coding styles. This enhancement will facilitate selection of the best coding style for a design project.

A one-hot output registered coding style permits the generation of non-glitching registered outputs. This is another enhancement under present consideration.

With the release of the Synopsys 1999.05 TCL interface, generation of a TCL script output file might also be a valuable future enhancement.

Another potential enhancement would be the generation of synthesizable VHDL code from fsm_perl source code. This should require little more than an alternate code generator.

As the art of FSM design using HDLs progresses, it is anticipated that additional styles will be generated by fsm_perl.

11.0 Conclusion

The fsm_perl syntax is short, simple to generate and easy to understand. Fsm_perl removes much of the tedious effort associated with the entry of synthesizable Verilog FSM code, plus fsm_perl generates the Synopsys script that is necessary to compile and examine the synthesized FSM design.

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 19 years of ASIC, FPGA and system design experience and nine years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, chaired the VSG Behavioral Task Force, which was charged with proposing enhancements to the Verilog

language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

E-mail Address: cliffc@sunburst-design.com
This paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of September 7th, 2001)

fsm1 - fsm_perl source

```
IDLE :: (req) BBUSY,
        IDLE;

BBUSY:: (dly && done) BWAIT,
        (!dly && done) BFREE
        BBUSY,
        {gnt};

BWAIT:: (
        B
        {

BFREE:: (
        I
```

fsm_perl compile command:
`fsm_perl fsm1`

fsm1.v - Verilog output

```
module fsm1 (gnt, dly, done, req, clk, nrst);
  output gnt;
  input  dly, done, req;
  input  clk, nrst;
  reg    gnt;

  parameter [1:0] IDLE = 2'd0,
                  BBUSY = 2'd1,
                  BWAIT = 2'd2,
                  BFREE = 2'd3;

  reg [1:0] state, next;

  always @(posedge clk or negedge nrst)
    if (!nrst) state <= IDLE;
    else state <= next;

  always @(state or dly or done or req) begin
    next = 2'bx;
    gnt = 1'b0;
    case (state)
      IDLE: begin
          next = IDLE;
          if (req) next = BBUSY;
        end

      BBUSY: begin
          next = BBUSY;
          if (dly && done) next = BWAIT;
          if (!dly && done) next = BFREE;
          gnt = 1'b1;
        end

      BWAIT: begin
          next = BWAIT;
          if (!dly) next = BFREE;
          gnt = 1'b1;
        end

      BFREE: begin
          next = IDLE;
          if (req) next = BBUSY;
        end

    endcase
  end
endmodule
```

fsm1.scr - Synopsys script

```
design_list = { fsm1 }
foreach (DESIGN, design_list) {
  rpt_file = DESIGN + ".rpt"
  echo DESIGN + " Synthesis Run" >
rpt_file
  read -f verilog DESIGN + ".v"
  current_design = DESIGN
  compile
  create_schematic -size infinite
  write_timing -f sdf-v2.1 -context
verilog -o DESIGN + ".sdf"
  write -f verilog -hier -output DESIGN +
".vg"
  report_area   >> rpt_file
  report_timing >> rpt_file
}
```

Figure 11 - fsm_perl Source, Verilog File, Synthesis Script

fsm1 - fsm_perl source

```
IDLE :: (req) BBUSY,
        IDLE;

BBUSY:: (dly && done) BWAIT,
        (!dly && done) BFREE
        BBUSY,
        {gnt};

BWAIT:: (
        B
        {

BFREE:: (
        I
```

fsm_perl compile command:
`fsm_perl -1 fsm1`

fsm1_1fp.v - Verilog output

```
module fsm1_1fp (gnt, dly, done, req, clk, nrst);
  output gnt;
  input  dly, done, req;
  input  clk, nrst;
  reg    gnt;

  parameter [3:0] IDLE = 4'd0,
                  BBUSY = 4'd1,
                  BWAIT = 4'd2,
                  BFREE = 4'd3;

  reg [3:0] state, next;

  always @(posedge clk or negedge nrst)
    if (!nrst) begin
      state <= 4'b0;
      state[IDLE] <= 1'b1;
    end
    else state <= next;

  always @(state or dly or done or req) begin
    next = 4'b0;
    gnt = 1'b0;
    case (1'b1)  // synopsys full_case parallel_case
      state[IDLE]: begin
          next[IDLE] = 1'b1;
          if (req) next[BBUSY] = 1'b1;
        end
      state[BBUSY]: begin
          next[BBUSY] = 1'b1;
          if (dly && done) next[BWAIT] = 1'b1;
          if (!dly && done) next[BFREE] = 1'b1;
          gnt = 1'b1;
        end
      state[BWAIT]: begin
          next[BWAIT] = 1'b1;
          if (!dly) next[BFREE] = 1
          gnt = 1'b1;
        end
      state[BFREE]: begin
          next[IDLE] = 1'b1;
          if (req) next[BBUSY] = 1
        end

    endcase
  end
endmodule
```

fsm1_1fp.scr - Synopsys

```
design_list = { fsm1_1fp }
foreach (DESIGN, design_list) {
  rpt_file = DESIGN + ".rpt"
  echo DESIGN + " Synthesis Run" >
rpt_file
  read -f verilog DESIGN + ".v"
  current_design = DESIGN
  compile
  create_schematic -size infinite
  write_timing -f sdf-v2.1 -context
verilog -o DESIGN + ".sdf"
  write -f verilog -hier -output DESIGN +
".vg"
  report_area   >> rpt_file
  report_timing >> rpt_file
}
```

Figure 12 - fsm_perl Source, One-Hot Verilog File, Script

fsm1i - fsm_perl source

```
//fsm clock CK
//fsm period 10

IDLE :: (req) BBUSY,
        IDLE;

BBUSY:: (dly && done) BWAIT,

BWAIT::
```

fsm_perl compile command:
`fsm_perl -e fsm1i`

fsm1i_e.v - Verilog output

```
module fsm1i_e (gnt, dly, done, req, CK, nrst);
  output gnt;
  input  dly, done, req;
  input  CK, nrst;
  reg    gnt;

  parameter [1:0] // synopsys enum code
                IDLE = 2'd0,
                BBUSY = 2'd1,
                BWAIT = 2'd2,
                BFREE = 2'd3;

  // synopsys state_vector state
  reg [1:0] // synopsys enum code
                state, next;

  always @(posedge CK or negedge nrst)
    if (!nrst) state <= IDLE;
    else state <= next;

  always @(state or dly or done or req) begin
    next = 2'bx;
    gnt = 1'b0;
    case (state)
      IDLE: begin
          next = IDLE;
          if (req) next = BBUSY;
        end
      BBUSY: begin
          next = BBUSY;
          if (dly && done) next = BWAIT;
          if (!dly && done) next = BFREE;
          gnt = 1'b1;
        end
      BWAIT: begin
          next = BWAIT;
          if (!dly) next = BFREE;
          gnt = 1'b1;
        end
      BFREE: begin
          next = IDLE;
          if (req) next = BBUSY;
        end
    endcase
  end
endmodule
```

fsm1i_e.scr - Synopsys script

```
design_list = { fsm1i_e }
foreach (DESIGN, design_list) {
...
  read -f verilog DESIGN + ".v"
  current_design = DESIGN
  create_clock CK -period 10
  compile
...
  extract
  set_fsm_encoding_style gray
  compile
...
  extract
  set_fsm_encoding_style onehot
  compile
...
  extract
  set_fsm_encoding_style binary
  compile
...
}
```

Figure 13 - fsm_perl Source with Clock & Period Directives, Verilog File, 4-Pass-Compile Synthesis Script