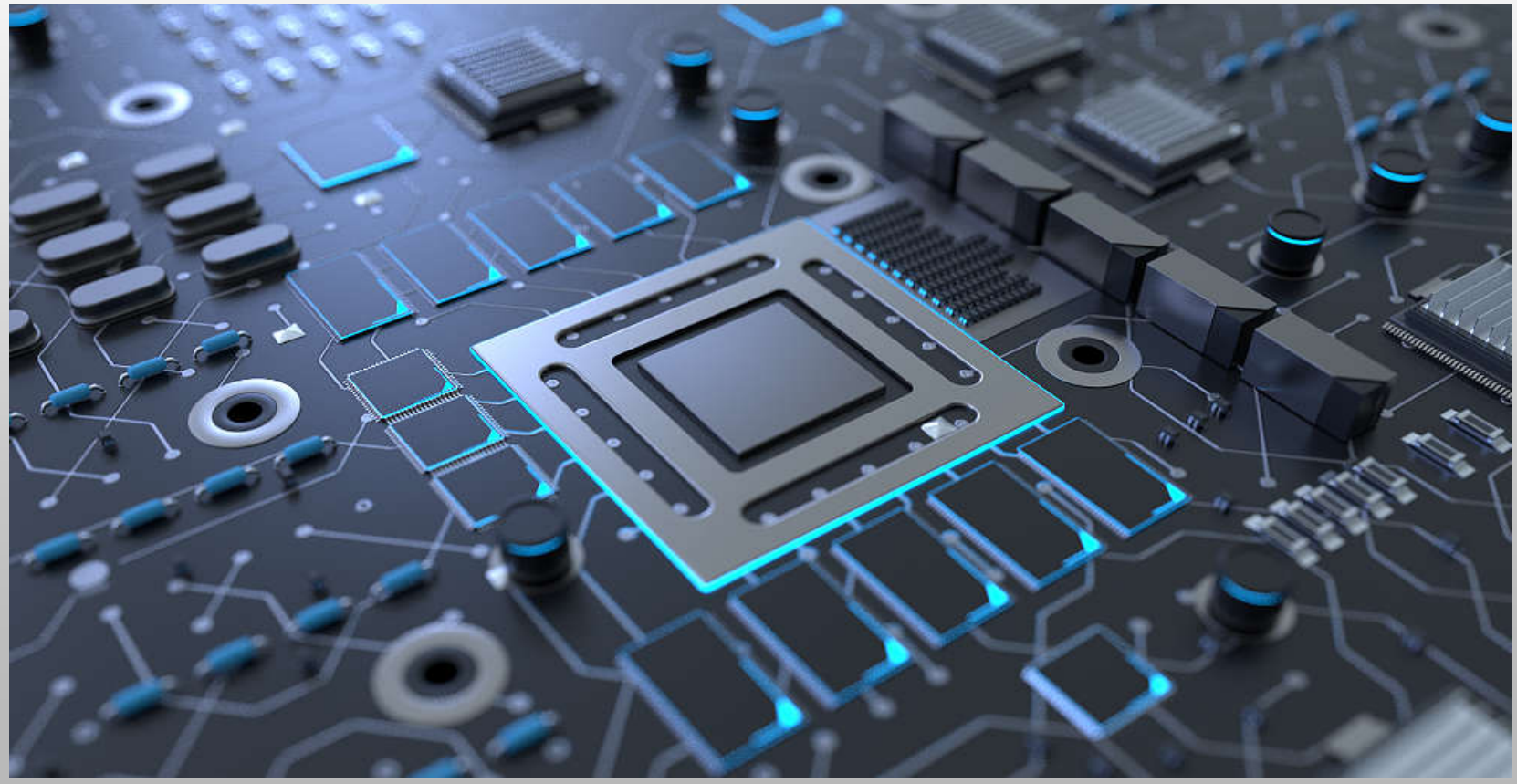
HDLGEN 工具介绍

WILSON CHEN

2022-2023



目录

HDLGEN是什么

HDLGen是一个HDL生成工具，支持在Verilog里内嵌Perl或Python script来帮助快速、高效地生成期望的设计，支持Perl或者Python的所有数据结构和语法，有若干内嵌函数来提高效率，也支持Perl的扩展API(Python API扩展目前还不支持)，通过内嵌script和API来减少手动工作、提高开发效率和降低出错几率。 本工具支持自动instance,自动信号生成(instance & logic)，自定义电路(模块生成)。 本工具可以实现EMACS veirlog-mode的所有功能，另外再支持正则表达式、IPXACT、XML、interface、JSON、Hash等输入，并且使用方法和感觉是写HDL而不是HLS或者DSL,无论从功能还是上手容易度都比商业的SOC集成工具更高效、快捷。

约定以"//:"开始的行为单行Perl script;

约定以"//:Begin"为起始标志和"//:End"或者为结束标志的行为多行Perl script;

约定以"//#"开始的行为单行Python script;

约定以"//#Begin"为起始标志和"//#End"为结束标志的行为多行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
```

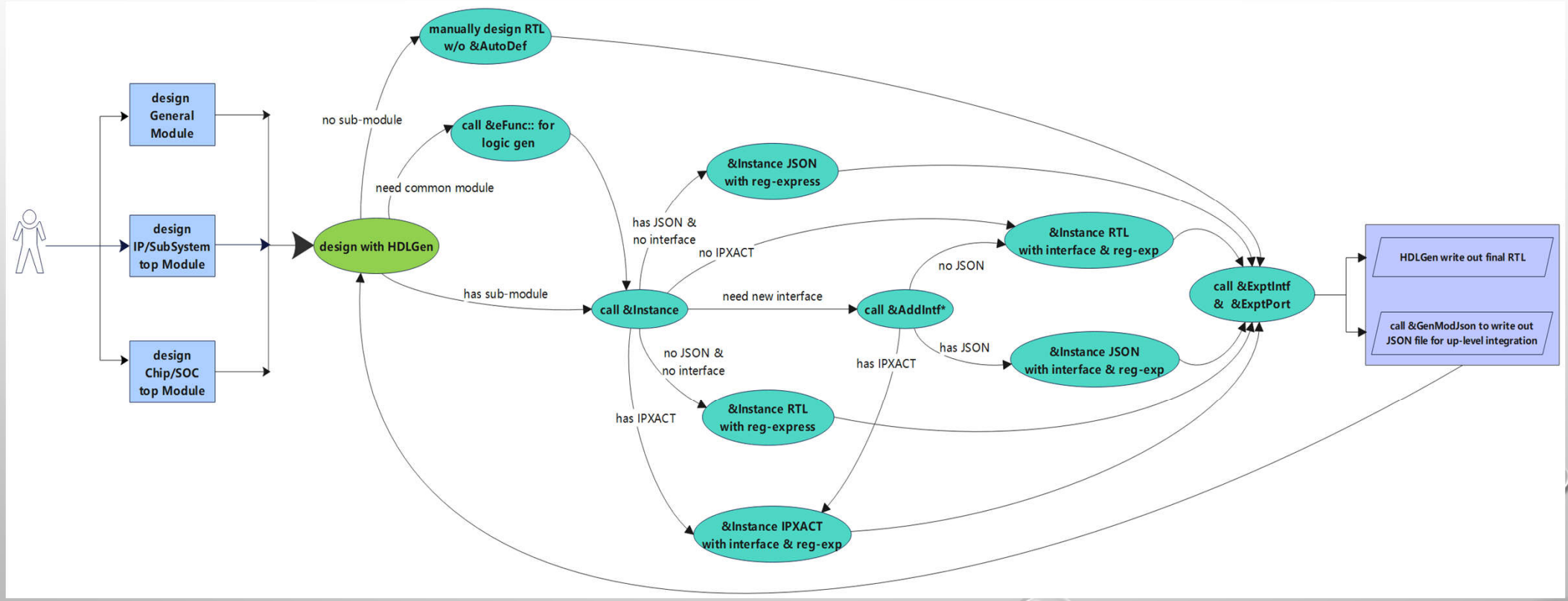
HDLGEN可以干什么

分类	具体功能
RTL组装和生成	• 从RTL或IPCAXT或JSON以Verilog的方式直接Instance module； • 用正则表达式来匹配端口名字，做信号连接； • 自动产生Instance端口信号的定义(reg & wire)，宽度自动学习、生成； • 自动产生logic(assign & always)的信号定义(还不完美); • 识别并报警任何Instance模块上没有被真实使用的信号； • 用内嵌的Perl或者Python生成任何想要的code(structure/for/generator)；
Interface使用和生成	• 从 IPXACT,JSON,RTL,SV code, Hash数组, *YAML(后续支持)*读入interface； • 向任何Interface追加port/signal； • 从任何Interface移除port/signal； • 以简洁模式打印显示Interface信号(用debug);
IPXACT/JSON使用和生成	• 读入标准IPXACT或JSON文件并新增Interface和Port； • 以简洁模式打印显示IPXACT内定义的所有interface(for debug) • 翻译IPXACT文件到JSON格式(for debug or integration) • 输出port/signal输出到JSON文件(for integragation) • 将当前顶层模块的Name/Interfaces/Ports输出到JSON文件(for integration) • ~~*将当前顶层模块输出到标准的IPXACT文件---已决定不支持*~~

功能逻辑产生	• 用内嵌的函数生成特定电路或模块，并且可定制
	  • Clk, Reset, Fuse, Pmu, Fifo, Async-interface, Memories 等；
	• *通过标准输入增加、定制私有的逻辑或模块(in development)*
	  • *以Verilog, JSON, YAML, EXCEL等为config输入.*
	  • *以一个简单的函数调用来输出code；*

工作流程

HDLGen可在不同的设计模块或任务上使用，从模块级到IP级，到子系统级，再到chip top或soc top级，大体流程如下图所示：



使用说明

HDLGen支持单个文件和列表文件输入，会一一生成同名的Verilog HDL，默认情况下只需要要一个输入文件的参数即可，比如：

HDLGen.pl -i my_design.src 会在当前目录生成 **my_design.v**
HDLGen.pl –f in_flist 会在当前目录生成 由**in_flist**里指定名字的**.v**

其他参数使用说明如下：

-u[usage] ：打印使用说明
-o[output] ：指定输出文件（通常不需要使用，除非想改文件名）
-d[debug] ：调试模式，会多上传一些中间文件，利于后续调试
-v[verbose]：调试模式，会在屏幕上打印及其多的执行信息，以利于flow调试

建议：

1. 默认模式只使用 **-i** 或者**-f**;
2. 如果出错则只加上调试模式(**-d**);

```
--- 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.pl -i HDL_Design.src
   --> ../HDLGen.pl -i HDL_Design.src -o HDL_Design_NewName.v ( default is HDL_Design.v )
   --> ../HDLGen.pl -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]
```

自由变量

我们可以在Verilog code的任意地方使用Perl的变量, 不过这些变量需要事先定义.

```
//: 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
```

注意：

- 这些变量可以直接内嵌在任意code中，并不需要加“**//:**”;
- 这些变量需要声明，类型必须是“**our**”，变量声明的行需要用“**//:**”;
- 这些变量必须用花括号包起来，比如 **${reset} ---**主要用于区别于Verilog内嵌函数（比如$display);

内嵌和扩展函数

HDLGen有若干内嵌函数，可以完成RTL生成、集成的各种功能。

注意：

- 调用函数有2种方法：**&函数名 参数**；或 **&函数名(参数)**，其中“**&**”是可选的；
- 函数的参数有2种形式：
  - 多个字符串用**”,”**分隔，需要严格按照参数列表的先后顺序保序；
  - 类似于shell命令行的 **-option** 参数值，不需要保序；
    - 具体使用参考各函数的具体定义
- 内嵌函数可以直接调用；
- 扩展API库函数需要加上前缀来调用：**&eFunc::**函数名(参数)

```
&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
```

```
//: &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;
```

```
//: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
```

内嵌函数- print

此工具支持2种**print**函数:

1. Perl 和 Python的标准的 “**print**”;
2. Verilog code print函数“**vprintl**”;

Usage：

//: print(“string to print\n”);

//: vprintl(“string to print\n”);

```
//: 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
```

注意：

- 2个函数的区别是:
  - **vprintl**只会输出到RTL code,永远不会打印到屏幕;
  - Print则有可能输出到屏幕;
- **Python** script只有 原生的**print** function.
- 多行输出(到屏幕或者RTL code),采用各自方法
  - 比如 **EOF … EOF** 或 **""" … """**

内嵌函数 - SRC

此函数是增加源文件搜索目录，多次使用依次增加. 当遇到所需要的源文件不在当前目录下的时候，就需要用这个函数来更新(增加)搜索目录（类似于+incr)以利于tool寻找到所需源文件(RTL或者其他）

Usage：

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

注意：

- 你可以在需要的地方使用完整全目录，但是有时候很可能并不方便(比如module instance）

内嵌函数 - AutoDef

此函数可以自动生成wire和reg 信号定义，也即可以直接写出功能HDL而不需要事先定义信号，不过目前只支持有限的简单语法，比如：

```
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]
```

用法： //: &AutoDef;

```
//|: &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                       ;
```

```
assign plic_int_vld[`INT_NUM_PLIC+15:0] = {pad_plic_int_vld[`INT_NUM_PLIC-1:0],14'b0,l2c_plic_ecc_int_vld,1'b0};
assign plic_int_cfg[`INT_NUM_PLIC+15:0] = {pad_plic_int_cfg[`INT_NUM_PLIC-1:0],16'b0};
//assign plic_int_cfg = {pad_plic_int_cfg[`INT_NUM_PLIC-1:0],16'b0};
assign plic_int_cfg_test_0 = {pad_plic_int_cfg[`INT_NUM_PLIC-1:0],16'b0};
assign plic_int_cfg_test_1 = {pad_plic_int_cfg[239:0],16'b0};
assign plic_int_cfg_test_2 = {pad_plic_int_cfg[240-1:0],16'b0};

assign plic_int_cfg_test_3 = pad_plic_int_cfg[239:0];

assign plic_int_cfg_test_4 = {pad_plic_int_cfg[240+2:0],16'b0};

assign plic_int_cfg_test_5 = pad_plic_int_cfg[NUM_INT-1:0];
assign plic_int_cfg_test_6 = {pad_plic_int_cfg[${NUM_INT_VAR}-1:0],16'b0};

assign plic_int_cfg_test_7 = {pad_plic_int_cfg[NUM_TOTAL-1:0],16'b0};
```

```
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
```

注意：

- **如果使用，必须放在所有wire和reg定义行之前；**
  - 建议使用
  - 但是不能完全依赖(如果有好主意请告诉我)
    - 功能不完美，是backup方案
  - 复杂logic建议尽量手动定义wire/reg信号

内嵌函数-AutoInstSig

此函数可以自动生成module instance的port所连接的信号定义，不过必须是采用“Instance”函数所instance的module才会被生成，采用Verilog原始instance语法的信号并不会被处理(后续可能会支持，看需求)

用法： //: &AutoInstSig;

注意：

- 可以在任意地方使用，但是建议放在Reg/Wire定义行附近;
- Port所连接的信号默认是wire类型，但HDLGen会分析logic语句来判断是否需要改成“reg”类型;
- 信号位宽会自动分析并识别；
- 如果信号已经被手动定义(wire或reg)，则自动化定义会被bypass;

```
//: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                ;
```

强烈建议与&Instance一起使用，省时省力

内嵌函数- AutoInstSig- AutoWarning

当**AutoInstSig**使能时，HDLGen会分析所有用Instance调用的sub-module，如果有任何port连接的端口信号没有输入源头或者没有输出去处，则此类信号会被以“Warning”形式打印到输出的RTL文件中去)，同时一个Warning message会在屏幕打印输出：

用法： //: &AutoInstSig;

注意：

- 此功能没有独立关键词定义，由**AutoInstSig**控制；
- 目前只有用Instance调用的sub-module才会被分析(Verilog instance module后续支持)；
- 信号名会根据instance order排序输出；
- 此功能也许不完美但是绝大多数情况下功能正确；

```
//: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 ---------------
```

内嵌函数 – Instance -Verilog

此函数用于instance sub module，基本语法与Verilog instance比较接近，比如：

&Instance NV_NVDLA_CMAC_CORE_MAC_mul u_mul_0

或者：

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

或者不需指明Instance name:

&Instance NV_NVDLA_CMAC_CORE_MAC

注意：

- 使用&Instance的时候可以不用加注释头“//:”或者“//:Begin” “//:End”，当然也可以使用注释头，不过不建议用单行注释头；
- Instance往往跟 Connect同时使用，比如：

```
&Instance NV_NVDLA_CMAC_CORE_MAC_mul u_mul_0(
  &Connect exp_sft exp_sft_00[3:0];
  &Connect /op_a_(.*)/  wt_actv_\${1}0;
  &Connect /op_b_(.*)/  dat_actv_\${1}0;
```

- 如果Instance只给出module名字，那么instance名字采用默认值：u_module
- 如果Instance后面不跟 &Connect，则此sub-module的port都会连接到相同名字的信号上去。
- 如果在Instance之前有使用AutoInstSig函数，则此sub-module的instance的port所连接的信号都会自动定义好；
- 如果在Instance之前没有使用AutoInstSig函数，则此module instance的port所连接的所有信号都会以注释行的形式紧跟在instance之后，以利于后续手工处理；

```
&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
   );
```

内嵌函数 - Instance - IPXACT

此函数有第二种用法，即可直接调用IPXACT file，当作module 定义来做instance, 比如：

&Instance simple_spi.xml my_spi;

或者：

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

或者不需指明Instance name:

&Instance simple_spi;

```
//: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
   );
```

注意：

- IPXACT的文件名可以与module相同或者不同，module name最终由IPXACT的name定义；
- Instance往往跟 Connect同时使用
- 采用IPXACT做instance的时候，IPXACT文件里定义的所有的interface都会被新增到内嵌的interface数组中；
  - 也即后续可以直接使用这些interface;
- 其他与Verilog Instance用法一致；

内嵌函数 - Instance - JSON

此函数有第三种用法，即可直接调用JSON file，当作module 来做instance，比如：

&Instance my_test_design.JSON u_my_test_design;

或者：

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

或者不需指明Instance name:

&Instance my_test_design；

注意：

- JSON的文件名可以与module相同或者不同，module name最终由JSON内的的"module"来定义；
- 采用JSON做instance的时候，JSON文件里定义的所有的interface都会被新增到内嵌的interface数组中；
  - 也即后续可以直接使用这些interface;
- JSON文件里至少包括3种信息：
  - "module":模块名字；
  - "busInterfaces": 模块所有的interface定义；
    - 如果没有interface则可以为空；
  - "ports"：所有的输入输出port;
- 其他与Verilog Instance用法一致；

```
&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"

  }
}
```

内嵌函数 - Instance - Parameters

Instance函数支持多行parameter的使用，但是对格式有一定的要求，
示例：

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

注意：

- 采用多行parameter的时候，instance内容必须有3个内容，并且必须分行写出
  - 第一行为module或者IPXACT文件名字；
  - 第二行开始到以")"为止的行是参数行；
  - 最后一行是instance名字；
- 无其他特殊要求

```
//: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
   );
```

内嵌函数 – AddParam

AddParam 函数用于向Instance增加配置参数, 比如:

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

```
//: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
```

注意:

- 一行 AddParam 配置一个parameter;
- 多行AddParam配置多个parameter;
- 最好放在instance 行之后和Connect行之前;
- **使用了AddParam后就不能使用Instance的多parameter功能**;
  - 工具只会采用AddParam的参数表而忽略其他;
- 此函数是实现多行参数列表的另外一个方法;

```
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
  );
```

内嵌函数-Connect

此函数**必须**跟Instance 一起使用，用于实现instance sub-module时候做port连接。此函数最大特色在于支持正则表达式来做信号的连接和改名，同时也支持以interface为组来连接和改名(interface可以是内嵌默认比如AMBA bus，或者自定义—见后续)，示例:

```
&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} ; //连接APB3 bus到APB3
```

信号，都加上后缀"_$suffix"，并且信号名字全部大写

注意：

- 必须紧跟Instance，与Instance之间不能有空行（**空行即是结束标志**）；
- 可以指定对输入port(-input)还是输出port(-output)做匹配;
- 正则表达式支持标准写法，由2个紧跟的字符串组成:
  - 第一个是port名字的匹配，第二个是期望的名字转换;
  - 正则表达式中可以使用变量，比如$i, $name;
  - 使用变量的时候强烈建议加上**"{ }"**以避免出错;
  - 第一个参数加不加"**/**"是可选，不影响结果;
  - 第二个参数中如果使用正则表达式的匹配结果**$n**，则**必须加上"\"**;
    - 同时前面的匹配项必须加上**()**
- Port所连接的信号可以指定大写(-up)或者小写(-low);
  - 注意：-low优先于-up，同时使用只有-low有效
- 前后行有覆盖功能，即后面的命令行会覆盖掉前面的;
- 但是如果在使用的时候加上了"**-final**"，则此命令不会被后续覆盖;
- 如果想要按照interface来做port连接，则可以采用"-interface intf_name \{1}…"
  - 注意：**\${1}**必须明确写出，否则出错;
  - 默认interface只有AMBA bus;
  - 如果需要其他interface，则需要手动添加，见后续 Interface相关函数;

```
&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
   );
```

```
&Instance test_sys_ctrl_apb_regs;
  Connect -final /(psel)/  \${1}_master_dec ;
  Connect -final /(prdata)/  \${1}_slv_muxed ;
  Connect -final /(pready)/  \${1}_slv_ored ;
  Connect -final -interface APB3  -up \${1}_suffix ;
```

```
test_sys_ctrl_apb_regs  u_test_sys_ctrl_apb_regs (
    .pclk                       (PCLK_SUFFIX                      ), //|<-i
    .presetn                    (PRESETN_SUFFIX                   ), //|<-i
    .paddr                      (PADDR_SUFFIX[31:0]               ), //|<-i
    .penable                    (PENABLE_SUFFIX                   ), //|<-i
    .psel                       (psel_master_dec                  ), //|<-i
    .pwdata                     (PWDATA_SUFFIX[31:0]              ), //|<-i
    .pwrite                     (PWRITE_SUFFIX                    ), //|<-i
    .prdata                     (prdata_slv_muxed[31:0]           ), //|>-o
    .pready                     (pready_slv_ored                  ), //|>-o
    .pslverr                    (PSLVERR_SUFFIX                   ), //|>-o
    .sys_ctrl0_12c_strip_mode   (sys_ctrl0_12c_strip_mode[2:0]    ), //|>-o
    .sys_ctrl0_mem_repair_done  (sys_ctrl0_mem_repair_done[6:0]   ), //|>-o
    .sys_ctrl0_mem_repair_en    (sys_ctrl0_mem_repair_en[0:0]     ), //|>-o
    .sys_ctrl0_pdc_use_arm_ctrl (sys_ctrl0_pdc_use_arm_ctrl[0:0]  ), //|>-o
    .sys_ctrl0_smmu_mmusid      (sys_ctrl0_smmu_mmusid[4:0]       ), //|>-o
    .test_reg_test_field0       (test_reg_test_field0[3:0]        ), //|>-o
    .test_reg_test_filed1       (test_reg_test_filed1[1:0]        )  //|>-o
   );
```

内嵌函数– Interface

- 此工具中有多个读入、处理Interface的函数，可以从 RTL 文件，IPXACT文件，JSON文件，或者 Perl hash，或者内嵌的SV code读入interface定义，然后在后续的code中可以直接使用他们，比如做连接，或者打印，或者其他， 示例:

```
//: &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
```

可以得到:

```
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: 具体使用细节可以参考Sample Code

内嵌函数-新增修改和输出Interface

AddIntfByIPX	读入IPXACT文件，解析IPXACT(xml)文件的里interface并且把所有定义的interface加入的内嵌interface列表中去	**&AddIntfByIPX("./simple_spi.xml");**
AddIntfByJson	读入JSON文件，解析JSON里的interface定义，并且把所有interface加入的内嵌interface列表中去	**&AddIntfByJson("MyIntf.json");**
AddIntfByRTL	读入RTL 定义的port到指定interface中去，可以指定关键词过滤	**&AddIntfByRTL("MyIntf.v“. “MyIntfName”, “key_word”);**
AddIntfBySV	读入内嵌SystemVerilog定义的interface来新增到内部的interface列表	**&AddIntfBySV($sv_code);**
AddIntfByHash	读入Perl hash数组，通过匹配信号名字来新增interface	**&AddIntfByHash(\%MyIntf, “MyIntf”, “key_name”);**
AddIntfByName	向指定interface新增单个信号	**&AddIntfByName("clk “，”input:1", "intf_name");**
RmIntfPort	在指定的已经存在的内嵌interface中减少一个信号	**&RmIntfPort(“clk“, “intf_name”);**

可用于第三方IP或者自研模块集成

PrintIntfPort	打印出interface所定义的信号，有若干选项来指定期望输出	**&PrintIntfPort("-intf MyIntf -awd 18 -dwd 32 -pre Test_ -low-port -master -end ,");**
PrintAmbaBus	打印出标准AMBA Bus所定义的信号，有若干选项来指定期望输出	**&PrintAmbaBus("-type test_APB3 -awd 18 -dwd 32 -pre Test_ -suf _End -up -wire -end ,");**
ShowIntf	以哈希数组的形式显示输出指定interface的信息	**&ShowIntf(“intf_name”);**

内嵌函数-IPXACT和JSON

ReadIPX	读入IPXACT文件, 分析IPXACT内容，把定义的Interface增加到内嵌interface列表	**&ReadIPX("./cfg/simple_spi.xml");**
ShowIPXIntf	读入IPXACT文件, 分析IPXACT内容，把定义的Interface以JSON方式打印输出到文件$file.intf (用于集成时debug)	**&ShowIPXIntf("./cfg/ipx.xml");**
TransIPX	读入IPXACT文件,分析IPXACT内容，把定义的ModuleName, Interface, Port转换成JSON格式并且输出到IPXACT.xml.JSON (用于集成或debug)	**&TransIPX("ipx.xml");**
ExptIntf	输出指定的Interface --- 后续可以输出到Module JSON文件(GenModJson) 可改名、增加前后缀、更改大小写等	**&ExptIntf("-intf_name APB3 -name My_APB3 -upcase -prefix Pre_ -master");**
ExptPort	输出一个Port---后续可以输出到Module JSON文件(GenModJson)	**&ExptPort("clk", "input:1", "1");**
RmPort	移除一个Port---后续不会输出到Module JSON文件(GenModJson)	**&RmPort("clk");**
GenModJson	产生当前顶层模块的JSON文件, 包括模块名字，所有Export的Interface和Port(用于后续集成） 所有当前模块RTL里定义过的port都会被自动输出；	**&GenModJson();**
~~**GenModIPX**~~	~~Generate a standard IPXACT file (xml) as what RTL looks like~~	~~**&GenIPX("my_design", "MyCorp");**~~ ~~**On plan but not start**~~ **已摒弃**

用于第三方IP或者自研模块集成

内嵌函数– GenModJson

&GenModJson 函数用于产生当前模块的JSON文件，内置本模块关键信息，比如模块名字、Interfaces、Ports，此JSON文件可被用于更高一级的集成流程中；示例：

```
//============================================================
//: &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");
```

&GenModJson();

或者：

```
//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
```

&GenModJson(“my_design”);

产生JSON示例：

```
{
    "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"
    }
}
```

注意：

- 建议不指定模块名字，工具会自动拿到当前模块名字并输出到JSON 文件;
- 此JSON文件里至少包括3种信息：
  - “module”:模块名字;
  - “busInterfaces”: 模块所有的interface定义;
    - 所有调用 **&ExptIntf()** 函数的输出;
    - 如果任何interface的port并没有被本模块定义为端口，则不会输出;
    - 如果没有interface则为空;
  - “ports”：所有的输入输出port;
    - 所有当前模块RTL里定义过的input和output port都会在文件里;
    - 任何采用**&ExptPort()**函数的端口也会被输出到文件里;

后续支持JSON5

扩展库API – 定制模块生成

用于第三方IP或者自研模块集成

ClkGen	产生Clock logic，包括MUX, DIV, ICG	**flow done but need custom design**
RstGen	产生Reset logic	**flow done but need custom design**
PMUGen	产生PMU模块/logic	**flow done but need custom design**
FuseGen	Generate Fuse module, with parameters	**flow done but need custom design**
AsyncGen	通过参数配置产生各种fifo(同步异步)	**flow done but need custom design**
FifoGen	通过参数配置产生各种fifo(同步/异步)	**flow done but need custom design**
MemGen	根据工艺和设计要求产生sram	**flow done but need custom design**
…	…	…

- 需要参数配置文件(JSON)
- 需要设计模板文件(design template file)
  - 以Verilog Code为主
  - 采用ePerl 语法

扩展函数示例 – FifoGen

此工具中有多个扩展函数，用于生成设计模块，然后在后续的RTL code中可以直接使用他们，比如Instance并做连接。此类函数的使用方法非常类似，都需要3个输入，以FifoGen为例：

- 函数调用使用时需要指定生成的模块名字(mod_name);
- 需要指定参数配置文件(JSON file);
- 需要存在设计模板文件(design template file);
  - 此文件在使用时不可见，需要事先在特定目录下放置、实现;
  - 此文件完全由第三方定制、设计，与本工具无关;

配置文件(JSON):

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

使用示例:

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

&Instance Test_SFifo;

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

&Instance Test_AFifo;
```

在当前目录下会生成2个模块文件:

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

设计模板文件:

```
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                     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];";
    }
}
```

在RTL里生成如下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
  );
```

扩展函数 – 设计模板

此工具中的每个扩展函数至少需要一个模板设计文件（具体个数由扩展函数作用决定），此模板设计文件采用**ePerl**语法，简单介绍如下：

- 总体风格是Verilog语法，也即主要是Verilog Code；
- 在需要用变量的地方需要加上 **<:** 和 **:>** 来标注；
- 变量采用Perl风格，也即$var,比如 **<:$my_sig:>**；
- 如果需要任何控制语句，则也必须用 **<:** 和**:>** 包起来；
- 控制语句语法类C语言；
- 控制语句内可以直接填入带变量的RTL code；
- 如有需求，也可以采用 **$VOUT .=** 而不是**print**；

```
//// 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
```

其他函数

CallCmd	调用 Shell/Perl/Python 命令	**&CallCmd("create_design.py -n my_design -d 32 -a 18");**
DTIWire	生成DTI interface的wire信号，必须指定前缀名和数据位宽	**&DTIWire("top2me", 512);**
DTISlave	生成DTI interface的Slave port信号，必须指定前缀名和数据位宽	**&DTISlave("top2me", 512);**
DTIMaster	生成DTI interface的Master port信号，必须指定前缀名和数据位宽	**&DTIMaster"top2me", 512);**
…	…	…

用于私有工具调用;
或者通用接口产生;

DSL is really Cool

Verilog is still the King

Connection is what you need

Flexibility is really helpful

https://github.com/WilsonChen003/HDLGen
https://gitee.com/wilson_chen/HDLGen