Esencia

PLP v2 and SmGen

Rising the level of abstraction in Verilog design

6/11/2010

Esencia Technologies Inc.

www.esenciatech.com

Agenda

- Motivation
- **PLP v2** (PerL Preprocessor)
  - A language agnostic pre-processor that uses Perl to generate output code
- **SmGen** (State Machine generator)
  - Behavioral style Verilog conversion to synthesizable FSM
- A combined example
- Licensing
- Support

© Esencia Technologies Inc.

Motivation

- Verilog design is still low level compared to languages like C++
- Synthesis tools constraint coding to a set of synthesizable templates
- Some of them addressed by SystemVerilog but still can be improved
- People is not willing to embrace improvements because of the risk of back-end tools choking with the flow later on in the project

© Esencia Technologies Inc.

Examples of limitations

- Events need to appear at the beginning of always blocks: E.g.

```
always @(posedge clk or negedge rst_n) begin
  // event free sequential block
end

always @(a or b or c) begin
  // event free combinational block
end
```

Examples of limitations

- Many times a more natural alternative to a FSM is to write code sequentially as in the original algorithm

```
mul1 <= 'habc;
mul2 <= 1234;
do_mul <= 1;
@(posedge clk);
div1 < mul_res;
div2 <= 5678;
@(posedge clk);
res <= div_res;
```

But this is still cumbersome...

© Esencia Technologies Inc.

Examples of limitations

- Ideally one would write code as in the original algorithm and insert clock events as needed to distribute the load across clock cycles

```
Mul('habc, 1234, mul_res);
Div(mul_res, 5678, div_res);
```

- But “time consuming tasks” are not synthesizable by standard tools. Currently this needs to be implemented as a Finite State Machine (FSM)

© Esencia Technologies Inc.

Examples of limitations

- Other Verilog limitations are easily addressed by a preprocessor:
  - Generating multiple instantiations of a block based on a parameter
  - Extra constant / compile time functions
    - log2(x) for number of bits required to hold x
    - Stringify a parameter to make it readable on a waveform viewer
    - sin(x)/cos(x)/sqrt(x)... over constant values for table generation
    - min/max(x,y[,..])
  - Macro expansion:

    WiggleWire(a,1,0,1)

© Esencia Technologies Inc.

PLP – a simple perl based preprocessor

- Why Perl ?
  - Powerful / ubiquitous in ASIC design environments
  - Excellent text processing capabilities
  - Most designers familiar with it

© Esencia Technologies Inc.

PLP – The basic idea

- Phase 1: The input file is converted into a Intermediate Perl Script (or IPS in what follows)
  - By default if a line contains:

    aaa bb cc
  - The generated code just prints it as is:

    print “aaa bb cc\n”;
- Perl special characters are automatically quoted:

  input: $display($time, “ reset on\n”);

  IPS: print “\$display(\$time, \” reset on\\n\”);

© *Esencia Technologies Inc.*

PLP – inserting Perl code

- Lines starring with % (PLP's Perl scape character – re-definable with -ps option) are emitted to IPS as is:

input:
```
%for $i (1..3) {
hi
%}
```

IPS:
```
for $i (1..3) {
print “hi\n”;
}
```

Redefining Perl escape character may be interesting for other applications (e.g. // Pragma to expand pragmas given in comments). Defined by a regular expression.

© *Esencia Technologies Inc.*

PLP – output generation

- Phase 2: IPS is executed by the Perl interpreter and its output generates the output file

```
perl IPS > output
output:   hi
          hi
          hi
```

- If IPS contains errors, it can be easily debugged as it is visible by the user:
  - By inspection for simple syntax errors
  - With Perl debugger (perl -d IPS) or with ddd

© Esencia Technologies Inc.

PLP – value interpolation

- Perl variable values can be inserted in regular text by using ${varname} syntax. E.g.

input:
```
%for $i (1..3) {
hi $i is ${i}
%}
```

IPS:
```
for $i (1..3) {
print “hi \$i is ${i}\n”;
}
```

output:
```
hi $i is 1
hi $i is 2
hi $i is 3
```

© Esencia Technologies Inc.

PLP – expression interpolation

- A Perl function call or expression can also be called directly in regular text. Use $((expr)) syntax; the return value of the expression is interpolated in the output text. Ex.

input:

```
%$max=256;
%for $i (1..3) {
input [ $((log2($max)-1)) : 0 ] x${i};
%}
```

output:

```
input [ 7 : 0 ] x0;
input [ 7 : 0 ] x1;
input [ 7 : 0 ] x2;
```

© Esencia Technologies Inc.

Esencia

PLP – including raw Perl code

- A set of Perl subroutines can be included for later use as follows

  input: %include("util.pl");

- Note that include() is just a built-in Perl sub contained in PLP itself that evals the code in util.pl

© *Esencia Technologies Inc.*

PLP – invocation

- If the extension of the filename is .plp, by default the output filename is constructed from the input one by dropping the .plp extension

  > plp fifo.v.plp (generates fifo.v)

- You can also explicitly define output filename with -o option

  > plp pre_fifo.v -o fifo.v

© *Esencia Technologies Inc.*

PLP – invocation (2)

- To pass a parameter for generation from the shell, invoke plp with **-p param=val** as many times as required

  > plp -p width=32 -p depth=4 pre_fifo.v -o fifo_w32_d4.v

  Prepends: $width=32;
  $depth=4;

To IPS to be used in the pre_fifo.v code so that the code can be generated according to those parameters

© *Esencia Technologies Inc.*

PLP – invocation (3)

- Sometimes a parametric file may want to generate the output filename programmatically within the body of the input file. E.g. parameters are width and depth and filename must have width and total_bits as part of the filename.
- Use embedded PLP_FILENAME directive

> plp -p width=32 -p depth=4 pre_fifo.v

```
% $bits = $width * $depth
// PLP_FILENAME=”fifo_w${width}_bits${bits}.v”
```

Would generate fifo_w32_bits128.v

© *Esencia Technologies Inc.*

PLP – 2 passes

- Pass 1 the pure Perl generation we have mentioned so far (with 2 phases)
- Pass 2 is Verilog specific so it is disabled by default (-2 to enable it). It runs the code through emacs in batch mode for /*AUTO...*/ directive expansion by using emacs Verilog mode. See

  http://www.veripool.org/wiki/verilog-mode

  For details on Verilog mode AUTO directives

PLP – command line options

Perl Based Preprocessor

USAGE: plp [options] filename.ext ...

-h : Display this message
-q : Quiet mode
-n : No comment. Remove initial comment on generated file
-ips f : Use f as filename for Intermediate Perl Script (def plp_tmp*.pl)
-c : Compile only (don't execute fileTmp)
-[no]1 : Perl preprocessor step (def yes)
-[no]2 : auto/emacs pass (def no)

© *Esencia Technologies Inc.*

PLP – command line options (2)

-pp str : Perl parameters. Will add "parameters" to 1st line of macrop_tmp*.pl (e.g -w for #!/usr/bin/perl -w)

-o f : Output filename (requires only one input file name given) when multiple files are given the filename is expected to have .plp extension which is removed to generate the output filename

-d dir : Destination directory for resulting file (def .)

-cs str : Specify comment start sequence (def '//')

-ce str : Specify comment end sequence (def '')

-ps str : Specify escape character for perl (def '%')

-ms str : Specify start escape char for phase 1 calls (def '@')

-ts str : Specify escape character for phase 2 calls (def '')

-es str : Specify start escape char for Perl eval (def '$((')

-ee str : Specify end escape char for Perl eval (def '))')

© *Esencia Technologies Inc.*

PLP – command line options (3)

-p var=value : pass a parameter to file to process
(e.g. -p WIDTH=32) multiple can be given with
several -p parameters

An Intermediate Perl Script (fileTmp) will be created.
The execution of that file generates the post-processed file on stdout.
It can be used to debug the code embedded in the pre-preprocessed file

if the string PLP_FILENAME="filename" is found in the generated output after a comment (as per -cs option) then the output filename is overridden by this value (this allows to compute the filename within the body of the file based on command line parameters)

© *Esencia Technologies Inc.*

PLP – includes

- Sometimes is convenient to 'execute' a Perl function while PLP parses your input file in phase 1, instead of just emitting its code to IPS. For example to include a plp file to be processed

  @plp_include(my_plp_lib_file)

Will process &plp_include("my_plp_lib_file") function during the IPS generation (phase 1). plp_include is a built-in function but the same would happen with user defined Perl sub's
NOTE that this includes plp code (not Perl code)

© Esencia Technologies Inc.

Esencia

PLP – start-up

- You may want to preload a set of PLP files for a given file type.
- PLP automatically evals (and makes available to phase 1) the following Perl files in this order

  plp_path/**plp_begin.pl**

  plp_path/{file_type}/**plp_begin.pl**

Where file_type is derived as follows:

If (plp_path/file_extension exists) { file_type = file_extension }

else if (file_type in c/h } {file_type = c }

else if (file_type in c/cc/cxx/cpp/hpp/C/H } {file_type = cpp }

else if (file_type in v/vh } {file_type = v }

else { file_type = file_extension }

© Esencia Technologies Inc.

PLP – start-up (2)

- You may want to preload a set of PLP files for a given file type and make the available to phase 2
- PLP automatically includes (copies verbatim to IPS) the following Perl files in this order

  plp_path/**plp_lib.pl**

  plp_path/{file_type}/**plp_lib.pl**

- For instance functions like log2/sign_extend etc. are interesting under v/plp_lib.pl so that they become code generators for all Verilog files

© *Esencia Technologies Inc.*

PLP – finishing-up

- PLP automatically evals the following Perl files in this order

  `plp_path/{file_type}/`**`plp_end.pl`**

  plp_path/**plp_end.pl**

- This allows you to emit code you may have captured in variables and purposely delayed towards the end of the processing

© *Esencia Technologies Inc.*

Esencia

PLP – advanced features

- Inserting @func(param1, param2) causes PLP to invoke &func(“param1”, “param2”) during IPS generation (phase 1) and emits to IPS whatever &func **returns**. This can be used to emit complex Perl sequences to IPS on the fly (like generating a subroutine declaration with a specific template)

- &func must have been defined in Perl elsewhere (for instance in one of the plp_begin.pl automatically included)

© *Esencia Technologies Inc.*

PLP – advanced features (2)

- For example see MacroDef / MacroEnd implementation in plp_begin.pl (auto-loaded on start-up for Perl code) :

```
sub MacroDef {
  my ($name, @pars) = @_;
  local $"=",\$";
  my $res = "sub $name {\n";
  if ($#pars >= 0) {
    $res .= "my (\$@pars) = \@_;\n";
  }
  return $res;
}

sub MacroEnd {
  return "}\n";
}
```

PLP – advanced features (3)

- ... make the following two definitions equivalent:

```
%sub macro_min1 {
% my ($x1, $x2, $res) = @_;
  if (${x1} < ${x2})
    ${res} = ${x1};
  else
    ${res} = ${x2};
%}

@MacroDef(macro_min1, x1, x2, res);
  if (${x1} < ${x2})
    ${res} = ${x1};
  else
    ${res} = ${x2};
@MacroEnd;
```

© Esencia Technologies Inc.

PLP – advanced features (4)

- In order to clean-up the syntax, PLP allows function calls of the type

```
%  &func("par1", "par2", ..., "parn");
```

- To be entered as

[\s*] [ts] func(par1, par2, ..., parn) [;]

Where ts is an optional start symbol (empty by default, see –ts option)

E.g. `ProcCall(t1, a1, a2);`

© Esencia Technologies Inc.

- For example:

```
%sub outReg {
% my ($name, $w) = @_;
% if (defined($w) && $w != 0) {
      output [${w} – 1: 0 ] ${name} ;
      reg [${w} – 1: 0 ] ${name} ;
% }else {
      output ${name} ;
      reg ${name} ;
% }
%}
```

- Allows you to do anywhere in the code:

```
input go;
outReg(done);  // generates output and reg decl
outReg sum, 10 // parenthesis / ; are optional
```

© Esencia Technologies Inc.

Esencia

PLP – summary

- Brings all the text processing capabilities of Perl to your design cycle
- Encourages reuse and brevity in the code. Per language libraries being developed

```
plp/c/*
plp/v/*
```

- Targets clean syntax so code can look close to the original language
- The intent is to allow you to easily augment your original language in a simple way
- Check-out examples included in the distribution of further use

```
plp/examples/*
```

© Esencia Technologies Inc.

Esencia

SmGen – State Machine Generation

- Translates sequential code into FSM's
- Complex FSMs are still required when full pipelining is an overkill in many designs
- Typical Flow:
  - Write behavioral code within Smg.. blocks
  - Pre-process with PLP if needed (assumed here)
  - Generate output through smgen choosing
    - Behavioral output (basically same as input but with thin wrapper code)
    - FSM 1-block style (synthesizable)
    - FMS 2-block style (synthesizable)
- Process with PLP one last time as smgen may generate directives that need PLP (auto expansion)

© Esencia Technologies Inc.

SmGen – input structure

SmgBegin
  flop_declaration (flop_declaration)*
[**SmgCombo**
  combo_declaration (combo_declaration)*]
SmgForever
…
SmgEnd

flop_declaration := [**local**] [**reg**] [width_declaration] var_name [**<=** reset_value] **;**

combo_declaration := [**local**] [**reg**] [width_declaration] var_name [**=** init_value] ;

width_declaration := <empty> | **[** integer_expr **:** integer_expr **]**

© Esencia Technologies Inc.

SmGen – example

```
SmgBegin
  reg [31:0] x <= 1'b1;
  reg [7:0] cnt <= 4'b0;
SmgForever
  while (cnt != 4'b1111) begin // wait a number of clocks
    cnt <= cnt + 1'b1;
    `tick;
  end
  `tick;
  while (~ack) `tick;  // wait for ack to arrive
  x <= 0; // drive a signal
  while (cnt != 4'b0000) begin // wait some more clocks
    cnt <= cnt + -1'b1;
    `tick;
  end
  `tick;
  while (~ack) `tick; // wait for another ack
  x <= 1;
SmgEnd
```

© Esencia Technologies Inc.

SmGen – example notes

- Flop declaration section defines reset value and which entities have its output registered
- Clock name/polarity, reset name/polarity are command line options to SmGen
- **`tick** represents a clock event but allows abstracting clock name/polarity at this level. It also implies “go back to the reset condition if reset is asserted”
- SmgForever block can use sequential code. This statement inserts an infinite loop around your code:

```
while(1) begin
   `tick
   .. code between SmgForever/SmgEnd here...
end
```

© Esencia Technologies Inc.

SmGen – invocation

- Behavioral output

  > smgen sample.vb -beh > sample.v

- 1-block FSM style (use this one by default)

  > smgen sample.vb > sample.v

- 2-block FSM style (more flexible control)

  > smgen sample.vb -sep > sample.v

- We'll use .vb in the examples for code containing this type of Behavioral Verilog

© Esencia Technologies Inc.

SmGen – invocation

State Machine generator

Usage: smgen [options] input_file > output_file

Where options is any combination of the following:

-[no]sync	Specifies sync reset vs. asynchronous (default async)
-[no]high	Specifies active high reset (default low)
-[no]fall	Specifies falling edge clk as active (def rising)
-beh	Output is behavioral (default is RTL 1-block FSM)
-sep	if !beh, Output is RTL 2-block FSM style
-help	Display this message

© *Esencia Technologies Inc.*

SmGen – command line options (2)

Following options require an extra parameter
(s=string, n=integer number)

-prefix s	Prefix for state names (def ST)
-clk s	Clock name (def clk)
-rst s	Reset name (def rst_n)
-name s	Used to derive generated block name etc. (def behav)
-state s	Name of state variable generated (def state)
-dbg n	Set debug level (def 0)

© *Esencia Technologies Inc.*

SmGen – Example of invocation

- 1-block FSM output, synchronous reset active high

```
> smgen sample.vb –high –sync > sample.v
```

- 2-block FSM with explicit reset/clock names

```
> smgen sample.vb -sep –clk clock –reset resetN > sample.v
```

© Esencia Technologies Inc.

SmGen – Example - Arbiter

See http://www.asic-world.com/tidbits/verilog_fsm.html for full blown FSM verilog code and more thorough description. This is the SmGen version:

```
 1: //=================================================
 2: // This is FSM generation demo using SmGen
 3: // File Name   : arb.vb.plp
 4: //=================================================
 5: module fsm_using_smgen (/*AUTOARG*/);
 6:
 7: //=============Input Ports=============================
 8: input   clock,reset,req_0,req_1;
 9:  //=============Output Ports============================
10: output  gnt_0,gnt_1;
11:
12: @MacroDef(expect, expr);
13:      `tick; while (! (${expr}) ) `tick;
14: @MacroEnd;
```

© Esencia Technologies Inc.

SmGen – Example - Arbiter

```
15: SmgBegin
16:   reg gnt_0 <= 0;
17:   reg gnt_1 <= 0;
18: SmgForever
19:    if (req_0 == 1'b1) begin
20:       gnt_0 <= 1;
21:       expect(req_0 == 1'b0);
22:       gnt_0 <= 0;
23:    end else if (req_1 == 1'b1) begin
24:       gnt_1 <= 1;
25:       expect(req_1 == 1'b0);
26:       gnt_1 <= 0;
27:    end
28: SmgEnd
29:
30: endmodule // End of Module arbiter
```

© Esencia Technologies Inc.

SmGen – Example2 – Motor controller

- See

http://www.cse.nd.edu/courses/cse20221/www/handouts/L17_FSM%20Design%20Example%20with%20Verilog.pdf

for a detailed description of the problem and full blown FSM Verilog code solution

NAME	TYPE	FUNCTION
activate	input	starts the door to go up/down or stops the motion
up_limit	input	indicates maximum upward travel
dn_limit	input	indicates maximum downward travel
motor_up	output	Causes motor to run in direction to raise the door
motor_dn	output	Causes motor to run in direction to lower door
reset	input	Force the controller to enter into the initial state

© Esencia Technologies Inc.

SmGen – Example2 – Motor controller

SmGen version:

```
 1: module DoorOpener(/*AUTOARG*/);
 2: input clk, activate, up_limint, dn_limit, reset;
 3: output motor_up, motor_dn;
 4:
 5: @MacroDef(expect, expr);
 6:    `tick; while ( !(${expr}) ) `tick;
 7: @MacroEnd;
 8:
 9: SmgBegin
10:    reg motor_up <= 0;
11:    reg motor_dn <= 0;
12: SmgForever
13:    if (up_limit) begin
14:       expect(activate);
15:       motor_dn <= 1;
16:       expect(dn_limit);
17:       motor_dn <= 0;
18:    end
19:    else begin
20:       expect(acivate);
21:       motor_up <= 1;
22:       expect(up_limit);
23:       motor_up <= 0;
24:    end
25: SmgEnd
26: endmodule
```

© Esencia Technologies Inc.

Esencia

SmGen – Example2 – Motor controller

- Note `expect()` PLP macro is so usual that deserves a place in v/plp_lib.pl to be automatically available
- Code is much more concise (26 vs. 73 lines)
- Complex FSM's become a piece of cake!

© Esencia Technologies Inc.

Esencia

SmGen – Summary

- FSMs are too low level and error prone
- More code means more chances for bugs
- SmGen code is much more concise (2.5-3x)
- More readable and natural once you get used to this type of representation.
- Closer to the original algorithm and less error prone
- Check-out more eamples under:

  smgen/examples/*

© Esencia Technologies Inc.

Esencia

Licensing and support

- LGPL licensing
  - Your HW is yours
  - Your SW is yours
  - Your extension libraries are yours, but we encourage you to share
  - If you change the tools themselves, changes should be made available to others
- estool@esenciatech.com for questions/bugs

© *Esencia Technologies Inc.*