gen_regs_py

Generated: May 27, 2020

Contents

Welcome to gen_regs_py's documentation!

Here we talk about `gen_regs_py`, a tool for creating registers

Overview

gen_regs_py is a Python based Register RTL generation tool. The aim of gen_regs_py is to automate the register design and creation process, resulting in quicker RTL development and fewer bugs along the way.

Using simple text file input, a designer can quickly develop registers. No clunky spreadsheets, no crazy syntaxes, just simple text files.

The tool will create a register block that has a APB/AHB slave interface to be used for transactions

Running the Script (Options)

`-h, --help` Shows the HELP message

-i, -input_file (REQUIRED)

Input file to be used for parsing. There are no requirements on the file extention

-p, -prefix (REQUIRED)

PREFIX NAME to be used. This has no effect on the bitfield names in the RTL

-b, -block (REQUIRED)

BLOCK NAME to be used. This has no effect on the bitfield names in the RTL

-ahb (Optional)

Creates the register block with an AHB-Lite supported interface.

> ***Note***
>
> Register operation is not changed by interface type.

-sphinx (Optional)

Prints out a Sphinx formatted table for documentation purposes.

-dv (Optional)

Creates 'DV' related files that are used for DV and/or fed into the `gen_uvm_reg_model` script.

-dbg (Optional)

Prints some info to the console during building. Can be used to track down any incorrect input file setup

> ***Note***
>
> The **PREFIX** and **BLOCK** names are used to *uniquify* the design. For the RTL the only place these are seen is in the output RTL and module name. During DV, these are used as qualifiers to specific blocks.

After running the script, provided no errors for setup, you should receive a verilog file in the following format: `<prefix>_<block>_regs_top.v`

Here is an example of an input file

```
REG1                RW
  bf1               5'b0                  Some description1
  bf1_mux           1'b1                  Some description2
  bf2               5'b0                  Some description1
  bf2_mux           1'b1                  Some description2
  bf3               4'ha
  bf3longname       5'd10

AREADONLYREG        RO
  some_status_in    1'b0                  A signal I want to observe
```

And here is what part of the output Verilog would look like

```
//---------------------------
// REG1
// bf1 - Some description1
// bf1_mux - Some description2
// bf2 - Some description1
// bf2_mux - Some description2
// bf3 -
// bf3longname -
//---------------------------
wire [31:0] REG1_reg_read;
reg  [4:0]   reg_bf1;
reg  [4:0]   reg_bf2;
reg [3:0]   reg_bf3;
reg [4:0]   reg_bf3longname;

always @(posedge RegClk or posedge RegReset) begin
  if(RegReset) begin
    reg_bf1                             <= 5'h0;
    reg_bf1_mux                         <= 1'h1;
    reg_bf2                             <= 5'h0;
    reg_bf2_mux                         <= 1'h1;
    reg_bf3                             <= 4'ha;
    reg_bf3longname                     <= 5'ha;
  end else if(RegAddr == 'h0 && RegWrEn) begin
    reg_bf1                             <= RegWrData[4:0];
    reg_bf1_mux                         <= RegWrData[5];
    reg_bf2                             <= RegWrData[10:6];
    reg_bf2_mux                         <= RegWrData[11];
    reg_bf3                             <= RegWrData[15:12];
    reg_bf3longname                     <= RegWrData[20:16];
  end else begin
    reg_bf1                             <= reg_bf1;
    reg_bf1_mux                         <= reg_bf1_mux;
    reg_bf2                             <= reg_bf2;
    reg_bf2_mux                         <= reg_bf2_mux;
    reg_bf3                             <= reg_bf3;
    reg_bf3longname                     <= reg_bf3longname;
  end
end

assign REG1_reg_read = {11'h0,
        reg_bf3longname,
        reg_bf3,
        reg_bf2_mux,
        reg_bf2,
        reg_bf1_mux,
        reg_bf1};

//-----------------------

wire [4:0]  swi_bf1_muxed_pre;
wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf1[4:0] (
  .clk0    ( bf1                            ),
  .clk1    ( reg_bf1                        ),
  .sel     ( reg_bf1_mux                    ),
  .clk_out ( swi_bf1_muxed_pre              ));

assign swi_bf1_muxed = swi_bf1_muxed_pre;

//-----------------------
//-----------------------

wire [4:0]  swi_bf2_muxed_pre;
wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf2[4:0] (
  .clk0    ( bf2                            ),
  .clk1    ( reg_bf2                        ),
  .sel     ( reg_bf2_mux                    ),
  .clk_out ( swi_bf2_muxed_pre              ));

assign swi_bf2_muxed = swi_bf2_muxed_pre;

//-----------------------
//-----------------------
assign swi_bf3 = reg_bf3;

//-----------------------
assign swi_bf3longname = reg_bf3longname;




//---------------------------
// AREADONLYREG
// some_status_in - A signal I want to observe
//---------------------------
wire [31:0] AREADONLYREG_reg_read;
```

```
assign AREADONLYREG_reg_read = {31'h0,
        some_status_in};
```

Methodology

`gen_regs_py` builds software registers by using the following methodology of “Classes”:

```
Register Blocks
 |-- Registers
      |-- Bitfields
```

Bitfields

`Bitfields` are individual “flops” that are usually used for some specific purpose. For example, you have a software bit that is used to enable a piece of logic. You name this `logic_enable`. While only a singlebit, there is still room in the 32bit register for additional logic. `Bitfields` can be upto 32 bits in size. If you need control over a signal larger than this, you will need to create multiple bitfields across registers.

Registers

`Registers` are a collection of `bitfields` that comprise a 32bit sfotware register. Registers can be upto 32bits in size, but are not required to be. For example, if you define a single 16bit bitfield in a register, and no other bitfields, only the lower 16 bits are accessible. Any reads will result in the top 16 bits returning 16’d0.

Register Blocks

`Register Blocks` are a collection of `Registers` and is the essential output of the `gen_regs_py`. The `Register Block` is the actual RTL that you will instantiate in your design. These `Register Blocks` are then used in other register flows for DV and SW register generation.

> ***Note***
>
> When creating registers, you are really creating each bitfield and grouping them in a collection, which is a register. And this collection of registers is what constitutes the register block

Input Register File

gen_regs_py uses basic text files for describing registers. There are no requirements for file extensions, only that the file is readable.

Syntax

The gen_regs_py utilizes a flexible syntax strategy to allow for simple to complex register schemes to be implemented.

> ***Note***
>
> For simplicity sake, we will show the most common register formats here and describe more complex usage scenarios in the register/bitfields types page

Register Declaration

A register declaration will follow this syntax

```
<REGNAME>       <REGTYPE>    <{NO_REG_TEST}>   <DESCRIPTION>
```

REGNAME	Required	Name of the register. Must be unique to all other register for this block
REGTYPE	Required	Base type of register. Must be RW or RO and denotes the *default* bitfield types
{NO_REG_TEST}	Optional	When {NO_REG_TEST} is defined, DV output files will result in this register being excluded from register testing
DESCRIPTION	Optional	Description of register. Must be on one line

> ***Note***
>
> You may notice that there is no **address** declaration. This is because gen_regs_py will automatically assign addresses based on the location of the register in the file. The first register is assigned address 0x00, the second, address 0x04, and so on.
>
> If a user wants to place certain registers at certain addresses, the user would want to manually place the registers in the correct order. Reserved registers can be created by declaring a register and setting one or more bits as “reserved”.
>
> ```
> REG1        RW
>   bf1       1'b0
>
> RSVRD0      RW        //No registers are generated but the space is reserved
>   reserved  1'b0
>
> REG_AT_X8   RW
>   bf2
> ```

Bitfield Declaration

A bitfield declaration will follow this syntax

```
<BFNAME>       <BFRESET>    <BFTYPE> <{DFT}>  <DESCRIPTION>
```

BFNAME	Required	Name of the bitfield. Must be unique to all other bitefields for this block

BFRESET	Required	Denotes the width and reset value of this bitfield. Number prior to radix denotes the width, while the value after the width is assigned the reset value.
BFTYPE	Optional	Allows user to force a particular bitfield type for this register, regardless of how the REGTYPE is defined.
{DFT}	Optional	Creates DFT related overrides
DESCRIPTION	Optional	Description of bitfield. Not required. Must be on one line

> ***Note***
>
> Bitfields defined with BFNAME `reserved` are treated as reserved bitfield allocations. `gen_regs_py` will not create a bitfield for these location, and these locations always read back all zeros. The `reserved` bitfield keyword can be used multiple times.

Comments

Lines beginning with # are treated as comments and not parsed

Putting it all together

This is the general structure of each register in the input file

```
<REGNAME>       <REGTYPE>                 <{NO_REG_TEST}>   <DESCRIPTION>
  <BFNAME>      <BFRESET>    <BFTYPE>     <{DFT}>           <DESCRIPTION>
  <BFNAME>      <BFRESET>    <BFTYPE>     <{DFT}>           <DESCRIPTION>
  <BFNAME>      <BFRESET>    <BFTYPE>     <{DFT}>           <DESCRIPTION>
```

Here is a simple example of a register block with three registers being created. We have `REG1` in which we define as RW and define two bitfields, which are each RW. We have `AREADONLYREG` which we define as RO and define a single bitfield which in turn is a RO bitfield. And finally we have `RWREG_WITH_RO` which we have defined as RW, however we also define a bitfield as RO which will force the bitfield `somerobf` to a RO bitfield

```
REG1                RW                    This is the first register
  bf1               5'b0                  A description
  bf2               4'h3                  Look how I use 'h

AREADONLYREG        RO
  some_status_in    1'b0                  A signal I want to observe

RWREG_WITH_RO       RW
  somerwbf          1'b0                  This is a RW bitfield
  somerobf          3'd0      RO          But this one is read-only
```

Register/Bitfield Types

`gen_regs_py` supports the following register/bitfield types:

- RW - Read/Write
- RW with Mux Override - A Read/Write register that can use the signal value from external design or be forced through a SW register
- RO - Read-Only
- W1C - Write-One-to-Clear
- WFIFO - Creates a FIFO Write Interface
- RFIFO - Creates a FIFO Read Interface

Most bitfield types will create one or more input/output ports with the name of the bitfield .

RW (Read/Write) Bitfield

A `RW Bitfield` is a typical bitfield which can be read and written via the APB interface. `gen_regs_py` will create an output port named:

- `swi_<BFNAME> (output)` - Writes to this bitfield will cause a transition on the output port.

Here is an example RW bitfield declaration:

```
1 REG1               RW
2   bf1              5'b0                  My read-write bitfield
```

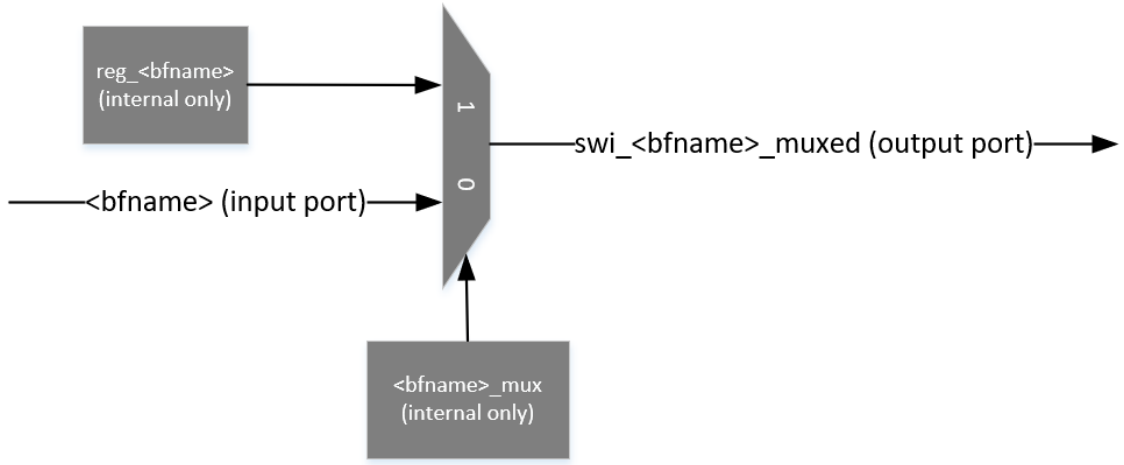
Here is the Verilog output ports for this register block

```
 1 module rw_reg_example_regs_top #(
 2   parameter    ADDR_WIDTH = 8,
 3   parameter    STDCELL    = 1
 4 )(
 5   //REG1
 6   output wire [4:0]   swi_bf1,
 7
 8   //DFT Ports (if used)
 9
10   // APB Interface
11   input  wire RegReset,
12   input  wire RegClk,
13   input  wire PSEL,
14   input  wire PENABLE,
15   input  wire PWRITE,
16   output wire PSLVERR,
17   output wire PREADY,
18   input  wire [(ADDR_WIDTH-1):0] PADDR,
19   input  wire [31:0] PWDATA,
20   output wire [31:0] PRDATA
21 );
22
23 // ...excluded for clarity
24
25 //----------------------------
26 // REG1
27 // bf1 - My read-write bitfield
28 //----------------------------
29 wire [31:0] REG1_reg_read;
30 reg [4:0]   reg_bf1;
31
32 always @(posedge RegClk or posedge RegReset) begin
33   if(RegReset) begin
34     reg_bf1                                <= 5'h0;
35   end else if(RegAddr == 'h0 && RegWrEn) begin
36     reg_bf1                                <= RegWrData[4:0];
37   end else begin
38     reg_bf1                                <= reg_bf1;
39   end
40 end
41
42 assign REG1_reg_read = {27'h0,
43         reg_bf1};
44
```

```
45 //-----------------------
46 assign swi_bf1 = reg_bf1;
```

RW (Read/Write) Bitfield with Mux Override

This type of bitfield structure will create a register that can be used to “override” a particular signal. This is often used in cases where you may be getting a signal from another block and want to have a way to control that signal through software. For example, you have a calibration state machine that has an *enable* signal which is set from some controller. During testing, you want to be able to control that enable with software, but in normal usage want it to be controlled via the logic. This allows the user to take control without the need to add any additional logic.



A user can describe this structure by defining a bitfield along with another bitifled with `_mux` appended to the end of the name. This will trigger `gen_regs_py` to construct this type of structure.

`gen_regs_py` will create two ports for this structure:

- `<BFNAME> (input)` - External logic connection.
- `swi_<BFNAME>_muxed (output)` - Result of the register/mux override.

Here is an example

```
1 REG1               RW
2   bf1              5'b0                  My read-write bitfield
3   bf1_mux          1'b0                  Mux register select
```

And here is the Verilog output

```
 1 module rw_reg_mux_example_regs_top #(
 2   parameter    ADDR_WIDTH = 8,
 3   parameter    STDCELL    = 1
 4 )(
 5   //REG1
 6   input  wire [4:0]   bf1,
 7   output wire [4:0]   swi_bf1_muxed,
 8
 9   //DFT Ports (if used)
10
11   // APB Interface
12   input  wire RegReset,
13   input  wire RegClk,
14   input  wire PSEL,
15   input  wire PENABLE,
16   input  wire PWRITE,
17   output wire PSLVERR,
18   output wire PREADY,
```

```
19   input  wire [(ADDR_WIDTH-1):0] PADDR,
20   input  wire [31:0] PWDATA,
21   output wire [31:0] PRDATA
22 );
23
24 // ...excluded for clarity
25
26 //Regs for Mux Override sel
27 reg  reg_bf1_mux;
28
29
30
31 //---------------------------
32 // REG1
33 // bf1 - My read-write bitfield
34 // bf1_mux - Mux register select
35 //---------------------------
36 wire [31:0] REG1_reg_read;
37 reg  [4:0]   reg_bf1;
38
39 always @(posedge RegClk or posedge RegReset) begin
40   if(RegReset) begin
41     reg_bf1                              <= 5'h0;
42     reg_bf1_mux                          <= 1'h0;
43   end else if(RegAddr == 'h0 && RegWrEn) begin
44     reg_bf1                              <= RegWrData[4:0];
45     reg_bf1_mux                          <= RegWrData[5];
46   end else begin
47     reg_bf1                              <= reg_bf1;
48     reg_bf1_mux                          <= reg_bf1_mux;
49   end
50 end
51
52 assign REG1_reg_read = {26'h0,
53         reg_bf1_mux,
54         reg_bf1};
55
56 //-----------------------
57
58 wire [4:0]  swi_bf1_muxed_pre;
59 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf1[4:0] (
60   .clk0    ( bf1                             ),
61   .clk1    ( reg_bf1                         ),
62   .sel     ( reg_bf1_mux                     ),
63   .clk_out ( swi_bf1_muxed_pre               ));
64
65 assign swi_bf1_muxed = swi_bf1_muxed_pre;
66
67 //-----------------------
```

As you can see, there is an input port named `bf1` and an output port named `swi_bf1_muxed`. The input port would be the signal from some external logic, where as the output port is overridden value.

> ***Note***
>
> The `bitfield` and `bitfield_mux` bitfields do not need to be in the same regsiter.

RO (Read-Only) Bitfield

A Read-Only bitfield is a bitfield which can only be read. When a RO bitfield is created, `gen_regs_py` will create the following port:

- <BFNAME> `(input)` - Connection to observe logic

> ***Warning***
>
> RO bitfields are treated as asynchronous to `gen_regs_py`. If you need SW to sample stable inputs, a demet or other external logic to the output register verilog is required.

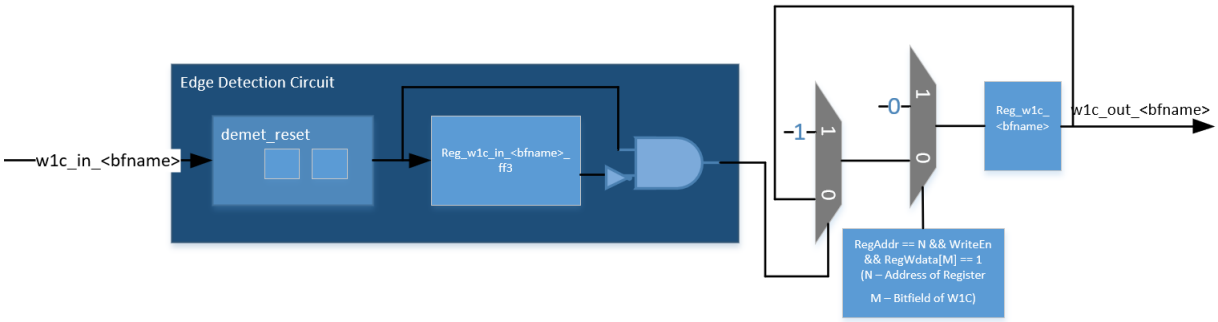
W1C (Write-One-to-Clear) Bitfield

A W1C bitfield is generally used in cases similar to interrupts (although not required). `gen_regs_py` will create the following:

- `w1c_in_<BFNAME> (input)` - Input from external logic
- `w1c_out_<BFNAME> (output)` - Output of the W1C bitfield, post rising edge detection
- A rising edge detection circuit

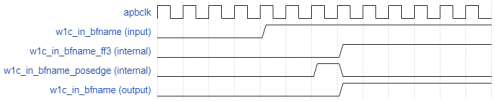
> ***Note***
>
> W1C bitfields are currently limited to single-bit width

Here is schematic representation of the W1C register logic

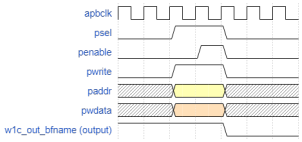


The input signal is sampled with a demet, then sent through a rising edge detection. On the rising edge the output register is set. The register will remain in this state until a 1 is written.

Example of the waveform when the input signal sets the regsiter



Exmample of the waveform when the register is written to clear



Here is an example of declaring this bitfield

```
1 REG1                RW
2   bf1               5'b0                  My read-write bitfield
3   bf1_mux           1'b0                  Mux register select
4
5
6 REG_WITH_W1C        RW
7   myinterrupt       1'b0      W1C         Using this like an interrupt
```

And here is the Verilog output

```
1 module w1c_example_regs_top #(
2   parameter    ADDR_WIDTH = 8,
3   parameter    STDCELL    = 1
4 )(
5   //REG1
```

```
  input  wire [4:0]   bf1,
  output wire [4:0]   swi_bf1_muxed,
  //REG_WITH_W1C
  input  wire         w1c_in_myinterrupt,
  output wire         w1c_out_myinterrupt,

  //DFT Ports (if used)

  // APB Interface
  input  wire RegReset,
  input  wire RegClk,
  input  wire PSEL,
  input  wire PENABLE,
  input  wire PWRITE,
  output wire PSLVERR,
  output wire PREADY,
  input  wire [(ADDR_WIDTH-1):0] PADDR,
  input  wire [31:0] PWDATA,
  output wire [31:0] PRDATA
);

// ...excluded for clarity

//---------------------------
// REG_WITH_W1C
// myinterrupt - Using this like an interrupt
//---------------------------
wire [31:0] REG_WITH_W1C_reg_read;
reg          reg_w1c_myinterrupt;
wire         reg_w1c_in_myinterrupt_ff2;
reg          reg_w1c_in_myinterrupt_ff3;

// myinterrupt W1C Logic
always @(posedge RegClk or posedge RegReset) begin
  if(RegReset) begin
    reg_w1c_myinterrupt        <= 1'h0;
    reg_w1c_in_myinterrupt_ff3 <= 1'h0;
  end else begin
    reg_w1c_myinterrupt        <= RegWrData[0] && reg_w1c_myinterrupt &&
                                  (RegAddr == 'h4) && RegWrEn ? 1'b0 :
                                  (reg_w1c_in_myinterrupt_ff2 & ~reg_w1c_in_myinterrupt_ff3 ? 1'b1 :
                                   reg_w1c_myinterrupt);
    reg_w1c_in_myinterrupt_ff3 <= reg_w1c_in_myinterrupt_ff2;
  end
end

demet_reset u_demet_reset_myinterrupt (
  .clk     ( RegClk                                 ),
  .reset   ( RegReset                               ),
  .sig_in  ( w1c_in_myinterrupt                     ),
  .sig_out ( reg_w1c_in_myinterrupt_ff2             ));

assign REG_WITH_W1C_reg_read = {31'h0,
        reg_w1c_myinterrupt};

//-----------------------
assign w1c_out_myinterrupt = reg_w1c_myinterrupt;
```

Note

To create an interrupt that is later sent out to an external IP (such as a CPU), a user can create the W1C bitfield and an associated RW bitfield to act as an enable:

```
INTERRUPT_ENABLES          RW
  int_en_trans_complete    1'b1            0-trans_complete doesn't assert the interrupt out, 1-asserts

INTERRUPT_STATUS           RO
  int_trans_complete       1'b0      W1C   Asserts when a transaction is compelte
```

And in the user logic, simply AND the output of the W1C with the interrupt enable to gate:

```
assign interrupt_out = (swi_int_en_trans_complete       && w1c_out_int_trans_complete);
```

WFIFO Bitfield

The WFIFO bitfield type is generally used in a case where you want to write to a specific address location that results in a FIFO being written.

`gen_regs_py` will create two ports for this bitfield type:

- `wfifo_<BFNAME>` - The data written to the FIFO
- `wfifo_winc_<BFNAME>` - A write increment/valid signal to the FIFO

> ***Note***
>
> There are actually no flops instantiated for this bitfield type. It is mainly a direct connection between the APB interface and the FIFO. For this reason, any reads to this bitfield will result in all zeros being read back.

> ***Warning***
>
> There is no FULL check added in the register block on the FIFO being written. A user should check FIFO state prior to writing the FIFO.

Eample input file:

```
1 REG1               RW
2   bf1              5'b0                  My read-write bitfield
3   bf1_mux          1'b0                  Mux register select
4
5
6 REG_WITH_WFIFO     RW
7   write_data       8'b0     WFIFO        Writes to the FIFO
```

And an example of the Verilog output:

```
 1 module wfifo_example_regs_top #(
 2   parameter    ADDR_WIDTH = 8,
 3   parameter    STDCELL    = 1
 4 )(
 5   //REG1
 6   input  wire [4:0]   bf1,
 7   output wire [4:0]   swi_bf1_muxed,
 8   //REG_WITH_WFIFO
 9   output wire [7:0]   wfifo_write_data,
10   output wire         wfifo_winc_write_data,
11
12   //DFT Ports (if used)
13
14   // APB Interface
15   input  wire RegReset,
16   input  wire RegClk,
17   input  wire PSEL,
18   input  wire PENABLE,
19   input  wire PWRITE,
20   output wire PSLVERR,
21   output wire PREADY,
22   input  wire [(ADDR_WIDTH-1):0] PADDR,
23   input  wire [31:0] PWDATA,
24   output wire [31:0] PRDATA
25 );
26
27 // ...excluded for clarity
28
29 //---------------------------
30 // REG_WITH_WFIFO
31 // write_data - Writes to the FIFO
32 //---------------------------
33 wire [31:0] REG_WITH_WFIFO_reg_read;
34
35 assign wfifo_write_data      = (RegAddr == 'h4 && RegWrEn) ? RegWrData[7:0] : 'd0;
36 assign wfifo_winc_write_data = (RegAddr == 'h4 && RegWrEn);
37 assign REG_WITH_WFIFO_reg_read = {24'h0,
38         8'd0}; //Reserved
39
40 //-----------------------
```

RFIFO Bitfield

The RFIFO bitifled type is similar to the WFIFO, except that this is for reading from a FIFO.

`gen_regs_py` will create two ports for this bitfield type:

- `rfifo_<BFNAME>` - The data read from the FIFO
- `rfifo_rinc_<BFNAME>` - A read increment/valid signal to the FIFO

> ***Note***
>
> There are actually no flops instantiated for this bitfield type. It is mainly a direct connection between the APB interface and the FIFO. Writes are essentially ignored for this bitfield.

> ***Warning***
>
> There is no EMPTY check added in the register block on the FIFO being read. A user should check FIFO state prior to reading the FIFO.

Eample input file:

```
1 REG1                RW
2   bf1               5'b0                    My read-write bitfield
3   bf1_mux           1'b0                    Mux register select
4
5
6 REG_WITH_RFIFO      RO
7   read_data         8'b0      RFIFO         Reads from the FIFO
```

And an example of the Verilog output:

```
 1 module rfifo_example_regs_top #(
 2   parameter    ADDR_WIDTH = 8,
 3   parameter    STDCELL    = 1
 4 )(
 5   //REG1
 6   input  wire [4:0]   bf1,
 7   output wire [4:0]   swi_bf1_muxed,
 8   //REG_WITH_RFIFO
 9   input  wire [7:0]   rfifo_read_data,
10   output wire         rfifo_rinc_read_data,
11
12   //DFT Ports (if used)
13
14   // APB Interface
15   input  wire RegReset,
16   input  wire RegClk,
17   input  wire PSEL,
18   input  wire PENABLE,
19   input  wire PWRITE,
20   output wire PSLVERR,
21   output wire PREADY,
22   input  wire [(ADDR_WIDTH-1):0] PADDR,
23   input  wire [31:0] PWDATA,
24   output wire [31:0] PRDATA
25 );
26
27
28 // ...excluded for clarity
29
30 //---------------------------
31 // REG_WITH_RFIFO
32 // read_data - Reads from the FIFO
33 //---------------------------
34 wire [31:0] REG_WITH_RFIFO_reg_read;
35
36 assign rfifo_rinc_read_data = (RegAddr == 'h4 && PENABLE && PSEL && ~(PWRITE || RegWrEn));
37 assign REG_WITH_RFIFO_reg_read = {24'h0,
38         rfifo_read_data};
39
40 //-----------------------
```

Advanced Features

There are several features to gen_regs_py which either happen behind the scenes or may be needed in certain circumstances.

- DebugBus - An auto-generated set of registers that allows the user to 'probe' signals that implement the `Mux Override` bitfield type. Also creates an output port for debugging.
- DV Files - Files needed for DV or for designer testing
- NO_REG_TEST - Excludes this register from register testing
- DFT Features - Logic settings for various DFT modes

DebugBus

`gen_regs_py` will create two additional registers if any `Mux Override` bitfield types are instantiated in the register block: * DEBUG_BUS_CTRL_SEL - Select signal for DEBUG_BUS_CTRL * DEBUG_BUS_CTRL_STATUS - Status output for DEBUG_BUS_STATUS

A port `debug_bus_ctrl_status` is also created. This is essentially the output of the RO DEBUG_BUS_CTRL_STATUS register. The intent is that a user can connect this to an external debug bus (to GPIOs for example) and have a way to probe signals with a scope or inside the testbench.

A Mux structure will be created so that a user can select the RO register or '_muxed' output to observe. DEBUG_BUS_CTRL_SEL is used to select the signal, and DEBUG_BUS_CTRL_STATUS can be read to see the value of the signal.

DEBUG_BUS_CTRL_SEL width is determined by the number of RO registers and muxed overrides. (e.g.2 seprate registers with RO bitfields and 7 muxed overrides would result in the bitfield being 4bits in width to handle 9 selections).

> ***Warning***
>
> Currently the debugbus would only support up to 2^32 overrides. If you need more than this, well, I don't know what to tell you, but you may want to re-evaluate what you are trying to do.

Each register with a RO bitfield or '_muxed' output is given it's own select value, and the position follows the following: * Registers with RO bitfields are set first, going from lowest address to highest * '_muxed' overrides are after and are based on the order in which they are declared in the file.

Here is an example of the DEBUG_BUS registers in the RTL:

```
1 //---------------------------
2 // DEBUG_BUS_CTRL
3 // DEBUG_BUS_CTRL_SEL - Select signal for DEBUG_BUS_CTRL
4 //---------------------------
5 wire [31:0] DEBUG_BUS_CTRL_reg_read;
6 reg [2:0]   reg_debug_bus_ctrl_sel;
7
8 always @(posedge RegClk or posedge RegReset) begin
9   if(RegReset) begin
10    reg_debug_bus_ctrl_sel                <= 3'h0;
11  end else if(RegAddr == 'h50 && RegWrEn) begin
12    reg_debug_bus_ctrl_sel                <= RegWrData[2:0];
13  end else begin
14    reg_debug_bus_ctrl_sel                <= reg_debug_bus_ctrl_sel;
15  end
16 end
17
18 assign DEBUG_BUS_CTRL_reg_read = {29'h0,
19         reg_debug_bus_ctrl_sel};
20
21 //----------------------
22 assign swi_debug_bus_ctrl_sel = reg_debug_bus_ctrl_sel;
23
24
```

```
25
26
27
28 //---------------------------
29 // DEBUG_BUS_STATUS
30 // DEBUG_BUS_CTRL_STATUS - Status output for DEBUG_BUS_STATUS
31 //---------------------------
32 wire [31:0] DEBUG_BUS_STATUS_reg_read;
33 reg  [31:0]  debug_bus_ctrl_status;
34
35 //Debug bus control logic
36 always @(*) begin
37   case(swi_debug_bus_ctrl_sel)
38     'd0 : debug_bus_ctrl_status = {swi_dac0_therm_lo_muxed};
39     'd1 : debug_bus_ctrl_status = {1'd0, swi_dac0_therm_hi_muxed};
40     'd2 : debug_bus_ctrl_status = {26'd0, swi_dac0_bin_muxed};
41     'd3 : debug_bus_ctrl_status = {swi_dac1_therm_lo_muxed};
42     'd4 : debug_bus_ctrl_status = {1'd0, swi_dac1_therm_hi_muxed};
43     'd5 : debug_bus_ctrl_status = {26'd0, swi_dac1_bin_muxed};
44     default : debug_bus_ctrl_status = 32'd0;
45   endcase
46 end
47
48 assign DEBUG_BUS_STATUS_reg_read = {          debug_bus_ctrl_status};
```

Note

The debug_bus is generally assumed to be a backup testing feature and/or an easy way to add those “just in case” type of status checks. It is not recommened to try to read the debug_bus in normal DV testing.

DV Files

If a user passes the `-dv` flag, two additional files will be created: * `<prefix>_<block>_dv.txt` - Used by `gen_uvm_reg_model` for UVM DV flows * `<prefix>_<block>_addr_defines.vh` - Potentually used by DV and can be used for normal verilog simulations

Note

Since `<prefix>_<block>_dv.txt` is technically an intermediate file, it will not be discussed here

`<prefix>_<block>_addr_defines.vh` will create a list of Verilog `` `defines `` that can be used for DV functions. Users will generally use these defines for simple Verilog test benches to keep up with addresses/bitfield slices. Larger DV environments can continue to use these where needed.

Registers

Registers will be defined with the following format:

`<PREFIX>_<BLOCK>_<REGNAME>  <ADDRESS>`

Bitfields

Bitfields will be defined with the following format (note the double underscore):

`<PREFIX>_<BLOCK>_<REGNAME>__<BFNAME>  <Bit select>`

Reset Value

The Reset value will be defined with the following format (note the tripple underscore):

`<PREFIX>_<BLOCK>_<REGNAME>___POR  <Reset Value>`

Here is an example of the defines from the RFIFO example:

```
1 `define RFIFO_EXAMPLE_REG1                                    'h00000000
2 `define RFIFO_EXAMPLE_REG1__BF1_MUX                                    5
3 `define RFIFO_EXAMPLE_REG1__BF1                                      4:0
4 `define RFIFO_EXAMPLE_REG1___POR                              32'h00000000
```

```
 5
 6 `define RFIFO_EXAMPLE_REG_WITH_RFIFO                                    'h00000004
 7 `define RFIFO_EXAMPLE_REG_WITH_RFIFO__READ_DATA                            7:0
 8 `define RFIFO_EXAMPLE_REG_WITH_RFIFO___POR                              32'h00000000
 9
10 `define RFIFO_EXAMPLE_DEBUG_BUS_CTRL                                    'h00000008
11 `define RFIFO_EXAMPLE_DEBUG_BUS_CTRL__DEBUG_BUS_CTRL_SEL                     0
12 `define RFIFO_EXAMPLE_DEBUG_BUS_CTRL___POR                              32'h00000000
13
14 `define RFIFO_EXAMPLE_DEBUG_BUS_STATUS                                   'h0000000C
15 `define RFIFO_EXAMPLE_DEBUG_BUS_STATUS__DEBUG_BUS_CTRL_STATUS              31:0
16 `define RFIFO_EXAMPLE_DEBUG_BUS_STATUS___POR                             32'h00000000
```

NO_REG_TEST

Occasionally there are registers in the design that are required to be excluded from normal register testing as they may interfere with other logic. To work around this, `NO_REG_TEST` can be added to registers during declaration. This will signal to `gen_uvm_reg_model` that this register should be added to the exclusion list. There is no effect on the RTL for declaring a register as `NO_REG_TEST`.

To exclude a register, simply add `{NO_REG_TEST}` to the register declaration line, after the REGTYPE, but prior to the description (if there is a description). Below is an example:

```
1 SPI0_CONTROLS                   RW      {NO_REG_TEST}
2   spi0_spi_en                   1'b1
3   spi0_spi_master_en            1'b0
```

DFT Features

Since many designs will place registers driving vital components (analog, resets, etc.) into certain states during DFT, there may be cases where a user wants to have control over the output value during various DFT modes. To accomplish this without the need to extra external logic, `gen_regs_py` allows a user to denote the value of a bitfield during specific DFT modes, and optionally add a Boundary SCAN Drive/Capture flop.

DFT Modes

Currently, `gen_regs_py` supports controls for the following DFT modes/settings:

- `CORESCAN` - DFT core scan mode for flop related testing (e.g. stuck-at)
- `IDDQ` - IDDQ Mode
- `HIZ` - Highz Mode
- BSCAN - Boundary Scan

> ### *Note*
>
> These DFT *modes* are not required to be a one-to-one match. If you wanted to use IDDQ as some type of global power down setting, you are free to do that. The naming is meant to give users a match to typical DFT modes if they are required for their design.

Declaring Bitfield Values in DFT

A user would declare a bitfield to have a DFT value by using the following syntax:

```
<BFNAME>       <BFRESET>    <BFTYPE> <{DFT}>  <DESCRIPTION>
```

The `<{DFT}>` portion of the bitfield declaration is actually expandable to allow a user to describe multiple DFT mode values.

The main syntax for each mode would be as follows:

```
CORESCAN:<VAL>    - Value during core_scan mode
IDDQ:<VAL>        - Value during iddq mode
HIZ:<VAL>         - Value during highz mode
BSCAN:<VAL>       - Value during bscan mode
DFT:<VAL>         - Value during all DFT modes, unless explicity set
BFLOP             - Instantiate a BSCAN Flop. If bitfield is RW then this is a drive flop
                    if bitfield is RO then this is capture flop
```

A special note about the `DFT:<VAL>` setting. This is used for cases where a user wants to say that any DFT mode not defined will have this value. It can be thought of as a "catch-all" for the DFT modes, and keep the input file simple. However, if you define `DFT:<VAL>` with any other DFT mode setting, the explicity value will be used in the respective mode.

> ***Note***
>
> **Each instance is to be separated by a 'pipe' character, and all enclosed in curly brackets.** Example: `{DFT:<VAL>|IDDQ:<VAL>}`

> ***Note***
>
> If declaring DFT modes for a bitfield that has a mux override, declare the DFT modes on the bitfield without the `_mux`

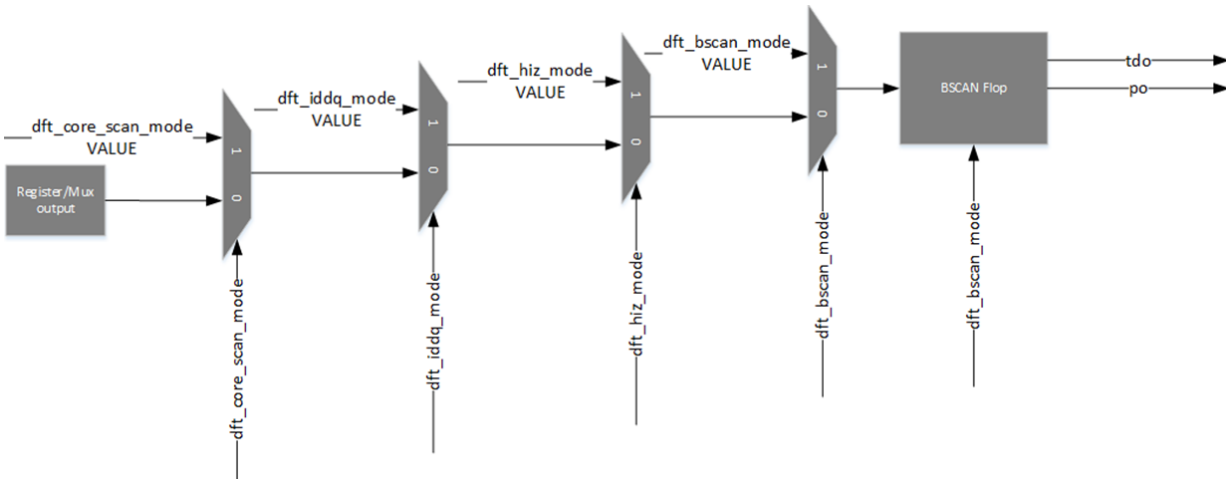
> ***Warning***
>
> RO registers can **ONLY** have Boundary Scan Flops inserted as they drive no logic in the design. Any addtional DFT modes are ignored.

DFT Priority

When declaring a bitfield for having DFT overrides, if more than one DFT mode is assigned, then the following priority is used:

1. BFLOP - If a Boundary Scan Flop is instantiated, it is last in the DFT override chain
2. BSCAN
3. HIZ
4. IDDQ
5. CORESCAN
6. Normal register operation

Below is an example of the circuit when all DFT modes are in effect.



Note

Only modes that are supported for each bitfield will have a DFT mux override instantiated. For example, you have a a CORE_SCAN mode and a BFLOP set. You will only have the CORE_SCAN mux override and a final BFLOP.

RTL Generation

If no DFT modes are set for any of the bitfields, there are no additional ports on the top level Verilog. If DFT modes are set, the additional ports are determined by modes needed for each bitfield.

This is a list of all the DFT related ports, and what would cause them to be instantiated:

```
1 //DFT Ports (if used)
2 input  wire dft_core_scan_mode,
3 input  wire dft_iddq_mode,
4 input  wire dft_hiz_mode,
5 input  wire dft_bscan_mode,
6 // BSCAN Shift Interface
7 input  wire dft_bscan_tck,
8 input  wire dft_bscan_trstn,
9 input  wire dft_bscan_capture,
10 input  wire dft_bscan_shift,
11 input  wire dft_bscan_update,
12 input  wire dft_bscan_tdi,
13 output wire dft_bscan_tdo,     //Assigned to last in chain
```

- `dft_core_scan_mode` - If `CORESCAN` is used, OR if `DFT` is used
- `dft_iddq_mode` - If `IDDQ` is used, OR if `DFT` is used
- `dft_hiz_mode` - If `HIZ` is used, OR if `DFT` is used
- `dft_bscan_mode` - If BSCAN is used, if `BFLOP` is used, OR if `DFT` is used
- `dft_bscan_* (shift interface)` - If `BFLOP` is used

Internally the muxes that are used will follow a naming convention of `clock_mux_<BFNAME>_<DFTMODE>`. Any BFLOPS are given the name `jtag_bsr_<BFNAME>`. Since bitfield names are required to be unique, there is no concern of modules with the same name.

Examples

DFT Mux Overrides

While there are several combinations of valid descriptions, here are a few examples with respective comments for what the user can expect:

```
1 REG1                RW
2   bf1               4'h3        {DFT:0}         Global setting of 0 during DFT modes
3   bf2               5'b0        {HIZ:1}         Put DFT on the non-mux. Only active in Hiz mode
4   bf2_mux           1'b0
5   bf3               1'b1        {IDDQ:0|DFT:1}  Set to 0 in IDDQ, but 1 in all other modes
6   set_core_scan     1'b0        {CORESCAN:1}    Set to 1 in CORESCAN mode
```

And this is what the RTL internals would look like:

```
 1 //---------------------------
 2 // REG1
 3 // bf1 - Global setting of 0 during DFT modes
 4 // bf2 - Put DFT on the non-mux. Only active in Hiz mode
 5 // bf2_mux -
 6 // bf3 - Set to 0 in IDDQ, but 1 in all other modes
 7 // set_core_scan - Set to 1 in CORESCAN mode
 8 //---------------------------
 9 wire [31:0] REG1_reg_read;
10 reg [3:0]   reg_bf1;
11 reg  [4:0]   reg_bf2;
12 reg         reg_bf3;
13 reg         reg_set_core_scan;
14
15 always @(posedge RegClk or posedge RegReset) begin
16   if(RegReset) begin
17     reg_bf1                              <= 4'h3;
18     reg_bf2                              <= 5'h0;
19     reg_bf2_mux                          <= 1'h0;
20     reg_bf3                              <= 1'h1;
21     reg_set_core_scan                    <= 1'h0;
22   end else if(RegAddr == 'h0 && RegWrEn) begin
23     reg_bf1                              <= RegWrData[3:0];
24     reg_bf2                              <= RegWrData[8:4];
25     reg_bf2_mux                          <= RegWrData[9];
26     reg_bf3                              <= RegWrData[10];
27     reg_set_core_scan                    <= RegWrData[11];
28   end else begin
29     reg_bf1                              <= reg_bf1;
30     reg_bf2                              <= reg_bf2;
31     reg_bf2_mux                          <= reg_bf2_mux;
32     reg_bf3                              <= reg_bf3;
33     reg_set_core_scan                    <= reg_set_core_scan;
34   end
35 end
36
37 assign REG1_reg_read = {20'h0,
38         reg_set_core_scan,
39         reg_bf3,
40         reg_bf2_mux,
41         reg_bf2,
42         reg_bf1};
43
44 //-----------------------
45
46 wire [3:0]  reg_bf1_core_scan_mode;
47 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf1_core_scan_mode[3:0] (
48   .clk0    ( reg_bf1                           ),
49   .clk1    ( 4'd0                              ),
50   .sel     ( dft_core_scan_mode                ),
51   .clk_out ( reg_bf1_core_scan_mode            ));
52
53
54 wire [3:0]  reg_bf1_iddq_mode;
55 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf1_iddq_mode[3:0] (
56   .clk0    ( reg_bf1_core_scan_mode            ),
57   .clk1    ( 4'd0                              ),
58   .sel     ( dft_iddq_mode                     ),
59   .clk_out ( reg_bf1_iddq_mode                 ));
60
61
62 wire [3:0]  reg_bf1_hiz_mode;
63 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf1_hiz_mode[3:0] (
64   .clk0    ( reg_bf1_iddq_mode                 ),
65   .clk1    ( 4'd0                              ),
66   .sel     ( dft_hiz_mode                      ),
67   .clk_out ( reg_bf1_hiz_mode                  ));
68
69
70 wire [3:0]  reg_bf1_bscan_mode;
71 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf1_bscan_mode[3:0] (
72   .clk0    ( reg_bf1_hiz_mode                  ),
73   .clk1    ( 4'd0                              ),
74   .sel     ( dft_bscan_mode                    ),
```

```
 75   .clk_out ( reg_bf1_bscan_mode                ));
 76
 77 assign swi_bf1 = reg_bf1_bscan_mode;
 78
 79 //-----------------------
 80
 81 wire [4:0]  swi_bf2_muxed_pre;
 82 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf2[4:0] (
 83   .clk0    ( bf2                               ),
 84   .clk1    ( reg_bf2                           ),
 85   .sel     ( reg_bf2_mux                       ),
 86   .clk_out ( swi_bf2_muxed_pre                 ));
 87
 88
 89 wire [4:0]  reg_bf2_hiz_mode;
 90 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf2_hiz_mode[4:0] (
 91   .clk0    ( swi_bf2_muxed_pre                 ),
 92   .clk1    ( 5'd1                              ),
 93   .sel     ( dft_hiz_mode                      ),
 94   .clk_out ( reg_bf2_hiz_mode                  ));
 95
 96 assign swi_bf2_muxed = reg_bf2_hiz_mode;
 97
 98 //-----------------------
 99 //-----------------------
100
101 wire        reg_bf3_core_scan_mode;
102 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf3_core_scan_mode (
103   .clk0    ( reg_bf3                           ),
104   .clk1    ( 1'd1                              ),
105   .sel     ( dft_core_scan_mode                ),
106   .clk_out ( reg_bf3_core_scan_mode            ));
107
108
109 wire        reg_bf3_iddq_mode;
110 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf3_iddq_mode (
111   .clk0    ( reg_bf3_core_scan_mode            ),
112   .clk1    ( 1'd0                              ),
113   .sel     ( dft_iddq_mode                     ),
114   .clk_out ( reg_bf3_iddq_mode                 ));
115
116
117 wire        reg_bf3_hiz_mode;
118 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf3_hiz_mode (
119   .clk0    ( reg_bf3_iddq_mode                 ),
120   .clk1    ( 1'd1                              ),
121   .sel     ( dft_hiz_mode                      ),
122   .clk_out ( reg_bf3_hiz_mode                  ));
123
124
125 wire        reg_bf3_bscan_mode;
126 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bf3_bscan_mode (
127   .clk0    ( reg_bf3_hiz_mode                  ),
128   .clk1    ( 1'd1                              ),
129   .sel     ( dft_bscan_mode                    ),
130   .clk_out ( reg_bf3_bscan_mode                ));
131
132 assign swi_bf3 = reg_bf3_bscan_mode;
133
134 //-----------------------
135
136 wire        reg_set_core_scan_core_scan_mode;
137 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_set_core_scan_core_scan_mode (
138   .clk0    ( reg_set_core_scan                 ),
139   .clk1    ( 1'd1                              ),
140   .sel     ( dft_core_scan_mode                ),
141   .clk_out ( reg_set_core_scan_core_scan_mode     ));
142
143 assign swi_set_core_scan = reg_set_core_scan_core_scan_mode;
```

As you can see, `bf1` is set to be `'d0` in all DFT modes, so each DFT mux is instantiated with `'d0` as the value during the respective mode.

`bf2` is only to be controlled in the `HIZ` mode, so only a mux override for `HIZ` mode is instantiated. Also note that the mux overrides for DFT occur *after* the software controlled mux override.

`bf3` is set to be `'d0` in `IDDQ` and `'d1` in all other modes. You can see that all modes have `'d1` except for the `IDDQ` mux input.

Boundary Scan Stitching

Here is an example of a user wanting to instantiate BFLOPs for several bitfields in the input file:

```
REG_WITH_BSCAN_FLOP   RW
  bscan_flop_drive    1'b0        {CORESCAN:1|BFLOP}          First in the chain since first in the file
  bscan_flop_capture  3'b0  RO    {BFLOP}                     2nd, 3rd, 4th in chain

LAST_BSCAN_FLOP       RW
  last_one_in_chain   1'b0        {BFLOP}                     Last one in the chain
```

And here is the output Verilog:

```
 1 //---------------------------
 2 // REG_WITH_BSCAN_FLOP
 3 // bscan_flop_drive - First in the chain since first in the file
 4 // bscan_flop_capture - 2nd, 3rd, 4th in chain
 5 //---------------------------
 6 wire [31:0] REG_WITH_BSCAN_FLOP_reg_read;
 7 reg         reg_bscan_flop_drive;
 8
 9 always @(posedge RegClk or posedge RegReset) begin
10   if(RegReset) begin
11     reg_bscan_flop_drive                  <= 1'h0;
12   end else if(RegAddr == 'h4 && RegWrEn) begin
13     reg_bscan_flop_drive                  <= RegWrData[0];
14   end else begin
15     reg_bscan_flop_drive                  <= reg_bscan_flop_drive;
16   end
17 end
18
19 assign REG_WITH_BSCAN_FLOP_reg_read = {28'h0,
20         bscan_flop_capture,
21         reg_bscan_flop_drive};
22
23 //-----------------------
24
25 wire        reg_bscan_flop_drive_core_scan_mode;
26 wav_clock_mux #(.STDCELL(STDCELL)) u_wav_clock_mux_bscan_flop_drive_core_scan_mode (
27   .clk0    ( reg_bscan_flop_drive              ),
28   .clk1    ( 1'd1                              ),
29   .sel     ( dft_core_scan_mode                ),
30   .clk_out ( reg_bscan_flop_drive_core_scan_mode    ));
31
32 wire  bscan_flop_drive_tdo;
33
34 wire bscan_flop_drive_bscan_flop_po;
35 wav_jtag_bsr u_wav_jtag_bsr_bscan_flop_drive (
36   .i_tck         ( dft_bscan_tck                    ),
37   .i_trst_n      ( dft_bscan_trstn                  ),
38   .i_bsr_mode    ( dft_bscan_mode                   ),
39   .i_capture     ( dft_bscan_capture                ),
40   .i_shift       ( dft_bscan_shift                  ),
41   .i_update      ( dft_bscan_update                 ),
42   .i_pi          ( reg_bscan_flop_drive_core_scan_mode    ),
43   .o_po          ( bscan_flop_drive_bscan_flop_po   ),
44   .i_tdi         ( dft_bscan_tdi                    ),
45   .o_tdo         ( bscan_flop_drive_tdo             ));
46
47
48 assign swi_bscan_flop_drive = bscan_flop_drive_bscan_flop_po;
49
50 //-----------------------
51 wire [2:0] bscan_flop_capture_tdo;
52
53 wav_jtag_bsr u_wav_jtag_bsr_bscan_flop_capture[2:0] (
54   .i_tck         ( dft_bscan_tck                    ),
55   .i_trst_n      ( dft_bscan_trstn                  ),
56   .i_bsr_mode    ( dft_bscan_mode                   ),
57   .i_capture     ( dft_bscan_capture                ),
58   .i_shift       ( dft_bscan_shift                  ),
59   .i_update      ( dft_bscan_update                 ),
60   .i_pi          ( bscan_flop_capture               ),
61   .o_po          ( /*noconn*/                       ),
62   .i_tdi         ( {bscan_flop_capture_tdo[1],
63                     bscan_flop_capture_tdo[0],
64                     bscan_flop_drive_tdo}     ),
65   .o_tdo         ( {bscan_flop_capture_tdo[2],
66                     bscan_flop_capture_tdo[1],
67                     bscan_flop_capture_tdo[0]}     ));
68
69
70
71
72
73
74 //---------------------------
75 // LAST_BSCAN_FLOP
76 // last_one_in_chain - Last one in the chain
77 //---------------------------
78 wire [31:0] LAST_BSCAN_FLOP_reg_read;
```

```
 79 reg         reg_last_one_in_chain;
 80
 81 always @(posedge RegClk or posedge RegReset) begin
 82   if(RegReset) begin
 83     reg_last_one_in_chain                 <= 1'h0;
 84   end else if(RegAddr == 'h8 && RegWrEn) begin
 85     reg_last_one_in_chain                 <= RegWrData[0];
 86   end else begin
 87     reg_last_one_in_chain                 <= reg_last_one_in_chain;
 88   end
 89 end
 90
 91 assign LAST_BSCAN_FLOP_reg_read = {31'h0,
 92         reg_last_one_in_chain};
 93
 94 //-----------------------
 95 wire  last_one_in_chain_tdo;
 96
 97 wire last_one_in_chain_bscan_flop_po;
 98 wav_jtag_bsr u_wav_jtag_bsr_last_one_in_chain (
 99   .i_tck         ( dft_bscan_tck                    ),
100   .i_trst_n      ( dft_bscan_trstn                  ),
101   .i_bsr_mode    ( dft_bscan_mode                   ),
102   .i_capture     ( dft_bscan_capture                ),
103   .i_shift       ( dft_bscan_shift                  ),
104   .i_update      ( dft_bscan_update                 ),
105   .i_pi          ( reg_last_one_in_chain            ),
106   .o_po          ( last_one_in_chain_bscan_flop_po   ),
107   .i_tdi         ( bscan_flop_capture_tdo[2]        ),
108   .o_tdo         ( last_one_in_chain_tdo            ));
109
110
111 assign swi_last_one_in_chain = last_one_in_chain_bscan_flop_po;
112
113
114 // ... excluded for clarity
115
116 //=======================
117 // Final BSCAN Connection
118 //=======================
119 assign dft_bscan_tdo = last_one_in_chain_tdo;
```

Since `bscan_flop_drive` is the first bitfield defined as a BFLOP, it will be the first one in the chain. The user also declared that this bitfield should be driven to a different value in CORESCAN mode, so a mux override for CORESCAN was placed prior to the BFLOP.

You can then see that bitfield `bscan_flop_capture` is next in the input file, so it is placed in the 2nd, 3rd, and 4th locations in the chain (since it is multi-bit). You can see that `gen_regs_py` has automatically stitched these next flops in. The PO of the BFLOP is not connected for RO bitfields, as this is taken directly to the APB read interface.

The last bitfield `last_one_in_chain` is also defined as a BFLOP and is stitched in as the last location in the chain. Since it is the last one, it is connected to `dft_bscan_tdo`, which is then connected to the next level of the design.