IP Generators - A Better Reuse Methodology

Amanjyot Kaur, Agnisys

Problem Statement

- Designers spend a huge amount of time creating standard IPs.
  - These IPs are very rigid and brittle at the same time.
- To control the flow and reuse of such IPs, engineers use a parameter-based flow,
  - but they generally break when used in a different environment because changes in a port list or customization of these IPs is difficult.
- Also, when the RTL changes, it becomes of utmost importance to vary the corresponding application programming interfaces (APIs) and sequences already created for it.

Proposed Methodology

- Configuring the IPs using parameters:
  - Generate time
  - Instance/elaboration time
- Customizations required, such as additional fields or registers added to the register map (regmap) of the IP
- Creating interconnections between different IPs
- Automatic generation of configuration APIs in UVM and C format
- Creation of test sequences for different platforms like firmware, validation, verification, and Automatic Test Equipment (ATE)
- The choice of the bus used to access the IP, such as, AHB, APB, AXI, etc, is abstracted out, to avoid the design becoming too brittle

Implementation Details

- General Purpose Input/Output (GPIO)
- Advanced Encryption Standard (AES)
- Programmable Interrupt Controller (PIC)
- Serial Peripheral Interface (SPI)
- Pulse Width Modulation (PWM)
- Direct Memory Access (DMA)
- Inter-Integrated Circuit (I2C)
- Integrated Inter-IC Sound Bus (I2S)
- Universal Asynchronous Receiver/Transmitter (UART)
- Timer

	IP generator approach	Parameterizable IP
Flexibility	Since IPs are being created on the fly at generation time, there is additional flexibility in controlling how to create the IP	Limited flexibility since only the parameters can be used for customization
Ability to change ports	Yes	No, parameters or generics cannot change the port list in the RTL
Ability to add registers or fields to the IP's regmap	Yes	No
Impact on other aspects of IPs	Along with RTL, design verification environment, firmware, software, and documentation can also be generated	The impact of parameters is only on Verilog/SystemVerilog/VHDL code (no link to other aspects)
Development resources required	Lower	Higher

Table 1. Comparison of IP generator approach and parameterizable approach

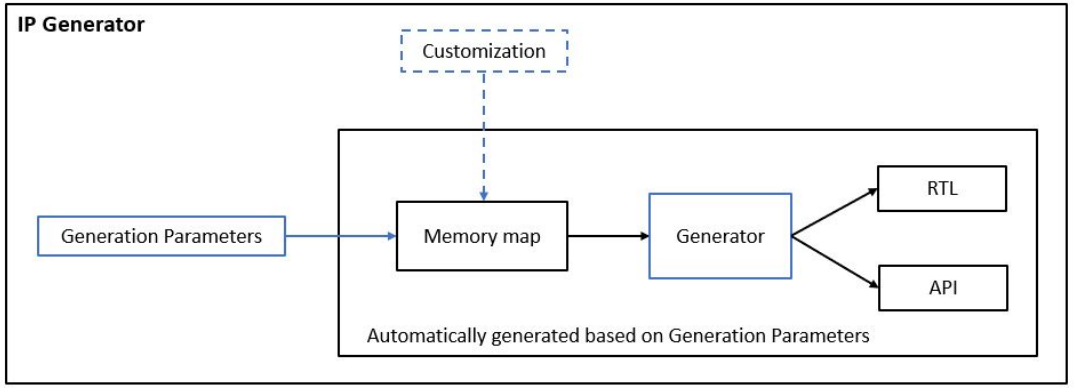


Figure 1. Block Diagram of IP Generators

Application

- Trigger Word Detector (TWD)
- Design can detect one of the four unique words in the audio sample and then drive the appropriate LED, LED0-LED3, using the GPIO interface
- If none of the words is detected, LED4 glows
- Each set of weights is trained to detect four unique words
- mem1_csr block is connected to a dual port memory
- On pressing the button[0] the first set of weights is loaded by the DMA into the sample regmap from the mem2_csr
- The neural net logic uses these weights and input audio sample spectrogram values to generate an output which in turn drives the LEDs
- Similarly, on pressing button[1] the second set of weights is loaded

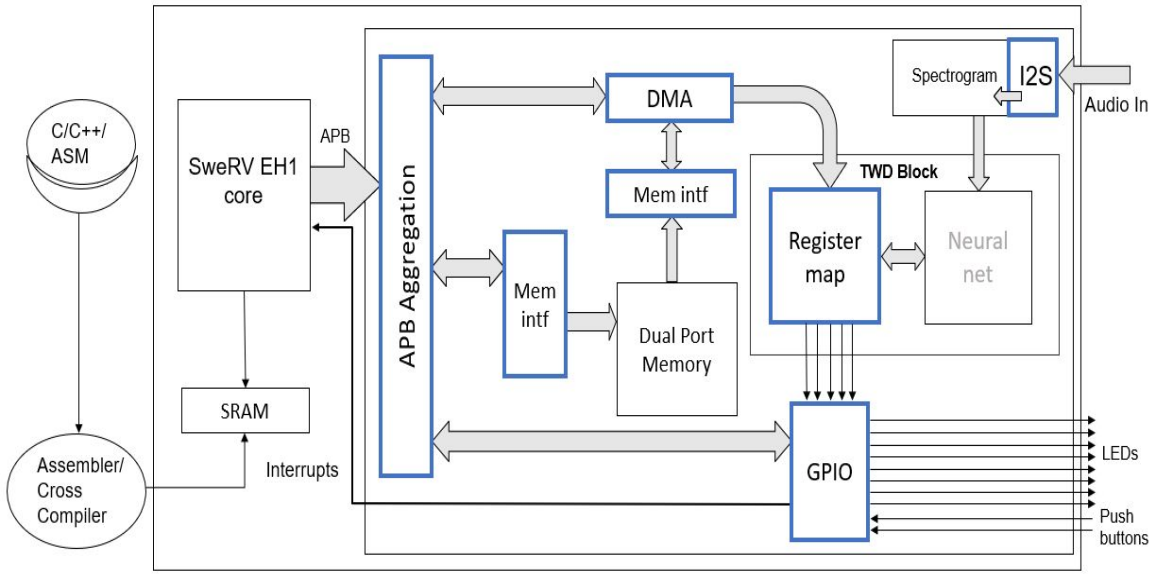


Figure 2. TWD Design Using DMA, I2S and GPIO IP

Result

```
module gpio_top#(
    localparam  NUM_GPIO = 1,
    localparam  NUM_SRC = 2,
    parameter   bus_width    = 32,
    parameter   addr_width   = 5,
    parameter   gpio_offset = 0
    )(
    ..............
    .........
        assign glb_intr_valid = (cfg_glb_intr_en_r_f != cfg_glb_intr_en_r);

        assign false_detection = glb_intr_valid | block_en_valid;

        generate
            genvar  gpio_count;
            for(gpio_count = 0;gpio_count < NUM_GPIO;gpio_count = gpio_count + 1)  begin  :  gpio

            gpio_detect_sync ds(
            //input
            .clk(clk),
            .reset(reset),
            .out_en(pin_cfg_out_en_wire[gpio_count]),
            .src(src),
            .src_sel(pin_cfg_src_sel_r),
            .ext_src_sel(pin_cfg_ext_src_srl_r[gpio_count]),
            .edge_detect_sel(pin_cfg_intr_detect_r[(gpio_count*3+2) : (gpio_count*3)]),
            .reg_data(gpio_out_data_r[gpio_count]),
            .false_detection(false_detection)
            );
        end
    endgenerate
    assign status_fld_in = (cfg_block_en_r & cfg_glb_intr_en_r) ? gpio_intr_src : {NUM_GPIO{1'b0}};

endmodule
```

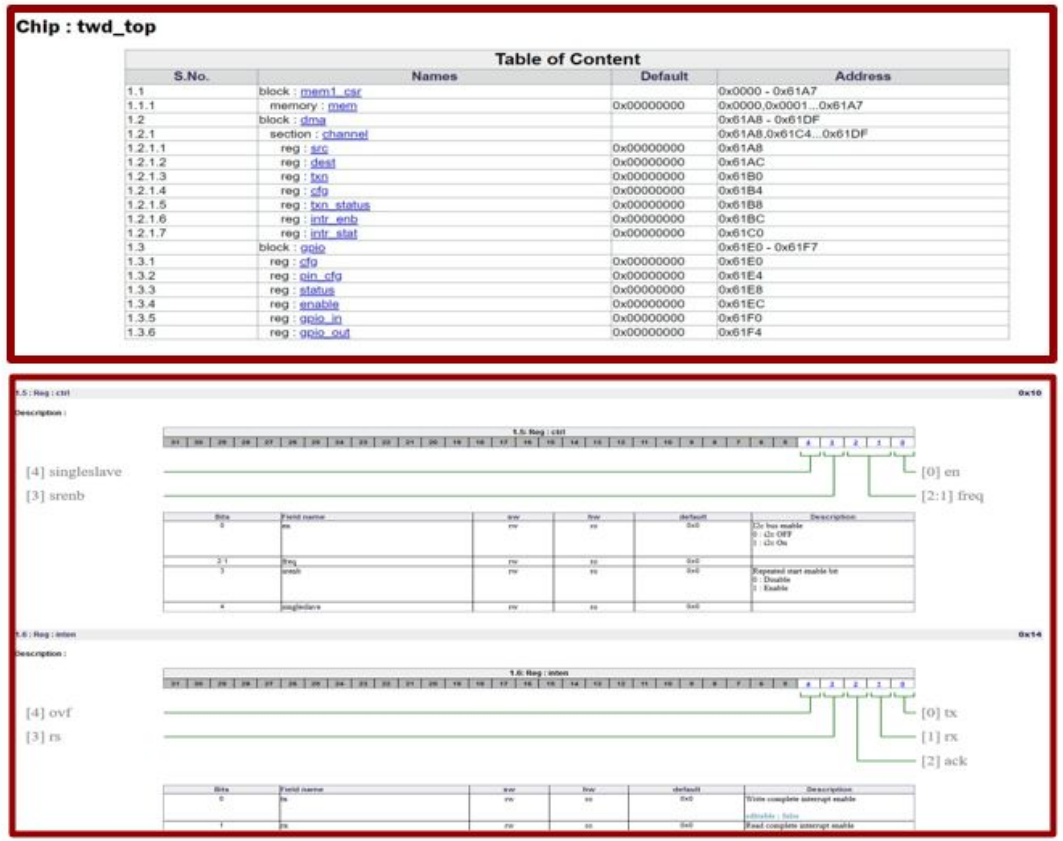
Verilog

```
/*-----------------------------------------------------------------------------
Class       : gpio_block
DESCRIPTION:-
-----------------------------------------------------------------------------*/
`ifndef CLASS_gpio_block
`define CLASS_gpio_block
class gpio_block extends uvm_reg_block;
    `uvm_object_utils(gpio_block)
    rand gpio_cfg cfg;
    rand gpio_pin_cfg pin_cfg;
    rand gpio_status status;
    rand gpio_gpio_in gpio_in;
    rand gpio_gpio_out gpio_out;

    ..........
    ........

    // Function : build
    virtual function void build();
        //define default map and add reg/regfiles
        default_map= create_map("default_map", 'h0, 4, UVM_BIG_ENDIAN, 1);

        //CFG
        cfg = gpio_cfg::type_id::create("cfg");
        cfg.configure(this, null, "cfg");
        cfg.build();
        default_map.add_reg( cfg, 'h0, "RW");

        //PIN_CFG
        pin_cfg = gpio_pin_cfg::type_id::create("pin_cfg");
        pin_cfg.configure(this, null, "pin_cfg");
        pin_cfg.build();
        default_map.add_reg( pin_cfg, 'h8, "RW");

        lock_model();
    endfunction

endclass
`endif
```

UVM

Figure 3. Generated Verilog and UVM for regmap

Result



Chip : twd_top

Table of Content

S.No.	Names	Default	Address
1.1	block : mem1_csr		0x0000 - 0x61A7
1.1.1	memory : mem	0x00000000	0x0000,0x0001...0x61A7
1.2	block : dma		0x61A8 - 0x61DF
1.2.1	section : channel		0x61A8,0x61C4...0x61DF
1.2.1.1	reg : src	0x00000000	0x61A8
1.2.1.2	reg : dest	0x00000000	0x61AC
1.2.1.3	reg : txn	0x00000000	0x61B0
1.2.1.4	reg : cfg	0x00000000	0x61B4
1.2.1.5	reg : txn_status	0x00000000	0x61B8
1.2.1.6	reg : intr_enb	0x00000000	0x61BC
1.2.1.7	reg : intr_stat	0x00000000	0x61C0
1.3	block : gpio		0x61E0 - 0x61F7
1.3.1	reg : cfg	0x00000000	0x61E0
1.3.2	reg : pin_cfg	0x00000000	0x61E4
1.3.3	reg : status	0x00000000	0x61E8
1.3.4	reg : enable	0x00000000	0x61EC
1.3.5	reg : gpio_in	0x00000000	0x61F0
1.3.6	reg : gpio_out	0x00000000	0x61F4

HTML

```
module ids_top_csr_apb_aggregation(

    pclk,      //Bus clock
    presetn,      //Reset
    psel,      //Select    : It indicates that the slave device is selected and a data tran
    penable,      //Enable    : This signal indicates the second and subsequent cycles of a
    pwrite,      //Direction : This signal indicates an APB write access when HIGH and an A
    pprot,      //Protection type : This signal indicates the normal, privileged, or secure
    pstrb,      //Write strobes : This signal indicates which byte lanes to update during a
    pwdata,      //Write data
    paddr,      //Address bus
    pready,      //Ready     : The slave uses this signal to extend an APB transfer
    prdata,      //Read data
    pslverr,      //pslverr : This signal indicates a transfer failure.


    //////////////Aggregation Logic////////////////

    wire  mem_csr_ids_select;
    assign mem_csr_ids_select =( ((block_offset + paddr >= 'h10000) && (block_offset + padd
    assign mem_csr_ids_psel = mem_csr_ids_select;

    assign gpio_csr_ids_pclk = pclk;
    assign gpio_csr_ids_presetn = presetn;
    assign gpio_csr_ids_penable = penable;
    assign gpio_csr_ids_pwrite = pwrite;
    assign gpio_csr_ids_pprot = pprot;
    assign gpio_csr_ids_pstrb = pstrb;
    assign gpio_csr_ids_pwdata = pwdata;
    assign gpio_csr_ids_paddr = paddr[gpio_csr_addr_width - 1 :0];

endmodule
```

Aggregation Logic

Figure 4. HTML and Aggregation Logic of TWD

Result

```c
int gpio_init_out( int out_pin,int ext_src_sel,enum gpio_EXT_SRC_ENB_e ext_src_enb) {
static const int block_enb = 1 ;
int dim_wr;
REG_WRITE(gpio_cfg_ADDRESS,0x00000000);
//-------------------------------------------------------------------------------
/*---- Configuring GPIO pin as output*/
//-------------------------------------------------------------------------------
dim_wr = (gpio_pin_cfg_OFFSET + (gpio_pin_cfg_PER_INSTANCE_SIZE * (out_pin))) + (gpio_s_OFFSET);
FIELD_WRITE(dim_wr,0x00000001,GPIO_PIN_CFG_OUT_EN_MASK,GPIO_PIN_CFG_OUT_EN_OFFSET);
//-------------------------------------------------------------------------------
/*---- Configuring the driving mode for gpio out pin*/
//-------------------------------------------------------------------------------
.   .   .
if( ext_src_enb == 1){
    //-------------------------------------------------------------------------------
    /*---- Select the Ext. Source which will drive GPIO o/p pin*/
    //-------------------------------------------------------------------------------

.   .   .
}
//-------------------------------------------------------------------------------
/*---- Enabling the GPIO Block*/
//-------------------------------------------------------------------------------
REG_WRITE(gpio_cfg_ADDRESS,block_enb);
return 0;
}
```

C API

```c
class uvm_gpio_init_out_seq extends uvm_reg_sequence#(uvm_sequence#(uvm_reg_item));
    `uvm_object_utils(uvm_gpio_init_out_seq)
    uvm_status_e status;
    gpio_block rm ;
.   .   .
    typedef enum  {
        INT_SOURCE  =  0,  /*    */
        EXT_SOURCE  =  1  /*    */
    }  gpio_EXT_SRC_ENB_e ;

    int out_pin=0; //GPIO pin number to be configured as output
    int ext_src_sel=0; //Select ext. source which will drive GPIO out pin
    gpio_EXT_SRC_ENB_e ext_src_enb=INT_SOURCE;
.   .   .
    task body;
.   .   .
        rm.cfg.write(status, 'h00000000, .parent(this));
        //-------------------------------------------------------------------------------
        /*---- Configuring GPIO pin as output*/
        //-------------------------------------------------------------------------------
        rm.pin_cfg[out_pin].out_en.write(status, 'h1, .parent(this));
        //-------------------------------------------------------------------------------
        /*---- Configuring the driving mode for gpio out pin*/
        //-------------------------------------------------------------------------------
.   .   .
        if (ext_src_enb == 1) begin
        //-------------------------------------------------------------------------------
        /*---- Select the Ext. Source which will drive GPIO o/p pin*/
        //-------------------------------------------------------------------------------
.   .   .
        end
        //-------------------------------------------------------------------------------
        /*---- Enabling the GPIO Block*/
        //-------------------------------------------------------------------------------
        rm.cfg.write(status, block_enb, .parent(this));
    endtask: body
endclass: uvm_gpio_init_out_seq
```

UVM sequence

Figure 5. Sample of Configuration APIs of GPIO in C and UVM

Result

IP related generated files (RTL) with number of lines

S.No.	File Name	Line of code
Configuration Bus File		
1	apb_widget.v	80
GPIO IP (Config bus : APB, Number of sources : 2, Number of outputs : 5)		
2	sync_ff.v	24
3	edge_detect.v	26
4	gpio_top.v	252
5	gpio.v	745
DMA IP (Config bus : APB, Bus type : APB, Number of channels : 2)		
6	dma_regmap_arbiter.v	227
7	dma_apb_master.v	132
8	dma_regmap_core.v	407
9	dma_regmap_txn.v	379
10	fifo.v	102
11	dma.v	742
I2S IP (Config Bus: APB, Interrupt generation with enable)		
12	i2s_master.v	310
13	prescaler.v	82
14	i2s_csr_block.v	970
15	txn_fifo.v	116
16	i2s_core.v	521

Conclusion

- Fully configurable
- Easily customizable
- IPs are generated from the command line or on a click of a button
- Unencrypted code
- Ability to target multiple platforms (Design, Verification, Firmware, Software, and Documentation)
- For about 10 different IPs, and about 50 different instantiations, users can easily generate 100,000 lines of RTL code, UVM, and C test environment

Questions