CHAPTER 1 CSL Register

TABLE 1.1 Chapter Outline

1.1	CSL Register Syntax and Command Summary
1.2	CSL Register Commands

1.1 CSL Register Syntax and Command Summary

1.1.1 CSL Register class

Registers are state elements that can perform various tasks (store data, increment values, shifting, etc.). CSL includes a set of built-in register classes that ease designs which use this type of state element. Support is included for counter and D type registers. This class allows full configuration of the RTL register code.

1.1.2 CSL Register class declaration

A CSL Register can only be declared in the global scope just like any other CSL class. The CSL Register class is declared as in the below example:

```
csl_register register_name {
  //no objects may be instantiated in a CSL register
  register_name(){
    (register methods calls)+
  }
};
```

Note for example above: **bold** text represents language reserved syntax, *italics* are user defined variables, green commented text details language notes and blue text is a short BNF representation of CSL commands/declarations.

In the register class' scope there aren't any objects to be specifically declared or instantiated as shown in Table 1.2

TABLE 1.2 Rules for instantiating objects in the register's scope

CSL class	Is instantiated in CSL Register scope
CSL Unit	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-

TABLE 1.2 Rules for instantiating objects in the register's scope

CSL class	Is instantiated in CSL Register scope
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

1.1.3 CSL Register mandatory commands

Registers are differentiated according to their type. Mandatory commands have to be called before optional commands.Setting the type and the width for a register is mandatory as shown below:

```
    csl_register::set_type(type);
    csl_register::set_width(numeric_expression);
```

1.1.4 CSL Register common commands

The following set of commands can be called on registers regardless of their type.

CSL Register common commands

```
    csl_register::set_attributes(attribute);
    csl_register::add_logic(neg_output);
    csl_register::add_logic(set[,set_value]);
    csl_register::add_logic(reset[,reset_value]);
```

1.1.5 CSL Register specific commands

Once a type is set for a register certain commands apply only to that specific register type. These commands are used to customize the final register according to the user's needs. Note that D type register does not have any specific commands.

1.1.5.1 Counter Register specific commands

The following commands apply only to the counter register. These optional commands are useful when customizing the features of the generated RTL counter register.

```
    csl_register::add_logic(count_amount,numeric_expression);
    csl_register::add_logic(start_value,numeric_expression);
    csl_register::add_logic(load);
    csl_register::add_logic(count_en, signal);
    csl_register::add_logic(dir_control, signal);
    csl_register::add_logic(end_value, numeric_expression);
```

CSL Counter register mandatory commands

```
    csl_register::add_logic(count_direction, up|down);
```

1.1.6 CSL Register usage and rules

CSL Registers are declared in the global scope and can be used in designs by being instantiated in csl_unit and csl_testbench.

CSL Register can be instantiated in Table 1.4:

TABLE 1.3 CSL Register declaration rules

CSL class	Instantiates CSL Register
global scope	YES
CSL Unit	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

CSL Register can be instantiated in Table 1.4:

TABLE 1.4 CSL Register instantiation rules

CSL class	Instantiates CSL Register
CSL Unit	YES
CSL Testbench	YES
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

1.2 CSL Register Commands

1.2.1 Registers

csl_register::set_type(type);

DESCRIPTION :

Sets the type of the register. Supported types include counter or register. Counter infers a user configurable counter register while register type infers a D type register. Register types are mutually exclusive. Only one type may be set for a register. Register counter needs a count signal and a direction control signal.

TABLE 1.5 Register types

type	Results in
counter	counter register
register	D type register

EXAMPLE :

The counter has a count signal. The counter defaults to a start value of 1, and up direction counter. An optional down or up/down direction(s)s may be set.
NOTE: it is illegal to use counter add logic options without setting the register type to counter.

CSL CODE

```
csl_register r4{
  r4(){
    set_width(6);
    set_type(counter);
    add_logic(count_direction,up);
  }
};
```

VERILOG CODE

```
module r4(reset_,
          enable,
          clock,
          reg_out);
parameter MIN_VALUE={6{0}};
input reset_;
input enable;
input clock;
output [5:0] reg_out;
reg [5:0] st_reg;
assign   reg_out = st_reg;
always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      st_reg <= MIN_VALUE;
    end
    else if ( enable )  begin
        st_reg <= st_reg+1;
      end
  end
  endmodule
```

csl_register::set_width(numeric_expression**);**

DESCRIPTION :

Sets the width (in bits) of the register.

EXAMPLE :

In this example the type of register is set as register and the width is set 16 (bits).

CSL CODE

```
csl_register r1{
  r1(){
    set_type(register);
    set_width(16);
  }
};
```

VERILOG CODE

```
module r1(reset_,
          enable,
          clock,
          reg_out,
          reg_in);
  input reset_;
  input enable;
  input clock;
  input [15:0] reg_in;
  output [15:0] reg_out;
  reg [15:0] st_reg;
  assign   reg_out = st_reg;
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      st_reg <= 0;
    end
    else if ( enable )  begin
        st_reg <= reg_in;
      end
  end
  endmodule
```

1.2.1.1 Register attribute

csl_register::set_attributes(attribute);

DESCRIPTION :

Apply access attributes to the memory map page using predefined **attribute** bits
object_name is any state element type which can be part of the memory map page (eg cells, register, table, register file, SRAM). Sets the attribute for the *reg_object_name* register. If the attribute is not set, the default value is read(rd).The memory attributes can be only one at a time like parameter (*rd* or *wr* or *sh* or *rd_wr*). Memory attributes control the access to a memory mapped structure and can be specified according to Table 1.6.

TABLE 1.6 CSL memory map element atrribute bits

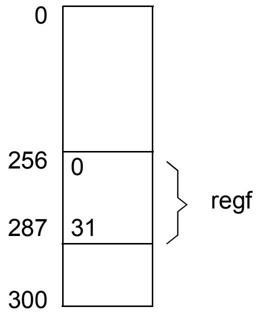
Attribute	Description
rd	read only register
wr	write only register
sh	shadow register
rd_wr	read/write register

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a memory map with a register file inside.

FIGURE 1.1 A memory map page with a register file element



CSL CODE

```
csl_register regA{
  regA(){
    set_type(register);
    set_width(32);
    set_attributes(rd);
  }
};
csl_memory_map_page mpage{
   regA regA;
   mpage(){
    add_address_range(0,300);
  }
};
```

VERILOG CODE

```
  `define <MMN>_OBJ_NAME_ATTRIBUTES attribute_list;
```

1.2.1.2 Register pins

csl_register::add_logic(*neg_output*);

port: output - neg_output

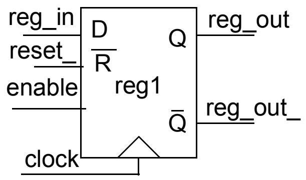
DESCRIPTION :

Automatical create the parallel negative output port with the name *neg_output* for the *reg_object_ name* register. The output port is n-bit width (user defined).

[CSL Register Syntax and Command Summary]

EXAMPLE :

FIGURE 1.2



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(register);
    set_width(4);
    add_logic(neg_output);
  }
};
```

VERILOG CODE

```
module reg1(reset_,
            enable,
            clock,
            reg_out,
            reg_in,
            reg_out_);
  input reset_;
  input enable;
  input clock;
  input [3:0] reg_in;
  output [3:0] reg_out;
  output [3:0] reg_out_;
  reg [3:0] st_reg;
  assign   reg_out = st_reg;
  assign   reg_out_ = ~st_reg;
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
```

```
        st_reg <= 0;
      end
      else if ( enable )  begin
          st_reg <= reg_in;
        end
    end
    endmodule
```

1.2.1.3 Values

csl_register::add_logic(set[,*set_value*]**);**

DESCRIPTION :

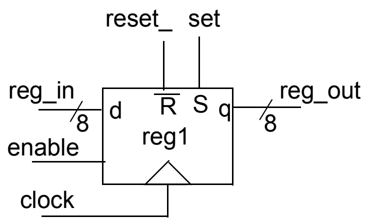
Add synchronous set logic to register.After a set operation the memory element is set to this value. Default is one.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a register named *reg1* that will have an synchronous set pin.

FIGURE 1.3 A register with synchronous set signal



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(register);
    set_width(8);
    add_logic(set,3);
  }
};
```

VERILOG CODE

```
module reg1(reset_,
            enable,
            clock,
            reg_out,
            reg_in,
            set);
  input reset_;
  input enable;
  input clock;
  input [7:0] reg_in;
  input set;
  output [7:0] reg_out;
  reg [7:0] st_reg;
  assign   reg_out = st_reg;
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
     st_reg <= {8{1'b0}};
    end
    else if ( set )  begin
        st_reg <= 8'd3;
      end
      else if ( enable )  begin
          st_reg <= reg_in;
      end
  end
  endmodule
```

csl_register::add_logic(reset[,*reset_value*]**);**

DESCRIPTION :

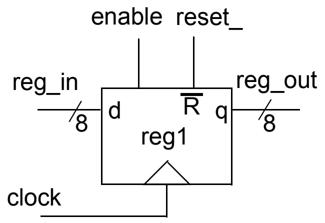
Add reset logic and reset value to register. After a reset operation the memory element is set to this value.Default is zero.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a register named *reg1* that will have an asynchronous reset pin.

FIGURE 1.4 A register with asynchronous reset signal



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(register);
    set_width(8);
    add_logic(reset,0);
  }
};
```

VERILOG CODE

```
module reg1(reset_,
            enable,
            clock,
            reg_out,
            reg_in);
  input reset_;
  input enable;
  input clock;
  input [7:0] reg_in;
  output [7:0] reg_out;
  reg [7:0] st_reg;
  assign   reg_out = st_reg;
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      st_reg <={8{1'b0}};
    end
    else if ( enable )  begin
```

```
          st_reg <= reg_in;
        end
    end
    endmodule
```

1.2.1.4 Register types

1.2.1.4.1 Counter register

csl_register::add_logic(count_amount,*numeric_expression*);

DESCRIPTION :

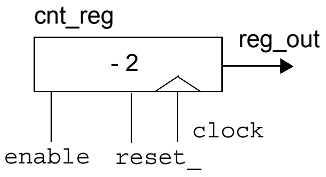
Set the value to increment/decrement the counters.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that will count down to zero from an initial value. The decrementation value is 2.

FIGURE 1.5



CSL CODE

```
  csl_register cnt_reg{
    cnt_reg(){
      set_type(counter);
      set_width(8);
      add_logic(count_direction, down);
      add_logic(start_value,8);
      add_logic(count_amount,2);
      add_logic(end_value,0);
    }
  };
```

VERILOG CODE

```
  module cnt_reg(reset_,
             enable,
             clock,
             reg_out);
    input reset_;
    input enable;
    input clock;
    output [7:0] reg_out;
    reg [7:0] st_reg;
    wire [7:0] START_VALUE = 8'h08;
    wire [7:0] END_VALUE = 8'h00;
    assign   reg_out = st_reg;
    always @( posedge clock or negedge reset_ )  begin
      if ( ~reset_ )  begin
        st_reg <= START_VALUE;
      end
    else if ( enable )  begin
        if ( st_reg == END_VALUE )  begin
          st_reg <= START_VALUE;
        end
        else  begin
          st_reg <= st_reg - 2;
        end
      end
  end
  endmodule
```

csl_register::add_logic(start_value,*numeric_expression*);

DESCRIPTION :

Set the start value. When the counter reaches the end value then the counter resets to the start value. Default is zero.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that count from start_value 0 to 128. The decrementation value is 1.The count direction is up.

CSL CODE

```
csl_register reg_cnt{
  reg_cnt(){
    set_type(counter);
    set_width(8);
    add_logic(count_direction,up);
    add_logic(start_value,0);
    add_logic(count_amount,1);
    add_logic(end_value,128);
  }
};
```

VERILOG CODE

```
module reg_cnt(reset_,
               enable,
               clock,
               reg_out);
  input reset_;
  input enable;
  input clock;
  output [7:0] reg_out;
  reg [7:0] st_reg;
  wire [7:0] START_VALUE = 8'h00;
  wire [7:0] END_VALUE = 8'd128;
  assign   reg_out = st_reg;
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      st_reg <= START_VALUE;
    end
    else if ( enable )  begin
        if ( st_reg == END_VALUE )  begin
          st_reg <= START_VALUE;
        end
        else  begin
          st_reg <= st_reg + 1;
        end
      end
  end
endmodule
```

csl_register::add_logic(load);

DESCRIPTION :

This command infers a load and load trigger port for the counter. When the load trigger is active, the counter is loaded with the value on the load port and continues to count from this value. The load port has the same width as the counter and the load trigger port is 1 bit.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that count up. It is loaded with the value on the load port.

CSL CODE

```
csl_register r1{
   r1 (){
   set_type(counter);
   set_width(8);
   add_logic(count_direction,up);
   add_logic(load);
  }
};
```

VERILOG CODE

```
module r1(reset_,
          enable,
          clock,
          reg_out,
          load,
          load_en);
  parameter MIN_VALUE = {8 {1'b0}};
  input reset_;
  input enable;
  input clock;
  input [7:0] load;
  input load_en;
  output [7:0] reg_out;
  reg [7:0] st_reg;
  assign   reg_out = st_reg;
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      st_reg <= MIN_VALUE;
    end
    else if ( enable )  begin
        if ( load_en )  begin
          st_reg <= load;
      end
      else  begin
        st_reg <= st_reg + 1;
      end
    end
end
endmodule
```

`csl_register::add_logic``(`**`count_en`**`, signal);`

DESCRIPTION :

Add a signal which enables counting.

I

[CSL Register Syntax and Command Summary]

EXAMPLE :

//

CSL CODE

```
    //
```

VERILOG CODE

```
    //
```

```
csl_register::add_logic(dir_control, signal);
```

DESCRIPTION :

Add a signal which controls the count direction.

[CSL Register Syntax and Command Summary]

EXAMPLE :

//

CSL CODE

```
    //
```

VERILOG CODE

```
    //
```

csl_register::add_logic(**end_value**, numeric_expression);

DESCRIPTION :

Sets the end value. When the counter reaches the end value then the counter resets to the start value. Default is zero.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Creates a counter that count to 128. The decrementation value is 1. The count direction is up.

CSL CODE

```
csl_register reg_cnt{
  reg_cnt(){
    set_type(counter);
    set_width(8);
    add_logic(count_direction,up);
    add_logic(start_value,0);
    add_logic(count_amount,1);
    add_logic(end_value,128);
  }
};
```

VERILOG CODE

```
module reg_cnt(reset_,
           enable,
           clock,
           reg_out);
  input reset_;
  input enable;
  input clock;
  output [7:0] reg_out;
  reg [7:0] st_reg;
  wire [7:0] START_VALUE = 8'd0;
  wire [7:0] END_VALUE = 8'd128;
  assign   reg_out = st_reg;
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      st_reg <= START_VALUE;
    end
    else if ( enable )  begin
        if ( st_reg == END_VALUE )  begin
          st_reg <= START_VALUE;
        end
        else  begin
```

```
            st_reg <= st_reg + 1;
          end
        end
    end
    endmodule
```

csl_register::add_logic(count_direction, up|down);

DESCRIPTION :

This command set the count direction by modifing a flag bit for direction named *count_direction* . Default direction is up.

TABLE 1.7 Count direction and flag value

Count direction	Description
up	count up
down	count down

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that will count down to zero from an initial value. The count direction is down.

CSL CODE

```
csl_register reg_cnt{
  reg_cnt(){
    set_type(counter);
    set_width(8);
    add_logic(count_direction, down);
    add_logic(start_value,128);
    add_logic(count_amount,1);
    add_logic(end_value,0);
  }
};
```

VERILOG CODE

```
module reg_cnt(reset_,
             enable,
             clock,
             reg_out);
  input reset_;
  input enable;
  input clock;
  output [7:0] reg_out;
  reg [7:0] st_reg;
  wire [7:0] START_VALUE = 8'd128;
  wire [7:0] END_VALUE = 8'd0;
  assign   reg_out = st_reg;
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      st_reg <= START_VALUE;
    end
    else if ( enable )  begin
        if ( st_reg == END_VALUE )  begin
          st_reg <= START_VALUE;
        end
        else  begin
          st_reg <= st_reg - 1;
        end
      end
  end
  endmodule
```