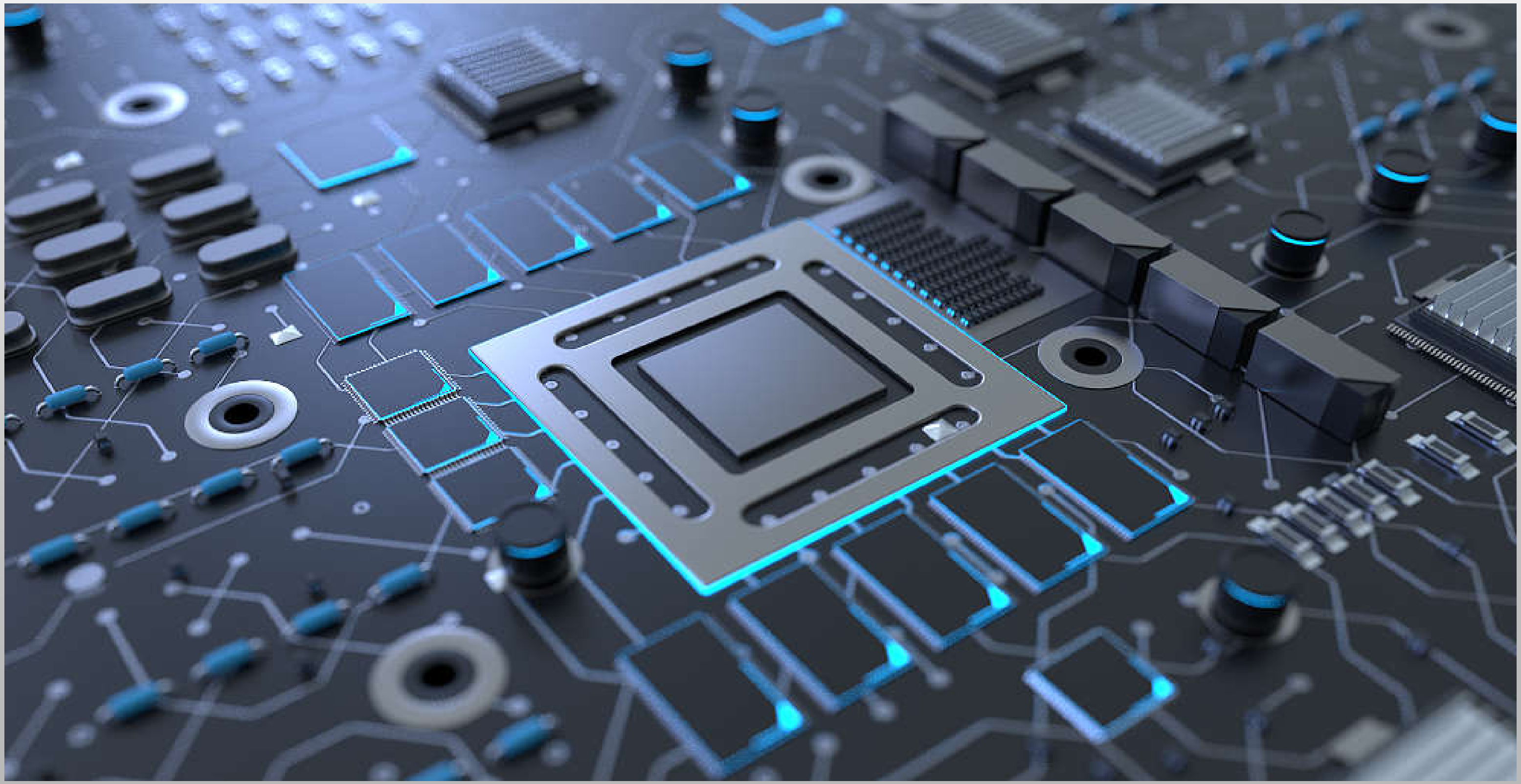
HDLGEN INTRODUCTION

WILSON CHEN

2023-02



CONTENT

- OVERVIEW
- USAGE
- EMBEDDED FUNCTIONS
- EXTENDED FUNCTIONS
- OTHER FUNCTIONS
- THANKS & NOTICE

OVERVIEW

HDLGen is a tool for HDL generation, it enables embedded Perl or Python scripts in Verilog source code, and support Perl style variable anyway, to generate desired HDL in an easy and efficient way.

It supports all syntax and data structure of Perl or Python, and has a few predefined functions for signal define, module instance, port connection etc.

This tool also supports extended API functions in Perl format(Python API not on plan yet), for any function or module that you want or have.

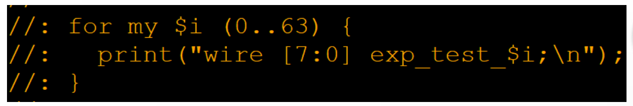
HDL and script mixed design file can be any name, while final generated RTL file will be Verilog(.v)

Assume line starting with "//:" to be single line Perl script;

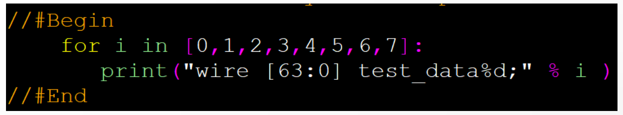
Assume line starting with "//:Begin" and ending with "//:End" are multi-line Perl script;

Assume line starting with "//#" to be single line Python script;

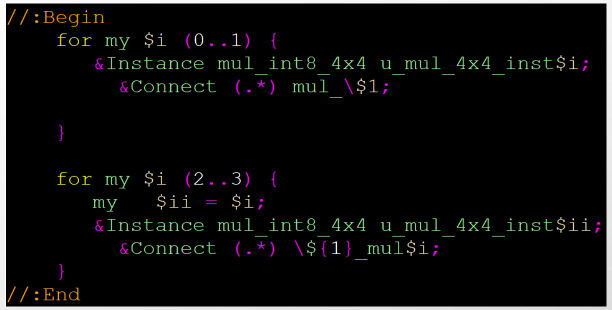
Assume line starting with "//#Begin" and ending with "//#End" are multi-line Python script;



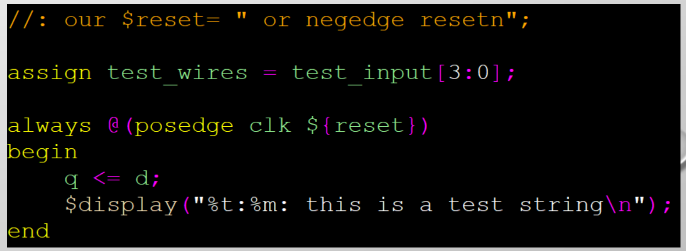
```
//: for my $i (0..63) {
//:   print("wire [7:0] exp_test_$i;\n");
//: }
```



```
//#Begin
    for i in [0,1,2,3,4,5,6,7]:
       print("wire [63:0] test_data%d;" % i )
//#End
```



```
//:Begin
    for my $i (0..1) {
       &Instance mul_int8_4x4 u_mul_4x4_inst$i;
         &Connect (.*) mul_\$1;

    }

    for my $i (2..3) {
       my   $ii = $i;
       &Instance mul_int8_4x4 u_mul_4x4_inst$ii;
         &Connect (.*) \${1}_mul$i;
    }
//:End
```



```
//: our $reset= " or negedge resetn";

assign test_wires = test_input[3:0];

always @(posedge clk ${reset})
begin
    q <= d;
    $display("%t:%m: this is a test string\n");
end
```

USAGE - INPUT

HDLGen supports single source file or multi-file(through a filelist file)， generate Verilog HDL with same name. It only need 1 input option default, like:

HDLGen.pm -i my_design.src

and **my_design.v** will be generated as a pure Verilog HDL file.

or **HDLGen.pm –f src.flist,** all files in src.flist will be processed and generate one Verilog file for 1 input.

Other options usage：

-u[usage] **:** print usage or helping message

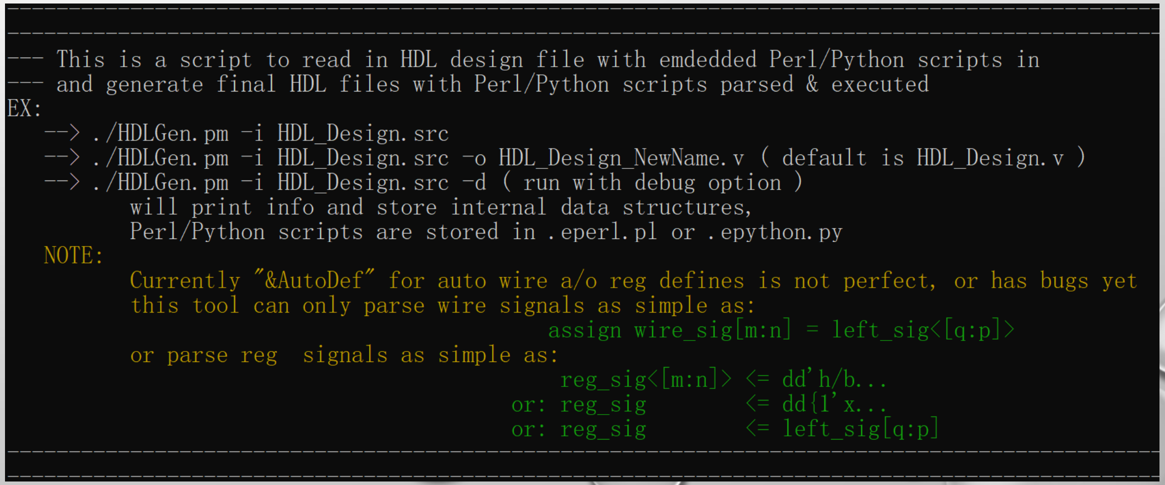
-o[output] **:** override output file

-d[debug] **:** debug, several intermedia files will be saved to help debug

-v[verbose] : verbal mode， will print a lot of information on screen, if –debug turned on too(rarely used)

Suggestion：

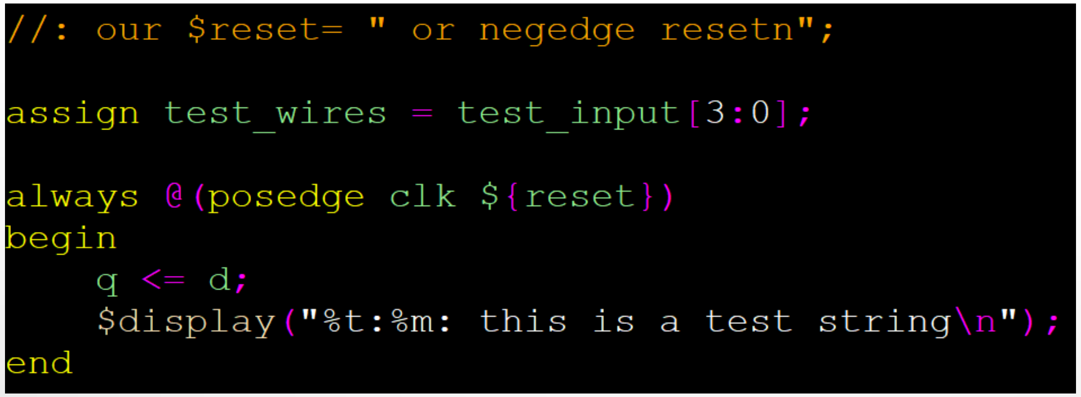
- Only use (-I) in most case；
- Turn on (-d) if error message not understood;



```
------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------
--- This is a script to read in HDL design file with emdedded Perl/Python scripts in
--- and generate final HDL files with Perl/Python scripts parsed & executed
EX:
   --> ./HDLGen.pm -i HDL_Design.src
   --> ./HDLGen.pm -i HDL_Design.src -o HDL_Design_NewName.v ( default is HDL_Design.v )
   --> ./HDLGen.pm -i HDL_Design.src -d ( run with debug option )
          will print info and store internal data structures,
          Perl/Python scripts are stored in .eperl.pl or .epython.py
   NOTE:
          Currently "&AutoDef" for auto wire a/o reg defines is not perfect, or has bugs yet
          this tool can only parse wire signals as simple as:
                                             assign wire_sig[m:n] = left_sig<[q:p]>
          or parse reg  signals as simple as:
                                              reg_sig<[m:n]> <= dd'h/b...
                                          or: reg_sig        <= dd{1'x...
                                          or: reg_sig        <= left_sig[q:p]
------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------
```

USAGE – FLEXIBLE VARIABLES

You can use Perl style variable wherever in Verilog code, as long as you defined such variable before using it.



```
//: our $reset= " or negedge resetn";

assign test_wires = test_input[3:0];

always @(posedge clk ${reset})
begin
    q <= d;
    $display("%t:%m: this is a test string\n");
end
```

Note:

- Such variable can be used wherever as native Verilog code, without any “**//:**”;
- But such variable must be defined as “**our**” type;
- And these variables must be used with “**{}**”, like **${reset},** to differentiate from Verilog embedded functions;

EMBEDDED AND EXTENDED FUNCTIONS

HDLGen has a few embedded functions, which help to achieve RTL generation and IP integration.
NOTE:

- Function call has 2 ways: **&Function()**, or **Function();**
  - "**&**" is optional but suggested;
- Function parameters have 2 style:
  - Direct string split by **","**, order is critical;
  - Linux command **-option** like, order is meaningless;
  - Details refer to each function.
- Embedded function can be used directly;
- Extended functions need to add package prefix : **&eFunc::**fun(…)

```
//: &eFunc::ClkGen("Test_Clk", "./cfg/Clk_Cfg.json");

//: &eFunc::RstGen("Test_Rst", "./cfg/Rst_Cfg.json");

//: &eFunc::FuseGen("Test_Fuse", "./cfg/Fuse_Cfg.json");

//: &eFunc::PmuGen("Test_Pmu", "./cfg/Pmu_Cfg.json");

//: &eFunc::MemGen("Test_Mem", "./cfg/Mem_Cfg.json");

//: &eFunc::AsyncIntfGen("Test_AsyncIntf", "./cfg/AsyncIntf_Cfg.json");

//: &eFunc::FifoGen("Test_SFifo", "./cfg/SFifo_Cfg.json");

&Instance Test_SFifo;

//: &eFunc::FifoGen("Test_AFifo", "./cfg/AFifo_Cfg.json");

&Instance Test_AFifo;
```

```
&Instance test_sys_ctrl_apb_regs;
  Connect -final -interface APB3  -up \${1}_suffix ;
```

```
&Instance NV_NVDLA_CMAC_CORE_MAC_mul u_mul_$i;
  &Connect exp_sft       exp_sft_$ii;
   Connect op_a_dat      wt_actv_data${i};
   Connect op_a_nz       wt_actv_nz${ii};
   Connect op_a_pvld      wt_actv_pvld[${i}];
  &Connect op_b_dat      dat_actv_data${i};
  &Connect op_b_nz       dat_actv_nz${i};
  &Connect op_b_pvld      dat_actv_pvld[${i}];
   Connect /(res_.*)/     \${1}_$ii;
   Connect -final (res_tag)    \${1}_$i; ### override above line
```

```
//:Begin
  my $sv = "
  interface test_if(input clk);
    logic rst_n,
    wire  [1:0] port_a_0 ;
    logic [12:0] port_a_1 ;
    wire port_b_0 ;
    logic port_b_1 ;
   endinterface
";
    &AddIntfBySV($sv);
    &ShowIntf("test_if");
    &PrintIntfPort("-intf test_if -up");
//:End
```

EMBEDDED FUNCTION - print

There are 2 types of print functions in this tool:

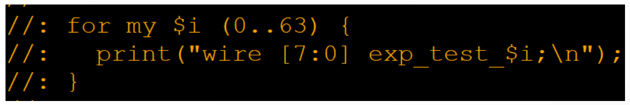
1. Perl standard “**print**”;
2. Verilog line(s) print as “**vprintl**”;

Usage:

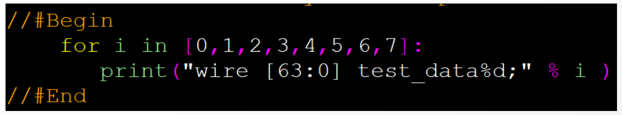
//: print(“string to print\n”);
//: vprint(“string to print\n”);

NOTE:

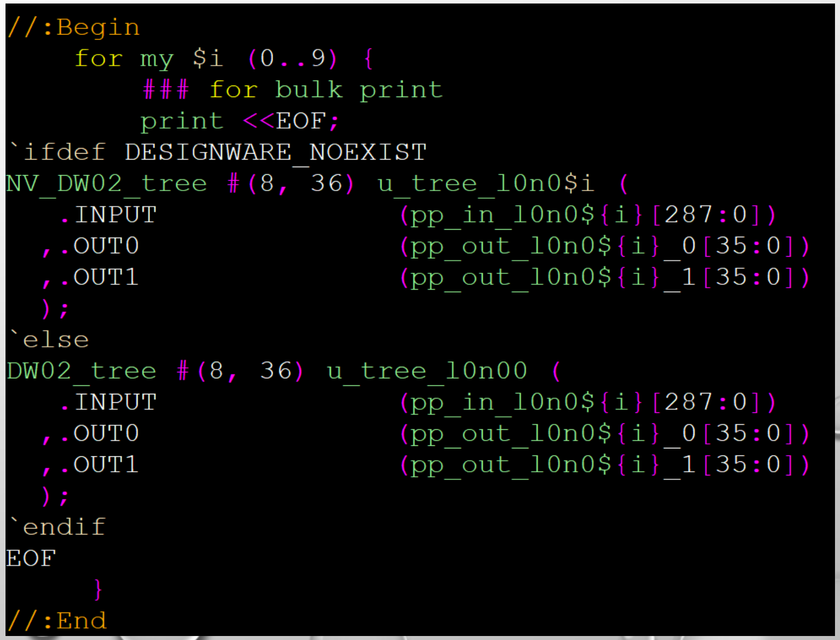
- the difference between 2 functions is that **vprintl** will NOT print to screen at all, it only update internal data structures, while **print** can print info on screen if you write correctly;
- it’s more safe to use “**print STDOUT** …” if you want to print on screen for debug;
- if you want to print a bulk of code, you can use **“print <<EOF; …EOF”;**
- Python is different! --- it only has Python native **print** function.



```
//: for my $i (0..63) {
//:   print("wire [7:0] exp_test_$i;\n");
//: }
```



```
//#Begin
    for i in [0,1,2,3,4,5,6,7]:
        print("wire [63:0] test_data%d;" % i )
//#End
```



```
//:Begin
    for my $i (0..9) {
        ### for bulk print
        print <<EOF;
`ifdef DESIGNWARE_NOEXIST
NV_DW02_tree #(8, 36) u_tree_l0n0$i (
    .INPUT                (pp_in_l0n0${i}[287:0])
  ,.OUT0                  (pp_out_l0n0${i}_0[35:0])
  ,.OUT1                  (pp_out_l0n0${i}_1[35:0])
  );
`else
DW02_tree #(8, 36) u_tree_l0n00 (
    .INPUT                (pp_in_l0n0${i}[287:0])
  ,.OUT0                  (pp_out_l0n0${i}_0[35:0])
  ,.OUT1                  (pp_out_l0n0${i}_1[35:0])
  );
`endif
EOF
     }
//:End
```

EMBEDDED FUNCTION - SRC

This function is used to update source file search path, and can be used multiple times for multiple paths. When other functions need a file not in current path, then tool will search in all search paths to find target file(will report error if no file in all paths)

Usage：

```
//: SRC ./incr;
//: &SRC ./cfg;
```

NOTE：

- you can use absolute path in other functions, but sometime it’s not so convenient;

EMBEDDED FUNCTION - AutoDef

This function is used to automatically generate wire or reg definition for all logic code in 1 source file, that's to say, you can write logic code directly without signal defined first.

But, please note this tool supports very limited syntax so far(**welcome any suggestion a/o solution to improve!**) , like:

assign wire_sig[m:n] = left_sig[q:p]

assign wire_sig = {left_sig[q:p],8'b0,…}

{wire_sig0,wire_sig1..} = {left_sig0,left_sig1…}

reg_sig[m:n] <= dd'h/b...

reg_sig <= dd{1'x...

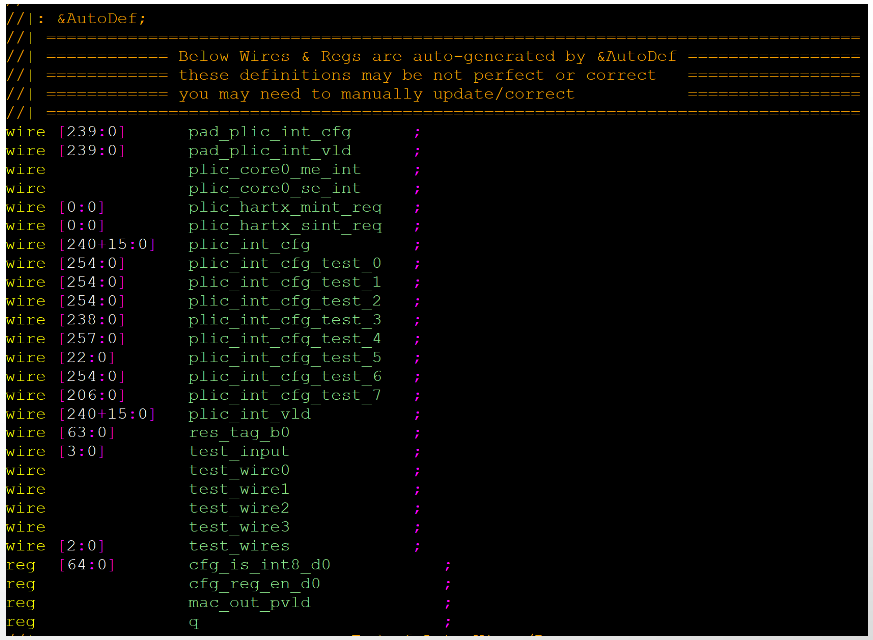
reg_sig <= left_sig[q:p]

Usage:

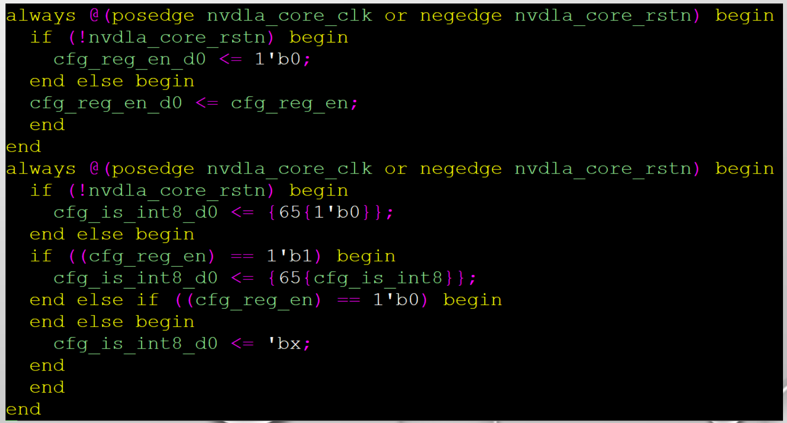
//: &AutoDef;

NOTE: this function need to put before all wire/reg define lines; simple parameter & define is supported

- Suggest to enable;
- But keep in mind this is not perfect
  - It's only a nice to be or backup solution
- Complex logic's signals suggested to manual declare



```
//|: &AutoDef;
//| ==========================================================================================
//| =========== Below Wires & Regs are auto-generated by &AutoDef ================
//| =========== these definitions may be not perfect or correct   ================
//| =========== you may need to manually update/correct           ================
//| ==========================================================================================
wire [239:0]      pad_plic_int_cfg      ;
wire [239:0]      pad_plic_int_vld      ;
wire              plic_core0_me_int     ;
wire              plic_core0_se_int     ;
wire [0:0]        plic_hartx_mint_req   ;
wire [0:0]        plic_hartx_sint_req   ;
wire [240+15:0]   plic_int_cfg          ;
wire [254:0]      plic_int_cfg_test_0   ;
wire [254:0]      plic_int_cfg_test_1   ;
wire [254:0]      plic_int_cfg_test_2   ;
wire [238:0]      plic_int_cfg_test_3   ;
wire [257:0]      plic_int_cfg_test_4   ;
wire [22:0]       plic_int_cfg_test_5   ;
wire [254:0]      plic_int_cfg_test_6   ;
wire [206:0]      plic_int_cfg_test_7   ;
wire [240+15:0]   plic_int_vld          ;
wire [63:0]       res_tag_b0            ;
wire [3:0]        test_input            ;
wire              test_wire0            ;
wire              test_wire1            ;
wire              test_wire2            ;
wire              test_wire3            ;
wire [2:0]        test_wires            ;
reg  [64:0]       cfg_is_int8_d0            ;
reg               cfg_reg_en_d0             ;
reg               mac_out_pvld              ;
reg               q                         ;
```



```
always @(posedge nvdla_core_clk or negedge nvdla_core_rstn) begin
  if (!nvdla_core_rstn) begin
    cfg_reg_en_d0 <= 1'b0;
  end else begin
  cfg_reg_en_d0 <= cfg_reg_en;
  end
end
always @(posedge nvdla_core_clk or negedge nvdla_core_rstn) begin
  if (!nvdla_core_rstn) begin
    cfg_is_int8_d0 <= {65{1'b0}};
  end else begin
  if ((cfg_reg_en) == 1'b1) begin
    cfg_is_int8_d0 <= {65{cfg_is_int8}};
  end else if ((cfg_reg_en) == 1'b0) begin
  end else begin
    cfg_is_int8_d0 <= 'bx;
  end
  end
end
```

EMBEDDED FUNCTION – AutoInstSig

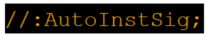
This function is to automatically generate module instance's port connected wire define, but it only supports those modules instanced by embedded function of "**&Instance**" , other module instanced by Verilog syntax is not supported yet (can support if requirement exist) **:**

Usage:

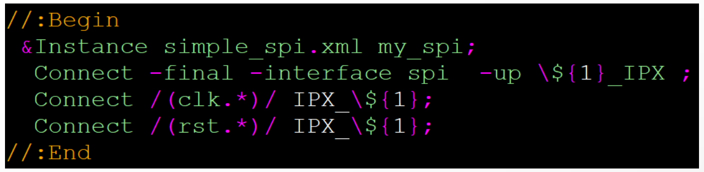
//: & AutoInstSig;

NOTE:

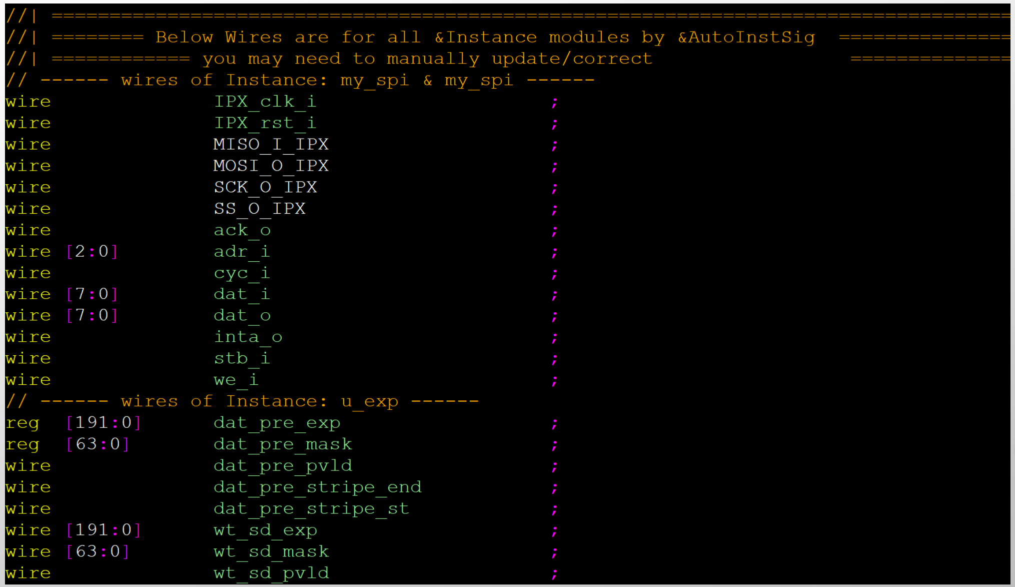
- **This function can be in anywhere, but suggest to be around wire/reg lines;**
- **port connection default is "wire" type, but tool will parse all existing code to see if "reg" type is correct for any signal**
- **Any manual defines(wire or reg) in original code will override and bypass those auto-signals.**
- **Signal width will auto-learning**



```
//:AutoInstSig;
```



```
//:Begin
 &Instance simple_spi.xml my_spi;
  Connect -final -interface spi  -up \${1}_IPX ;
  Connect /(clk.*)/ IPX_\${1};
  Connect /(rst.*)/ IPX_\${1};
//:End
```



```
//| ================================================================================
//| ======== Below Wires are for all &Instance modules by &AutoInstSig  ==============
//| =========== you may need to manually update/correct                  =============
// ------ wires of Instance: my_spi & my_spi ------
wire              IPX_clk_i                    ;
wire              IPX_rst_i                    ;
wire              MISO_I_IPX                   ;
wire              MOSI_O_IPX                   ;
wire              SCK_O_IPX                    ;
wire              SS_O_IPX                     ;
wire              ack_o                        ;
wire [2:0]        adr_i                        ;
wire              cyc_i                        ;
wire [7:0]        dat_i                        ;
wire [7:0]        dat_o                        ;
wire              inta_o                       ;
wire              stb_i                        ;
wire              we_i                         ;
// ------ wires of Instance: u_exp ------
reg  [191:0]      dat_pre_exp                  ;
reg  [63:0]       dat_pre_mask                 ;
wire              dat_pre_pvld                 ;
wire              dat_pre_stripe_end           ;
wire              dat_pre_stripe_st            ;
wire [191:0]      wt_sd_exp                    ;
wire [63:0]       wt_sd_mask                   ;
wire              wt_sd_pvld                   ;
```

Strongly recommended, it ease your work!

EMBEDDED FUNCTION – AutoInstSig- AutoWarning

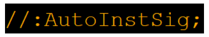
When is enabled, then HDLGen will check all instance’s port connected signals, if any input port has no source, or output has no sink in current design, then such signal will be printed out in final RTL as a “Warning”, and a warning message will list on screen to cause your attention on these signals as they may be wrong or unexpected.

Usage:

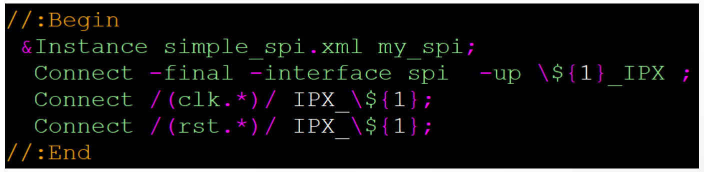
//: & AutoInstSig;

NOTE:

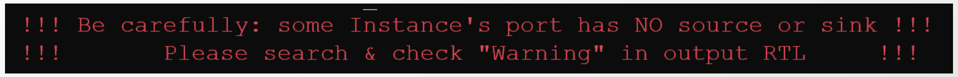
- **This function is auto-enabled whenever AutoInstSig is enabled;**
- **Only those sub-modules instanced by &Instance function will be parsed.**
- **These signals will be listed in instance order.**
- **This function may be not perfect but should be correct in most case.**



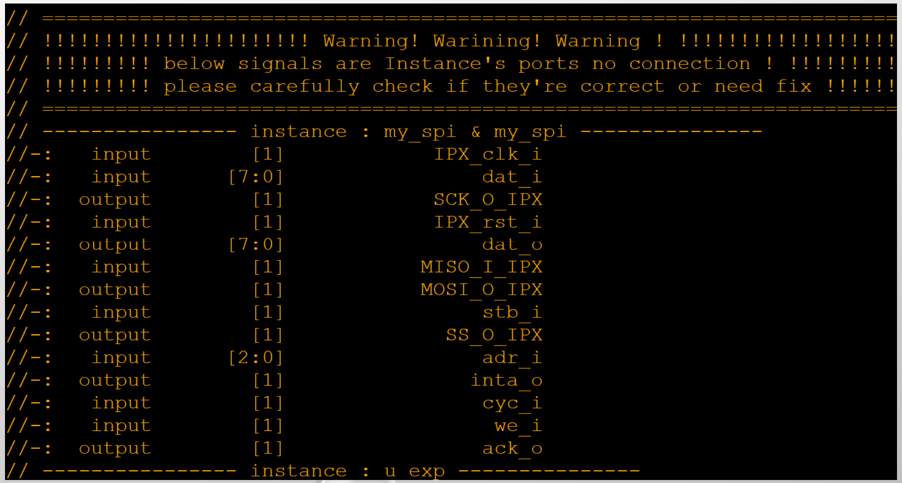
```
//:AutoInstSig;
```



```
//:Begin
 &Instance simple_spi.xml my_spi;
  Connect -final -interface spi  -up \${1}_IPX ;
  Connect /(clk.*)/ IPX_\${1};
  Connect /(rst.*)/ IPX_\${1};
//:End
```



```
 !!! Be carefully: some Instance's port has NO source or sink !!!
 !!!         Please search & check "Warning" in output RTL      !!!
```



```
// ======================================================================
// !!!!!!!!!!!!!!!!!!!!! Warning! Warining! Warning ! !!!!!!!!!!!!!!!!!!
// !!!!!!!!! below signals are Instance's ports no connection ! !!!!!!!!!
// !!!!!!!!! please carefully check if they're correct or need fix !!!!!!
// ======================================================================
// --------------- instance : my_spi & my_spi ---------------
//-:   input        [1]             IPX_clk_i
//-:   input      [7:0]                 dat_i
//-:  output        [1]             SCK_O_IPX
//-:   input        [1]             IPX_rst_i
//-:  output      [7:0]                 dat_o
//-:   input        [1]            MISO_I_IPX
//-:  output        [1]            MOSI_O_IPX
//-:   input        [1]                 stb_i
//-:  output        [1]              SS_O_IPX
//-:   input      [2:0]                 adr_i
//-:  output        [1]                inta_o
//-:   input        [1]                 cyc_i
//-:   input        [1]                  we_i
//-:  output        [1]                 ack_o
// --------------- instance : u exp --------------
```

Strongly recommended, it ease your work!

EMBEDDED FUNCTION -Instance

This function is used to instance sub-module, its syntax is very similar to Verilog instance, like:

&Instance NV_NVDLA_CMAC_CORE_MAC_mul u_mul_0

Or:

&Instance NV_NVDLA_CMAC_CORE_MAC_mul #(.param0(xxx), .param1(yy) ..) u_mul_0

Or without instance name:

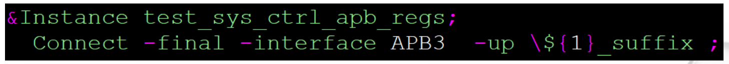
&Instance NV_NVDLA_CMAC_CORE_MAC

NOTE:

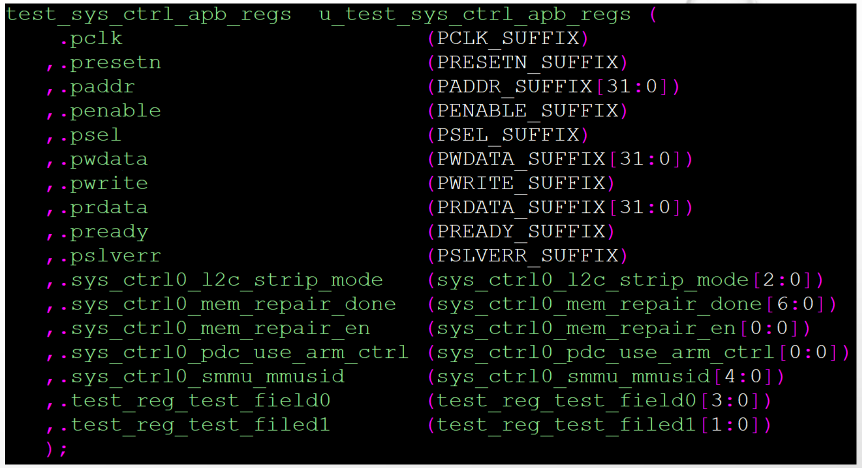
- &Instance codes can be with or without starting head of "**//:**"or"**//:Begin**" "**//:End**", which mean it can be treated as native Verilog code(suggested mode);
- If &Instance only has module name, then instance will be default as: **u_module**
- &Instance need to be used together with Connect in most case, like:

  &Instance NV_NVDLA_CMAC_CORE_MAC_mul u_mul_0
  &Connect exp_sft exp_sft_00[3:0];
  &Connect /op_a_(\w*)/ wt_actv_\${1}0;
- If no **&Connect** line after &Instance line, then all ports of this instance will be connected to wires as same name of port
- If **&AutoInstSig** function is called before &Instance, then all wires connected to this instance's ports will be auto-defined at right place;
- If there is no **&AutoInstSig** function before &Instance, then all wires connected to this instance's ports will be generated below the instance as commented lines (starting with //), and you can manually copy/change later.

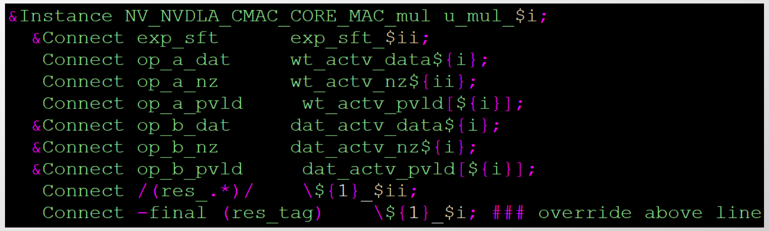
Thanks NVIDIA for NVDLA as a testing source



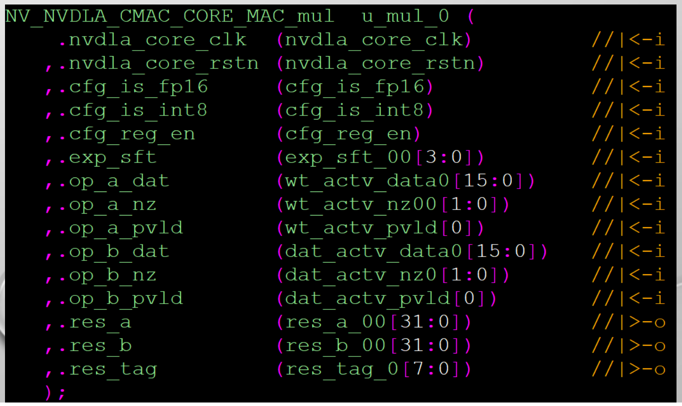
```
&Instance test_sys_ctrl_apb_regs;
  Connect -final -interface APB3  -up \${1}_suffix ;
```



```
test_sys_ctrl_apb_regs  u_test_sys_ctrl_apb_regs (
    .pclk                           (PCLK_SUFFIX)
   ,.presetn                        (PRESETN_SUFFIX)
   ,.paddr                          (PADDR_SUFFIX[31:0])
   ,.penable                        (PENABLE_SUFFIX)
   ,.psel                           (PSEL_SUFFIX)
   ,.pwdata                         (PWDATA_SUFFIX[31:0])
   ,.pwrite                         (PWRITE_SUFFIX)
   ,.prdata                         (PRDATA_SUFFIX[31:0])
   ,.pready                         (PREADY_SUFFIX)
   ,.pslverr                        (PSLVERR_SUFFIX)
   ,.sys_ctrl0_l2c_strip_mode       (sys_ctrl0_l2c_strip_mode[2:0])
   ,.sys_ctrl0_mem_repair_done      (sys_ctrl0_mem_repair_done[6:0])
   ,.sys_ctrl0_mem_repair_en        (sys_ctrl0_mem_repair_en[0:0])
   ,.sys_ctrl0_pdc_use_arm_ctrl     (sys_ctrl0_pdc_use_arm_ctrl[0:0])
   ,.sys_ctrl0_smmu_mmusid          (sys_ctrl0_smmu_mmusid[4:0])
   ,.test_reg_test_field0           (test_reg_test_field0[3:0])
   ,.test_reg_test_filed1           (test_reg_test_filed1[1:0])
   );
```



```
&Instance NV_NVDLA_CMAC_CORE_MAC_mul u_mul_$i;
  &Connect exp_sft      exp_sft_$ii;
   Connect op_a_dat     wt_actv_data${i};
   Connect op_a_nz      wt_actv_nz${ii};
   Connect op_a_pvld     wt_actv_pvld[${i}];
  &Connect op_b_dat     dat_actv_data${i};
  &Connect op_b_nz      dat_actv_nz${i};
  &Connect op_b_pvld     dat_actv_pvld[${i}];
   Connect /(res_.*)/    \${1}_$ii;
   Connect -final (res_tag)    \${1}_$i; ### override above line
```



```
NV_NVDLA_CMAC_CORE_MAC_mul  u_mul_0 (
    .nvdla_core_clk  (nvdla_core_clk)          //|<-i
   ,.nvdla_core_rstn (nvdla_core_rstn)         //|<-i
   ,.cfg_is_fp16     (cfg_is_fp16)             //|<-i
   ,.cfg_is_int8     (cfg_is_int8)             //|<-i
   ,.cfg_reg_en      (cfg_reg_en)              //|<-i
   ,.exp_sft         (exp_sft_00[3:0])         //|<-i
   ,.op_a_dat        (wt_actv_data0[15:0])     //|<-i
   ,.op_a_nz         (wt_actv_nz00[1:0])       //|<-i
   ,.op_a_pvld       (wt_actv_pvld[0])         //|<-i
   ,.op_b_dat        (dat_actv_data0[15:0])    //|<-i
   ,.op_b_nz         (dat_actv_nz0[1:0])       //|<-i
   ,.op_b_pvld       (dat_actv_pvld[0])        //|<-i
   ,.res_a           (res_a_00[31:0])          //|>-o
   ,.res_b           (res_b_00[31:0])          //|>-o
   ,.res_tag         (res_tag_0[7:0])          //|>-o
   );
```

EMBEDDED FUNCTION – Instance - IPXACT

&Instance function has another way to use: **IPXACT** direct instance --- take IPXACT as a module to instance, like:

&Instance simple_spi.xml my_spi;

Or:

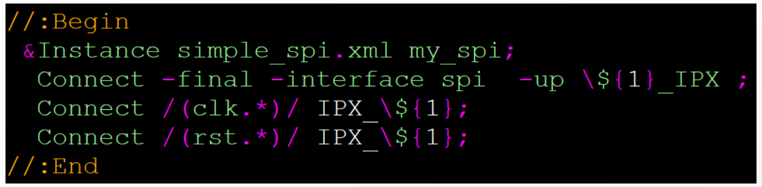
&Instance simple_spi.xml #(.param0(xxx), .param1(yy) ..) my_spi;

Or no Instance name: as

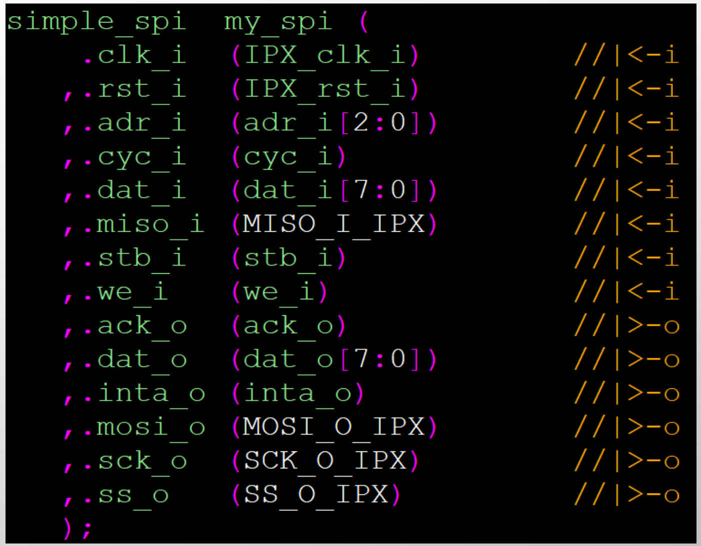
&Instance simple_spi.xml;

NOTE:

- **IPXACT** file name can be identical or different to module, module name will be defined by **IPXACT**'s "name" field;
  - if IPXACT has no "name" field then file name will be used;
- When instancing from **IPXACT** file, all interfaces defined in the **IPXACT** file will be automatically updated into internal interface list;
  - so you can use those interfaces directly;
- Other requirements/functions are common for &Instance;



```
//:Begin
 &Instance simple_spi.xml my_spi;
  Connect -final -interface spi  -up \${1}_IPX ;
  Connect /(clk.*)/ IPX_\${1};
  Connect /(rst.*)/ IPX_\${1};
//:End
```



```
simple_spi  my_spi (
    .clk_i   (IPX_clk_i)           //|<-i
   ,.rst_i   (IPX_rst_i)           //|<-i
   ,.adr_i   (adr_i[2:0])          //|<-i
   ,.cyc_i   (cyc_i)               //|<-i
   ,.dat_i   (dat_i[7:0])          //|<-i
   ,.miso_i  (MISO_I_IPX)          //|<-i
   ,.stb_i   (stb_i)               //|<-i
   ,.we_i    (we_i)                //|<-i
   ,.ack_o   (ack_o)               //|>-o
   ,.dat_o   (dat_o[7:0])          //|>-o
   ,.inta_o  (inta_o)              //|>-o
   ,.mosi_o  (MOSI_O_IPX)          //|>-o
   ,.sck_o   (SCK_O_IPX)           //|>-o
   ,.ss_o    (SS_O_IPX)            //|>-o
   );
```

EMBEDDED FUNCTION – Instance - JSON

&Instance function has another way to use: **JSON** direct instance --- take **JSON** as a module to instance, like:

&Instance my_test_design.JSON u_my_test_design;

Or:

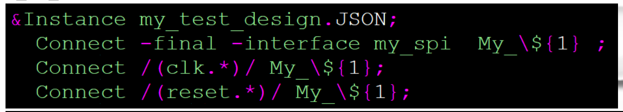
&Instance my_test_design.JSON #(.param0(xxx), .param1(yy) ..) u_my_test_design;

Or no Instance name: as

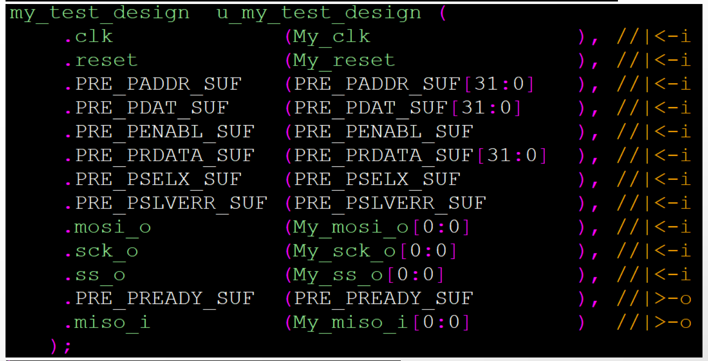
&Instance my_test_design;

NOTE:

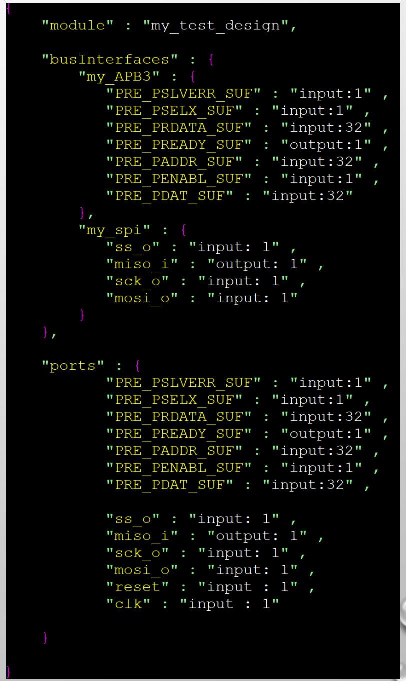
- **JSON** file name can be identical or different to module, module name will be defined by JSON's "module" field;
  - if JSON has no "module" field then file name will be used;
- Such JSON file normally has 3 fields:
  - "module" for top module name,
  - "busInterfaces" for all interface groups;
  - "ports" for all input/output ports;
- When instancing from **JSON** file, all interfaces defined in the file will be automatically updated into internal interface list;
  - so you can use those interfaces directly;
- Other requirements/functions are common for &Instance;



```
&Instance my_test_design.JSON;
  Connect -final -interface my_spi  My_\${1} ;
  Connect /(clk.*)/ My_\${1};
  Connect /(reset.*)/ My_\${1};
```



```
my_test_design  u_my_test_design (
    .clk              (My_clk                  ),  //|<-i
    .reset            (My_reset                ),  //|<-i
    .PRE_PADDR_SUF    (PRE_PADDR_SUF[31:0]     ),  //|<-i
    .PRE_PDAT_SUF     (PRE_PDAT_SUF[31:0]      ),  //|<-i
    .PRE_PENABL_SUF   (PRE_PENABL_SUF          ),  //|<-i
    .PRE_PRDATA_SUF   (PRE_PRDATA_SUF[31:0]    ),  //|<-i
    .PRE_PSELX_SUF    (PRE_PSELX_SUF           ),  //|<-i
    .PRE_PSLVERR_SUF  (PRE_PSLVERR_SUF         ),  //|<-i
    .mosi_o           (My_mosi_o[0:0]          ),  //|<-i
    .sck_o            (My_sck_o[0:0]           ),  //|<-i
    .ss_o             (My_ss_o[0:0]            ),  //|<-i
    .PRE_PREADY_SUF   (PRE_PREADY_SUF          ),  //|>-o
    .miso_i           (My_miso_i[0:0]          )   //|>-o
   );
```



```
{
    "module" : "my_test_design",

    "busInterfaces" : {
        "my_APB3" : {
           "PRE_PSLVERR_SUF" : "input:1" ,
           "PRE_PSELX_SUF" : "input:1" ,
           "PRE_PRDATA_SUF" : "input:32" ,
           "PRE_PREADY_SUF" : "output:1" ,
           "PRE_PADDR_SUF" : "input:32" ,
           "PRE_PENABL_SUF" : "input:1" ,
           "PRE_PDAT_SUF" : "input:32"
        },
        "my_spi" : {
           "ss_o" : "input: 1" ,
           "miso_i" : "output: 1" ,
           "sck_o" : "input: 1" ,
           "mosi_o" : "input: 1"
        }
    },

    "ports" : {
           "PRE_PSLVERR_SUF" : "input:1" ,
           "PRE_PSELX_SUF" : "input:1" ,
           "PRE_PRDATA_SUF" : "input:32" ,
           "PRE_PREADY_SUF" : "output:1" ,
           "PRE_PADDR_SUF" : "input:32" ,
           "PRE_PENABL_SUF" : "input:1" ,
           "PRE_PDAT_SUF" : "input:32" ,

           "ss_o" : "input: 1" ,
           "miso_i" : "output: 1" ,
           "sck_o" : "input: 1" ,
           "mosi_o" : "input: 1" ,
           "reset" : "input : 1" ,
           "clk" : "input : 1"

    }
}
```

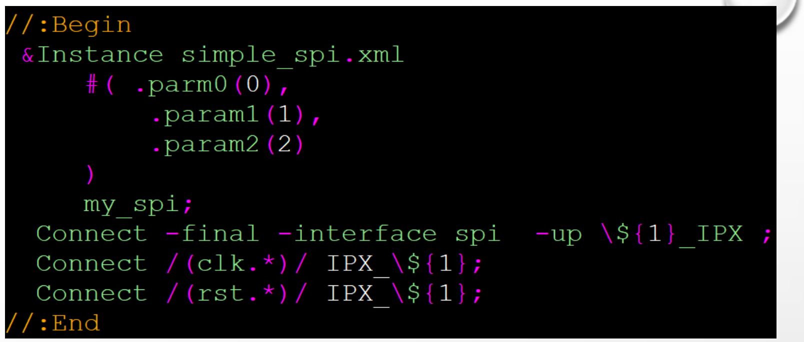
EMBEDDED FUNCTION – Instance - Parameters

&Instance function supports multi-line parameter， but has special requirements, like：

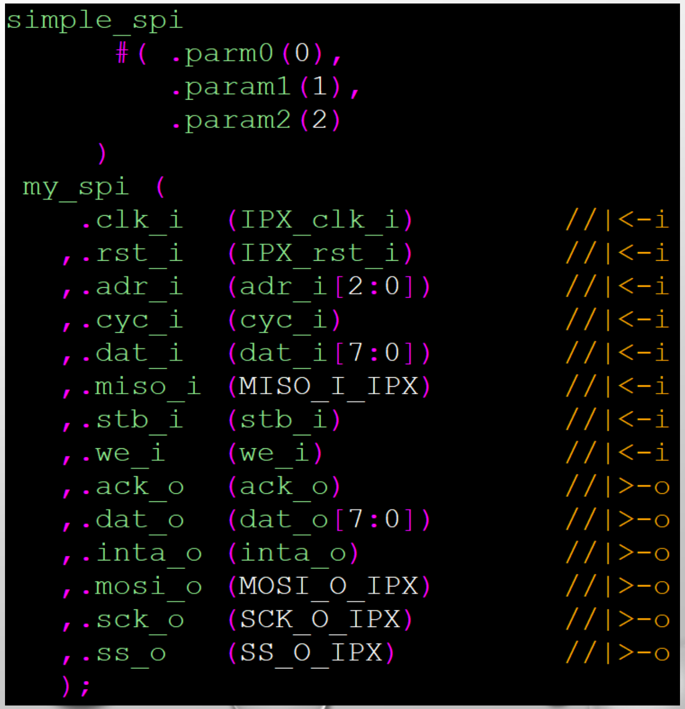
&Instance simple_spi.xml
#(.parm0(0),
.param1(1),
.param2(2)
)
my_spi;

NOTE：

- When Instancing with multi-line parameter, Instance command must be 3 parts:
    - 1st line for module or IPXACT name, and has **no “;”** ;
    - Second line to the line ending of **“)”** is for all parameters;
    - Last line is instance name, ending with **“;”** ;



```
//:Begin
 &Instance simple_spi.xml
     #( .parm0(0),
         .param1(1),
         .param2(2)
     )
     my_spi;
  Connect -final -interface spi  -up \${1}_IPX ;
  Connect /(clk.*)/ IPX_\${1};
  Connect /(rst.*)/ IPX_\${1};
//:End
```



```
simple_spi
      #( .parm0(0),
         .param1(1),
         .param2(2)
      )
 my_spi (
    .clk_i   (IPX_clk_i)         //|<-i
   ,.rst_i   (IPX_rst_i)         //|<-i
   ,.adr_i   (adr_i[2:0])        //|<-i
   ,.cyc_i   (cyc_i)             //|<-i
   ,.dat_i   (dat_i[7:0])        //|<-i
   ,.miso_i  (MISO_I_IPX)        //|<-i
   ,.stb_i   (stb_i)             //|<-i
   ,.we_i    (we_i)              //|<-i
   ,.ack_o   (ack_o)             //|>-o
   ,.dat_o   (dat_o[7:0])        //|>-o
   ,.inta_o  (inta_o)            //|>-o
   ,.mosi_o  (MOSI_O_IPX)        //|>-o
   ,.sck_o   (SCK_O_IPX)         //|>-o
   ,.ss_o    (SS_O_IPX)          //|>-o
   );
```

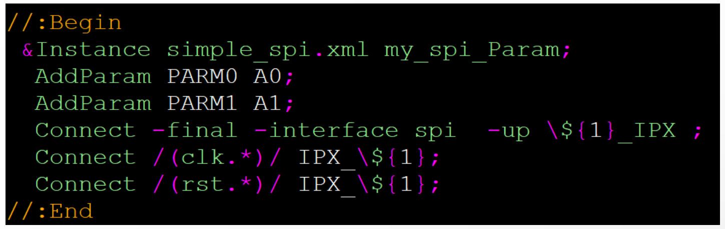
EMBEDDED FUNCTION – AddParam

AddParam function can be used add to define a parameter for an instance , like：

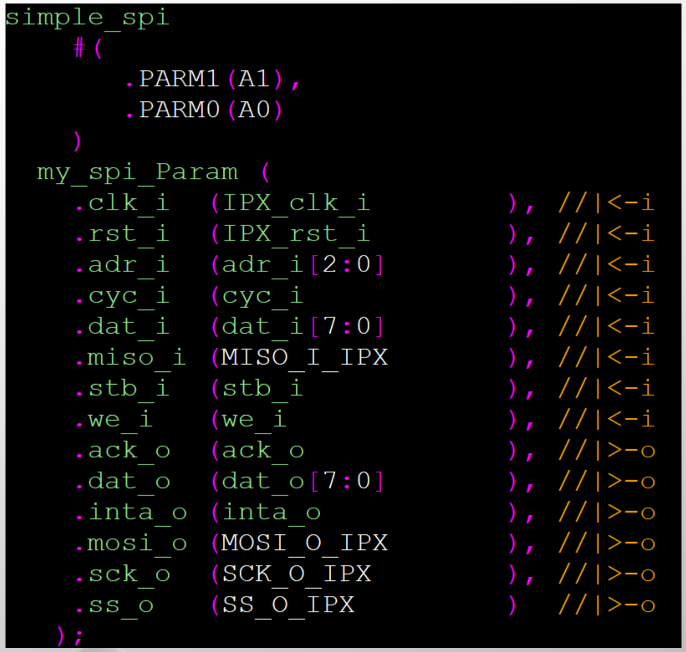
&Instance simple_spi.xml my_spi_Param;
AddParam PARM0 A0;
AddParam PARM1 A1;
Connect …

NOTE：

- One AddParam line for one parameter;
- Multi-line for multi-parameter;
- It's better placed after instance line and before Connect line;
- **AddParam** can't be used along with multi-parameter of **Instance** function;
  - Tool only use AddParam list but ignore others
- This Function is another way to support muli-paramater



```
//:Begin
 &Instance simple_spi.xml my_spi_Param;
  AddParam PARM0 A0;
  AddParam PARM1 A1;
  Connect -final -interface spi  -up \${1}_IPX ;
  Connect /(clk.*)/ IPX_\${1};
  Connect /(rst.*)/ IPX_\${1};
//:End
```



```
simple_spi
    #(
        .PARM1(A1),
        .PARM0(A0)
    )
  my_spi_Param (
    .clk_i   (IPX_clk_i         ), //|<-i
    .rst_i   (IPX_rst_i         ), //|<-i
    .adr_i   (adr_i[2:0]        ), //|<-i
    .cyc_i   (cyc_i             ), //|<-i
    .dat_i   (dat_i[7:0]        ), //|<-i
    .miso_i  (MISO_I_IPX        ), //|<-i
    .stb_i   (stb_i             ), //|<-i
    .we_i    (we_i              ), //|<-i
    .ack_o   (ack_o             ), //|>-o
    .dat_o   (dat_o[7:0]        ), //|>-o
    .inta_o  (inta_o            ), //|>-o
    .mosi_o  (MOSI_O_IPX        ), //|>-o
    .sck_o   (SCK_O_IPX         ), //|>-o
    .ss_o    (SS_O_IPX          )  //|>-o
   );
```

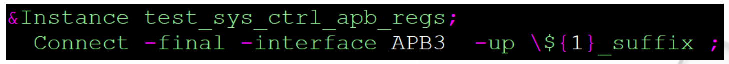
EMBEDDED FUNCTION – Connect

This function must be working along with **&Instance**, to achieve module's port connections. The function support regular expression for name matching , also support signal grouping by interface (interface can be standard AMBA bus, or manual defined --- as following intro), like:

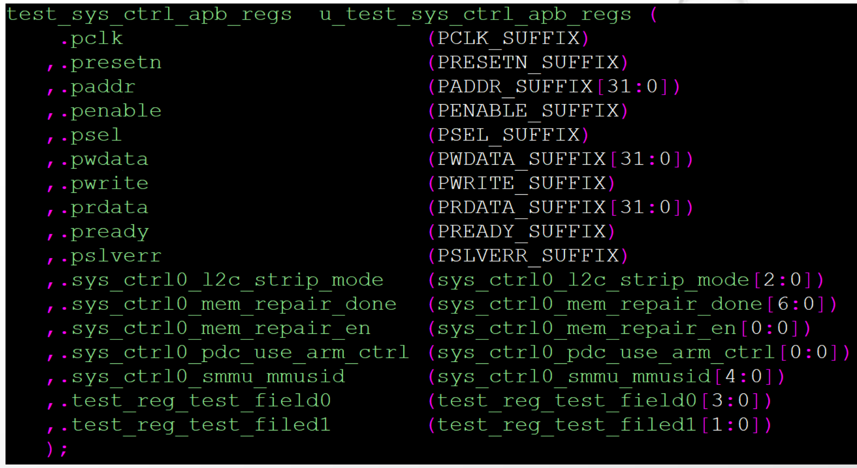
&Instance NV_NVDLA_CMAC_CORE_MAC_mul u_mul_0
&Connect exp_sft exp_sft_00[3:0];
&Connect /op_a_(\w*)/ wt_actv_\${1}0
&Connect -input /op_b_(.*)/ dat_actv_${1}0;
&Connect -final -interface APB3 -up ${1}_${suffix} ; //connect APB3 bus to wires has "_$suffix", and all wires upcased

NOTE:

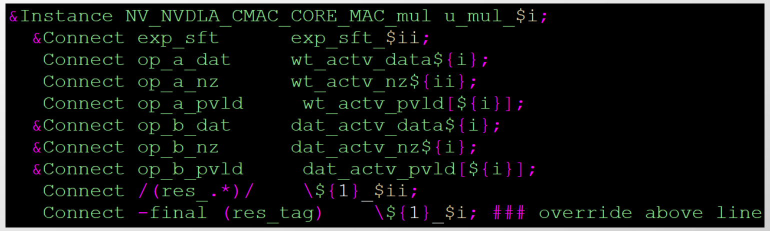
- Must follow &Instance line, no blank line from Instance line (blank line means "ending") ;
- Can control if only apply to input port(-input) or out port(-output);
- Regular express is native format, with 2 strings :
  - 1st is match pattern for port name;
    - Has "/" or has no "/" will get same result;
  - 2nd is name change with **$n** supported;
  - Support variable in express, like **$var**;
  - Please add **"\"** for regular express matched pattern **(\$1, \$2**);
  - Please add "**{}**" on variable to avoid any mistake;
- The wire going to connect can be upcase (-up) or lowcase (-low);
  - But keep in mind: **-low** is higher priority then-up, only –low action if both enabled.
- subsequent line's command will override previous lines;
- But override will be disabled if you enabled with "**-final**"
  - note: **-final** is highly recommended in most case
- If you want grouping by interface, then just use "**-interface intf_name**"
  - But please make sure interface does exist!
    - Default only standard AMBS bus exist
  - If need other interface, you need to manually add--- as following intro;
  - Interface default is "slave" mode --- can be changed by "**-master**" option;



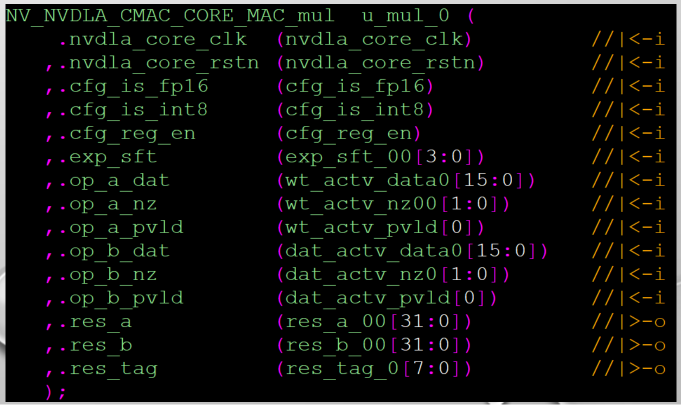
```
&Instance test_sys_ctrl_apb_regs;
  Connect -final -interface APB3  -up \${1}_suffix ;
```



```
test_sys_ctrl_apb_regs  u_test_sys_ctrl_apb_regs (
    .pclk                         (PCLK_SUFFIX)
   ,.presetn                      (PRESETN_SUFFIX)
   ,.paddr                        (PADDR_SUFFIX[31:0])
   ,.penable                      (PENABLE_SUFFIX)
   ,.psel                         (PSEL_SUFFIX)
   ,.pwdata                       (PWDATA_SUFFIX[31:0])
   ,.pwrite                       (PWRITE_SUFFIX)
   ,.prdata                       (PRDATA_SUFFIX[31:0])
   ,.pready                       (PREADY_SUFFIX)
   ,.pslverr                      (PSLVERR_SUFFIX)
   ,.sys_ctrl0_l2c_strip_mode     (sys_ctrl0_l2c_strip_mode[2:0])
   ,.sys_ctrl0_mem_repair_done    (sys_ctrl0_mem_repair_done[6:0])
   ,.sys_ctrl0_mem_repair_en      (sys_ctrl0_mem_repair_en[0:0])
   ,.sys_ctrl0_pdc_use_arm_ctrl   (sys_ctrl0_pdc_use_arm_ctrl[0:0])
   ,.sys_ctrl0_smmu_mmusid        (sys_ctrl0_smmu_mmusid[4:0])
   ,.test_reg_test_field0         (test_reg_test_field0[3:0])
   ,.test_reg_test_filed1         (test_reg_test_filed1[1:0])
   );
```



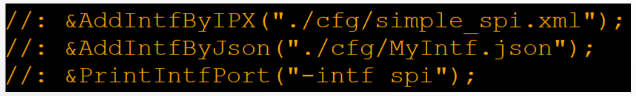
```
&Instance NV_NVDLA_CMAC_CORE_MAC_mul u_mul_$i;
  &Connect exp_sft      exp_sft_$ii;
   Connect op_a_dat     wt_actv_data${i};
   Connect op_a_nz      wt_actv_nz${ii};
   Connect op_a_pvld     wt_actv_pvld[${i}];
  &Connect op_b_dat     dat_actv_data${i};
  &Connect op_b_nz      dat_actv_nz${i};
  &Connect op_b_pvld     dat_actv_pvld[${i}];
   Connect /(res_.*)/    \${1}_$ii;
   Connect -final (res_tag)    \${1}_$i; ### override above line
```



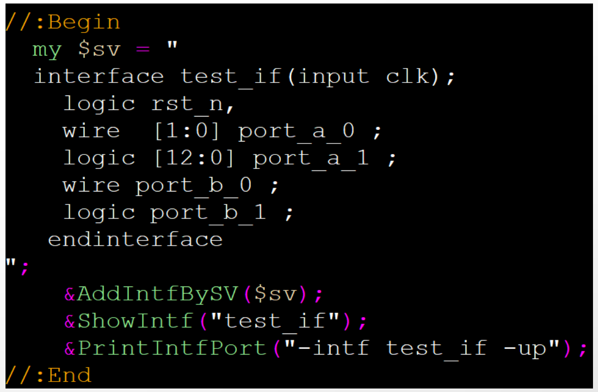
```
NV_NVDLA_CMAC_CORE_MAC_mul  u_mul_0 (
    .nvdla_core_clk  (nvdla_core_clk)          //|<-i
   ,.nvdla_core_rstn (nvdla_core_rstn)         //|<-i
   ,.cfg_is_fp16     (cfg_is_fp16)             //|<-i
   ,.cfg_is_int8     (cfg_is_int8)             //|<-i
   ,.cfg_reg_en      (cfg_reg_en)              //|<-i
   ,.exp_sft         (exp_sft_00[3:0])         //|<-i
   ,.op_a_dat        (wt_actv_data0[15:0])     //|<-i
   ,.op_a_nz         (wt_actv_nz00[1:0])       //|<-i
   ,.op_a_pvld       (wt_actv_pvld[0])         //|<-i
   ,.op_b_dat        (dat_actv_data0[15:0])    //|<-i
   ,.op_b_nz         (dat_actv_nz0[1:0])       //|<-i
   ,.op_b_pvld       (dat_actv_pvld[0])        //|<-i
   ,.res_a           (res_a_00[31:0])          //|>-o
   ,.res_b           (res_b_00[31:0])          //|>-o
   ,.res_tag         (res_tag_0[7:0])          //|>-o
   );
```

EMBEDDED FUNCTION – Interface

There are several functions for Interface management, you can use them to add interface from RTL file, JSON file, or Perl hash, or embedded SV code, then subsequent code can use these interfaces to print or connect, or do whatever you want, like:

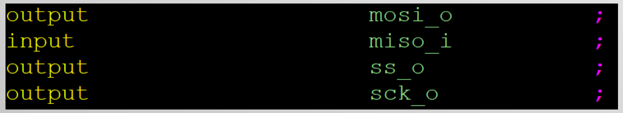


```
//: &AddIntfByIPX("./cfg/simple_spi.xml");
//: &AddIntfByJson("./cfg/MyIntf.json");
//: &PrintIntfPort("-intf spi");
```

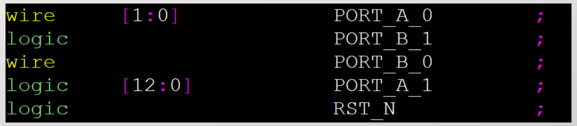


```
//:Begin
  my $sv = "
  interface test_if(input clk);
    logic rst_n,
    wire   [1:0] port_a_0 ;
    logic [12:0] port_a_1 ;
    wire port_b_0 ;
    logic port_b_1 ;
   endinterface
";
    &AddIntfBySV($sv);
    &ShowIntf("test_if");
    &PrintIntfPort("-intf test_if -up");
//:End
```

You'll get code as:



```
output                      mosi_o              ;
input                       miso_i              ;
output                      ss_o                ;
output                      sck_o               ;
```



```
wire      [1:0]                PORT_A_0          ;
logic                          PORT_B_1          ;
wire                           PORT_B_0          ;
logic     [12:0]               PORT_A_1          ;
logic                          RST_N             ;
```

NOTE: detail usage please refer to sample code, or user guide released later

EMBEDDED FUNCTION – Interface

- Add or modify embedded interfaces

AddIntfByIPX	Read in PXACT file， Parse IPXACT(xml)file for interface and add all of them into embedded list	**&AddIntfByIPX("./cfg/simple_spi.xml");**
AddIntfByJson	Read in JSON file， parse JSON signal define， add all of then as an interface into embedded list	**&AddIntfByJson("MyIntf.json");**
AddIntfByRTL	Read in RTL file, parse all port, filter with keyword(optional)， add as an interface into embedded list	**&AddIntfByRTL("MyIntf.v“. “MyIntfName”, “key_word”);**
AddIntfBySV	use SystemVerilog code to define interfaces, add all of them into embedded list	**&AddIntfBySV($sv_code);**
AddIntfByHash	Add Perl hash as an interface, into embedded list	**&AddIntfByHash(\%MyIntf, “MyIntf”, “key_name”);**
AddIntfByName	Add 1 port into an existing embedded interface	**&AddIntfByName("clk “， ”input:1", "intf_name");**
RmIntfPort	remove a port from an existing embedded interface	**&RmIntfPort(“clk“, “intf_name”);**

- for 3rd part or inhouse IP integration

PrintIntfPort	Print out an interface as signal list, according to config options	**&PrintIntfPort("-intf MyIntf -awd 18 -dwd 32 -pre Test_ -low-port -master -end ,");**
PrintAmbaBus	Print out standard AMBS bus signal list, according to config options	**&PrintAmbaBus("-type test_APB3 -awd 18 -dwd 32 -pre Test_ -suf _End -up -wire -end ,");**
ShowIntf	Show and interface into a JSON file (for debug)	**&ShowIntf(“intf_name”);**

EMBEDDED FUNCTION – IPXACT/JSON

ReadIPX	Read in IPXACT file, parse IPXACT(xml) interface info， add all of them io embedded list	**&ReadIPX("./cfg/simple_spi.xml");**
ShowIPXIntf	Read in IPXACT file, parse IPXACT(xml) interface info, and write interface in a simple way into a file --- for debug	**&ShowIPXIntf(“./cfg/ipx.xml”);**
TransIPX	Read in IPXACT file, parse IPXACT(xml) module/interface/port info, then write all of them into a JSON file--- for debug or integrations	**&TransIPX(“ipx.xml”);**
ExptIntf	Export an interface from embedded list to exporting list --- finally to JSON file for up-level integration	**&ExptIntf("-intf_name APB3 -name My_APB3 -upcase -prefix Pre_ -master");**
ExptPort	Add a new port to exporting list --- finally to JSON file for up-level integration	**&ExptPort("clk", "input:1", "1");**
RmPort	Remove a port from exporting list --- will not to JSON file	**&RmPort("clk");**
GenModJson	Generate a JSON file with module name, all interfaces, and all ports in, can be used for up-level integration Note: all ports of current module will be exported automatically	**&GenModJson();**
~~**GenModIPX**~~	~~Generate a standard IPXACT file (xml) as what RTL looks like~~	~~**&GenIPX(“my_design", "MyCorp");**~~ ~~**On plan but not start**~~**--abolished**

For module/IP/SoC integration

EMBEDDED FUNCTION – GenModJson

&GenModJson function can generate a **JSON** file with necessary information in: module name, all interfaces, all ports, which can be used as an integration source for up-level design, usage：

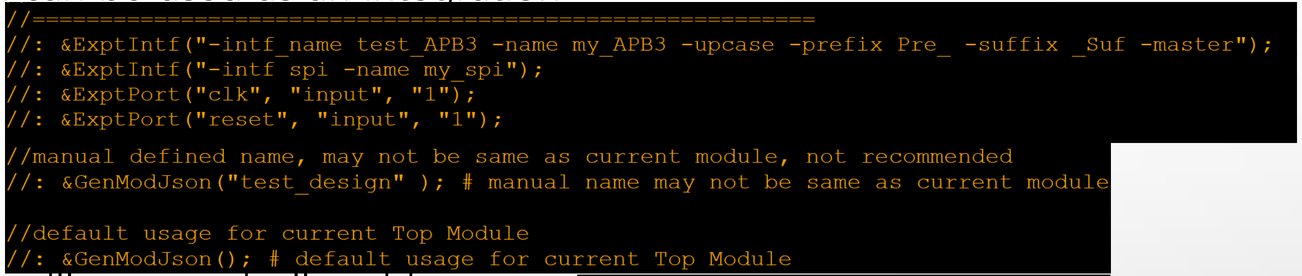
&GenModJson();

Or：

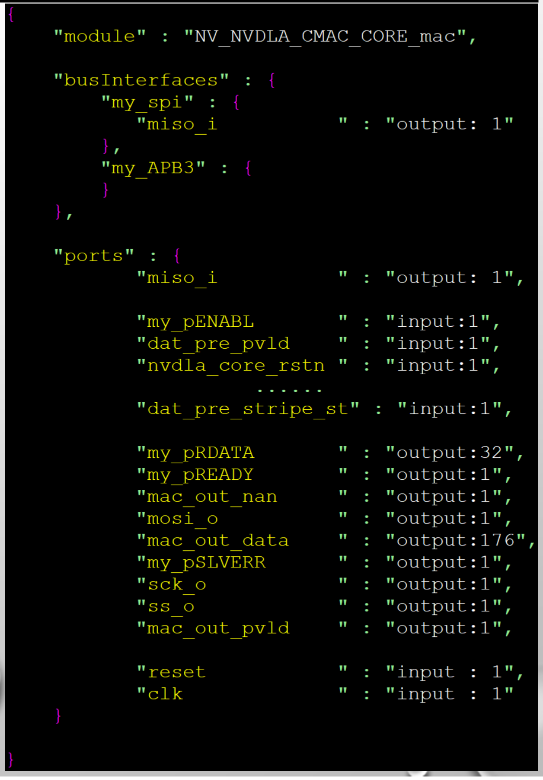
&GenModJson(“my_design”); .

NOTE：

- Suggest **not to** add design name, as tool will automatically add current module name in **JSON** file;
- Such JSON file has at least 3 fields:
  - “module” for top module name,
  - “busInterfaces” for all interfaces, if there is any manual exported list;
    - by **&ExptIntf()** function;
  - “ports” for all input/output ports
    - **all ports defined in RTL of current top module will in JSON**;
    - And any port added by **&ExptPort()** function;
- If a port of an interface is not on current module’s port, a warning will be on screen and such port will not in JSON file;
  - You need to double check input source code and generated JSON file to see what code need to update;



```
//============================================================
//: &ExptIntf("-intf_name test_APB3 -name my_APB3 -upcase -prefix Pre_ -suffix _Suf -master");
//: &ExptIntf("-intf spi -name my_spi");
//: &ExptPort("clk", "input", "1");
//: &ExptPort("reset", "input", "1");

//manual defined name, may not be same as current module, not recommended
//: &GenModJson("test_design" ); # manual name may not be same as current module

//default usage for current Top Module
//: &GenModJson(); # default usage for current Top Module
```



```
{
    "module" : "NV_NVDLA_CMAC_CORE_mac",

    "busInterfaces" : {
        "my_spi" : {
           "miso_i             " : "output: 1"
        },
        "my_APB3" : {
        }
    },

    "ports" : {
           "miso_i             " : "output: 1",

           "my_pENABL          " : "input:1",
           "dat_pre_pvld       " : "input:1",
           "nvdla_core_rstn    " : "input:1",
                   ......
           "dat_pre_stripe_st" : "input:1",

           "my_pRDATA          " : "output:32",
           "my_pREADY          " : "output:1",
           "mac_out_nan        " : "output:1",
           "mosi_o             " : "output:1",
           "mac_out_data       " : "output:176",
           "my_pSLVERR         " : "output:1",
           "sck_o              " : "output:1",
           "ss_o               " : "output:1",
           "mac_out_pvld       " : "output:1",

           "reset              " : "input : 1",
           "clk                " : "input : 1"
    }

}
```

EXTENDED API

- For inhouse design development
- & IP a/o SOC integration

ClkGen	Generate Clock Generation Module， including MUX, DIV, ICG, with parameters	**Basic flow done but need inhouse development**
RstGen	Generate Reset Generation Module, with parameters	**Basic flow done but need inhouse development**
PMUGen	Generate PMU module, with parameters	**Basic flow done but need inhouse development**
FuseGen	Generate Fuse module, with parameters	**Basic flow done but need inhouse development**
AsyncIntfGen	Generate Async interface, with or without fifo	**Basic flow done but need inhouse development**
FifoGen	Generate various fifo design, according to different parameters	**Basic flow done but need inhouse development**
MemGen	Generate sram design, based on foundary config and input constraints	**Basic flow done but need inhouse development**
…	…	…

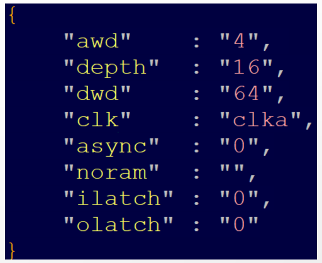
- Need parameter config file as JSON
- And design template file with ePerl
  (details refer to source code and test file)

EXTEND FUNCTION SAMPLE – FifoGen

This tool has a few extend functions, to generate design module first, then can be used in following RTL code directly, for sample Instance & wire connections. Those functions' usage is very similar, all need 3 inputs, FifoGen as sample:

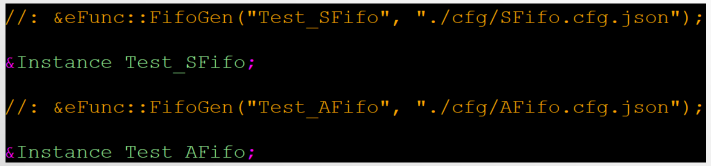
- Need module name (mod_name);
- Need parameter config file (JSON file);
- Need design template file(design template file);
  - This file in not exposed, need to prepared in tool's dir;
  - This file is pure in-house developed logic, nothing to do with this tool

parameter config (JSON):



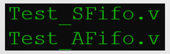
```
{
    "awd"    : "4",
    "depth"  : "16",
    "dwd"    : "64",
    "clk"    : "clka",
    "async"  : "0",
    "noram"  : "",
    "ilatch" : "0",
    "olatch" : "0"
}
```

Usage:



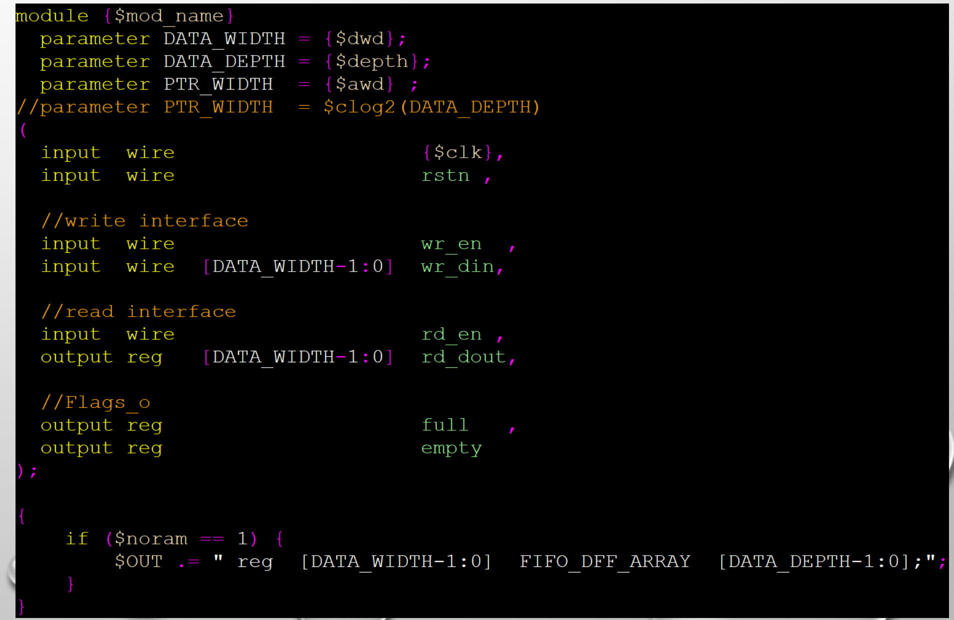
```
//: &eFunc::FifoGen("Test_SFifo", "./cfg/SFifo.cfg.json");

&Instance Test_SFifo;

//: &eFunc::FifoGen("Test_AFifo", "./cfg/AFifo.cfg.json");

&Instance Test_AFifo;
```

Design module generated:



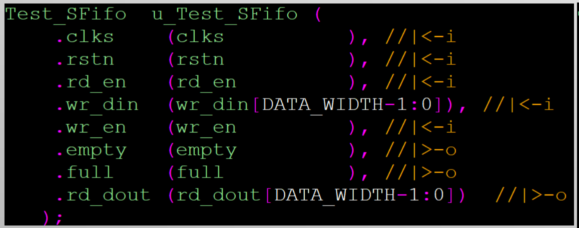
```
Test_SFifo.v
Test_AFifo.v
```

design template file:

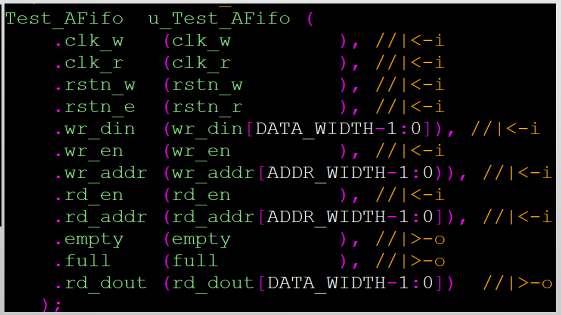


```
module {$mod_name}
  parameter DATA_WIDTH = {$dwd};
  parameter DATA_DEPTH = {$depth};
  parameter PTR_WIDTH  = {$awd} ;
//parameter PTR_WIDTH  = $clog2(DATA_DEPTH)
(
  input  wire                    {$clk},
  input  wire                    rstn ,

  //write interface
  input  wire   [DATA_WIDTH-1:0]  wr_en  ,
  input  wire   [DATA_WIDTH-1:0]  wr_din,

  //read interface
  input  wire                    rd_en ,
  output reg    [DATA_WIDTH-1:0]  rd_dout,

  //Flags_o
  output reg                     full   ,
  output reg                     empty
);

{
    if ($noram == 1) {
        $OUT .= " reg  [DATA_WIDTH-1:0]  FIFO_DFF_ARRAY  [DATA_DEPTH-1:0];";
    }
}
```

Generated RTL code:



```
Test_SFifo  u_Test_SFifo (
    .clks    (clks              ), //|<-i
    .rstn    (rstn              ), //|<-i
    .rd_en   (rd_en             ), //|<-i
    .wr_din  (wr_din[DATA_WIDTH-1:0]), //|<-i
    .wr_en   (wr_en             ), //|<-i
    .empty   (empty             ), //|>-o
    .full    (full              ), //|>-o
    .rd_dout (rd_dout[DATA_WIDTH-1:0])  //|>-o
  );
```

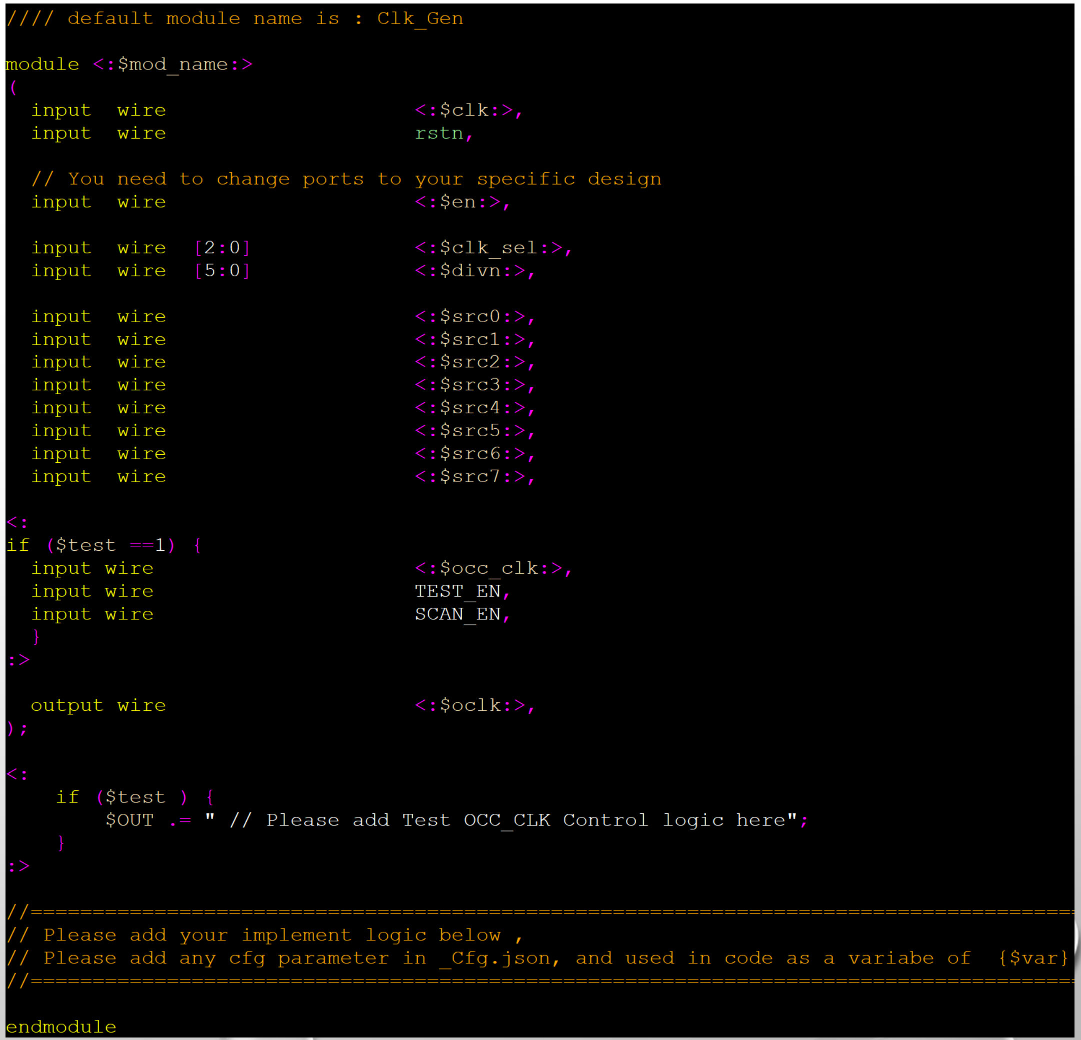


```
Test_AFifo  u_Test_AFifo (
    .clk_w   (clk_w             ), //|<-i
    .clk_r   (clk_r             ), //|<-i
    .rstn_w  (rstn_w            ), //|<-i
    .rstn_e  (rstn_r            ), //|<-i
    .wr_din  (wr_din[DATA_WIDTH-1:0]), //|<-i
    .wr_en   (wr_en             ), //|<-i
    .wr_addr (wr_addr[ADDR_WIDTH-1:0)), //|<-i
    .rd_en   (rd_en             ), //|<-i
    .rd_addr (rd_addr[ADDR_WIDTH-1:0]), //|<-i
    .empty   (empty             ), //|>-o
    .full    (full              ), //|>-o
    .rd_dout (rd_dout[DATA_WIDTH-1:0])  //|>-o
  );
```

EXTEND FUNCTION – DESIGN TEMPLATE FILE

This tool's any extended function need at least 1 design template file (how many template file is totally defined by extend function's implementation), these template file use **ePerl** syntax, simple introduction:

- Most code or syntax is **Verilog**, it's more similar to a Verilog RTL file;
- Please use **<:** & **:>** when you need variable;
- variable is Perl stye, like $var, EX: **<:$my_sig:>**;
- If you need any control code, please use **<:** & **:>** too;
- Control code only support **Perl** syntax(**no Python** so far);
- Suggest to use **$VOUT .=** but not **print** in control code;
    - ◆ It's more simple;



```
//// default module name is : Clk_Gen

module <:$mod_name:>
(
  input  wire                     <:$clk:>,
  input  wire                     rstn,

  // You need to change ports to your specific design
  input  wire                     <:$en:>,

  input  wire  [2:0]              <:$clk_sel:>,
  input  wire  [5:0]              <:$divn:>,

  input  wire                     <:$src0:>,
  input  wire                     <:$src1:>,
  input  wire                     <:$src2:>,
  input  wire                     <:$src3:>,
  input  wire                     <:$src4:>,
  input  wire                     <:$src5:>,
  input  wire                     <:$src6:>,
  input  wire                     <:$src7:>,

<:
if ($test ==1) {
  input wire                      <:$occ_clk:>,
  input wire                      TEST_EN,
  input wire                      SCAN_EN,
  }
:>

  output wire                     <:$oclk:>,
);

<:
    if ($test ) {
        $OUT .= " // Please add Test OCC_CLK Control logic here";
    }
:>

//=====================================================================================
// Please add your implement logic below ,
// Please add any cfg parameter in _Cfg.json, and used in code as a variabe of  {$var}
//=====================================================================================

endmodule
```

OTHER FUNCTIONS

CallCmd	call Shell/Perl/Python command	**&CallCmd("create_design.py -n my_design -d 32 -a 18");**
DTIWire	Generate DTI interface as wire signals， name prefix and data width is necessary	**&DTIWire("top2me", 512);**
DTISlave	Generate DTI interface as slave signals， name prefix and data width is necessary	**&DTISlave("top2me", 512);**
DTIMaster	Generate DTI interface as master signals， name prefix and data width is necessary	**&DTIMaster"top2me", 512);**
…	…	…

- For inhouse design development

Thanks PyGear for the name of "DTI"

Verilog is the King

Connection is what you need

Flexibility is really helpful

Thanks and Notice

Thanks NVIDIA for giving me the chance to know how powerful Perl is to run a big ASIC factory;
Thanks NVIDIA's VIVA to let me know how Perl can make Verilog easy, interesting and amazing;

Please note this tool was developed from scratch during the special spring time in Shanghai in 2022;
The things related to NVIDIA are:

- several function(Instance & Connect) names are identical;
- several HDL files of open sourced NVDLA are used to be test source