

Specification RFG V1.1

Tobias Markus

24.02.2015

Contents

1	Introduction	2
2	RFG API	2
2.1	User perspective	2
2.2	Developer perspective	2
3	Generator API	2
3.1	User perspective	2
3.2	Developer perspective	2
4	RFG Verilog Generator	2
4.1	Overview	2
4.2	Register	3
4.2.1	hardware/software Permissions	4
4.2.2	no_hardware_wen	7
4.2.3	software_write_clear	10
4.2.4	software_written	13
4.2.5	sticky	16
4.2.6	software_write_xor	19
4.2.7	hardware_clear	22
4.2.8	counter	25
4.2.9	rreinit_source, rreinit	29
4.3	RamBlock	33
4.3.1	attributes RamBlock	38
4.4	external/internal RegisterFiles	38

- 1 Introduction**
- 2 RFG API**
 - 2.1 User perspective**
 - 2.2 Developer perspective**
- 3 Generator API**
 - 3.1 User perspective**
 - 3.2 Developer perspective**
- 4 RFG Verilog Generator**
 - 4.1 Overview**

4.2 Register

Registers are the smallest addressable units in a registerFile. A register can consist of one or more fields with different widths and attributes. These variants will generate hardware depending on their attributes. The different attributes and the resulting hardware is given in this subsection.

```
## A register with several fields and different attributes
register test {
    field field_1 {
        width 32
        reset 32'h0
        software ro
        hardware wo
    }
    field field_2 {
        width 16
        reset 16'h0
        software rw
        hardware rw
    }
    field field_3 {
        width 16
        reset 16'h0
        software rw
    }
}
```

The size of a register can be set with an attribute in the registerFile (register_size). The default size of a register is 64 bit.

Each register in the following subsections are generated with the Generator script below:

```
package require osys::rfg 1.0.0
package require osys::generator 1.0.0

readRF [lindex $argv 0]

generator verilog {
    destinationPath "verilog/"
    options {
        reset sync
    }
}
```

4.2.1 hardware/software Permissions

The most common and important Attribute are the permissions. A register has a software and a hardware interface. Each Interface can have read and/or write permissions on a field in a register, defined with the attributes shown in the table below:

attribute name	description
ro	read only
wo	write only
rw	read and write

In this example we describe a register which has one 32 bit field with a reset value of zero and hardware read and write and software read and write permissions.

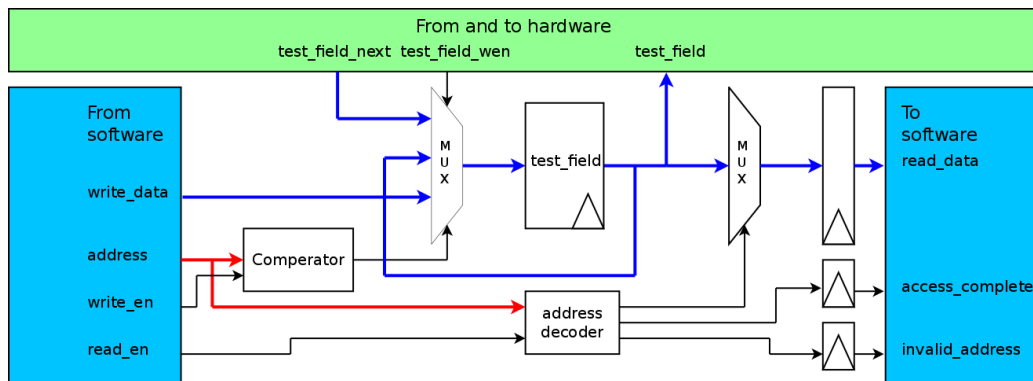
RFG Description:

```

registerFile reg_hrw_srw_hwen {
    register test {
        field test_field {
            width 32
            reset 32'h0
            software rw
            hardware rw
        }
    }
}

```

Block Diagramm:



Generated verilog from RFG description:

```
1  module reg_hrw_srw_nhwen
2  (
3      //Software Interface
4      input wire res_n ,
5      input wire clk ,
6      input wire[3:3] address ,
7      output reg[31:0] read_data ,
8      output reg invalid_address ,
9      output reg access_complete ,
10     input wire read_en ,
11     input wire write_en ,
12     input wire[31:0] write_data ,
13     // Hardware Interface
14     input wire[31:0] test_test_field_next ,
15     input wire test_test_field_wen ,
16     output reg[31:0] test_test_field
17 );
18
19     /* register test */
20     always @(posedge clk)
21     begin
22         if (!res_n)
23         begin
24             test_test_field <= 32'h0;
25         end
26         else
27         begin
28             if ((address[3:3]== 0) && write_en)
29             begin
30                 test_test_field <= write_data[31:0];
31             end
32             else if(test_test_field_wen)
33             begin
34                 test_test_field <= test_test_field_next;
35             end
36         end
37     end
38
39     always @(posedge clk)
40     begin
41         if (!res_n)
42         begin
```

```

43             invalid_address <= 1'b0;
44             access_complete <= 1'b0;
45         end
46     else
47     begin
48
49         casex( address [3:3])
50             1'h0:
51             begin
52                 read_data [31:0] <= test_test_field;
53                 invalid_address <= 1'b0;
54                 access_complete <= write_en || read_en;
55             end
56         default:
57         begin
58             invalid_address <= read_en || write_en;
59             access_complete <= read_en || write_en;
60         end
61     endcase
62 end
63 end
64 endmodule

```

Depending on the permission attributes the verilog output is slightly different.

The always block from line 21 to line 38, represents the software write and hardware write functionality to one field. If the field have no software write permissions line 29 to line 32 are not generated. If the field has no hardware write permission line 33 to line 35 are not generated. If the field has neither a software write nor a hardware write only the reset logic is generated. If the field also does not have a reset attribute the always block is not generated. These descriptions without any hardware or software permissions are used to define reserved fields in a register.

The hardware read is generated with the output reg on line 16 if there is no hardware read permission this signal is generated as internal reg.

In the second always block the address decoder for the software read is generated. Depending on the read permission line 51 is generated or not.

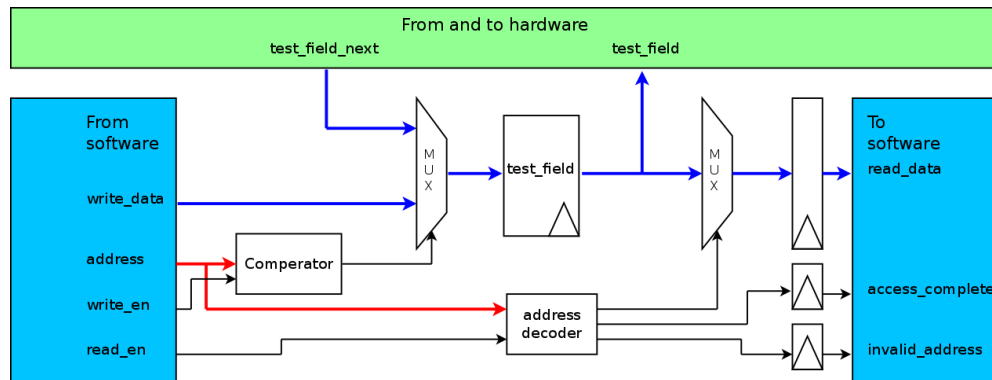
4.2.2 no_hardware_wen

With the no_hardware_wen attribute the hardware generator will not generate the hardware write enable signal on the register hardware interface. Attention when you write something with the software the hardware will rewrite the register in the next clock cycle.

RFG Description:

```
registerFile reg_hrw_srw_nhwen {  
    register test {  
        field test_field {  
            width 32  
            reset 32'h0  
            software rw  
            hardware {  
                rw  
                no_hardware_wen  
            }  
        }  
    }  
}
```

Block Diagramm:



Generated verilog from RFG description:

```
1  module reg_hrw_srw_nhwen
2  (
3      // Software Interface
4      input wire res_n ,
5      input wire clk ,
6      input wire[3:3] address ,
7      output reg[31:0] read_data ,
8      output reg invalid_address ,
9      output reg access_complete ,
10     input wire read_en ,
11     input wire write_en ,
12     input wire[31:0] write_data ,
13     // Hardware Interface
14     input wire[31:0] test_test_field_next ,
15     output reg[31:0] test_test_field
16 );
17
18
19     /* register test */
20     always @(posedge clk)
21     begin
22         if (!res_n)
23         begin
24             test_test_field <= 32'h0;
25         end
26         else
27         begin
28
29             if((address[3:3]== 0) && write_en)
30             begin
31                 test_test_field <= write_data[31:0];
32             end
33             else
34             begin
35                 test_test_field <= test_test_field_next;
36             end
37         end
38     end
39
40     always @(posedge clk)
41     begin
42         if (!res_n)
```



```

43         begin
44             invalid_address <= 1'b0;
45             access_complete <= 1'b0;
46         end
47         else
48             begin
49                 casex( address[3:3])
50                     1'h0:
51                         begin
52                             read_data[31:0] <= test_test_field;
53                             invalid_address <= 1'b0;
54                             access_complete <= write_en || read_en;
55                         end
56                     default:
57                         begin
58                             invalid_address <= read_en || write_en;
59                             access_complete <= read_en || write_en;
60                         end
61                 endcase
62             end
63         end
64     endmodule

```

The difference in this verilog output can be observed in line 33. Now there is no hardware write enable signal the register is written on each clock cycle. Keep this in mind if you write it via Software. The Hardware has then one cycle to react to it and then to rewrite the field.

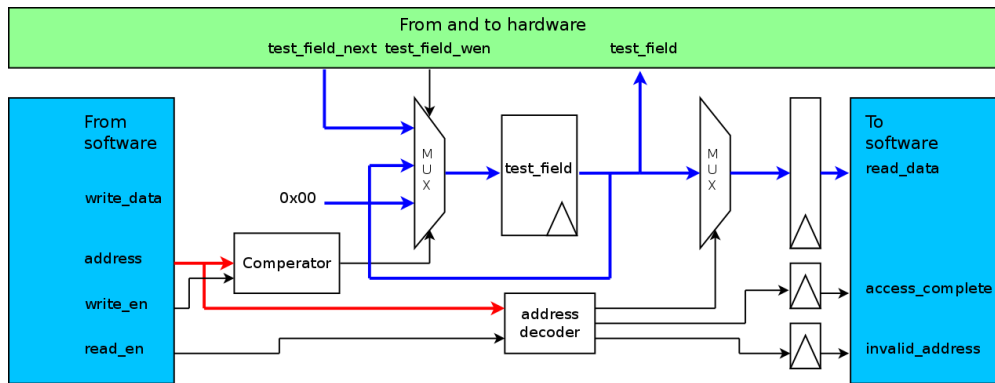
4.2.3 software_write_clear

With the software_write_clear attribute the field is cleared on a software write operation.

RFG Description:

```
registerFile reg_hrw_srw_swrite_clear {  
    register test {  
        field test_field {  
            width 32  
            reset 32'h0  
            software {  
                rw  
                software_write_clear  
            }  
            hardware rw  
        }  
    }  
}
```

Block Diagramm:



Generated verilog from RFG description:

```
1 module reg_hrw_srw_swrite_clear
2 (
3     input wire res_n ,
4     input wire clk ,
5     // Software Interface
6     input wire[3:3] address ,
7     output reg[31:0] read_data ,
8     output reg invalid_address ,
9     output reg access_complete ,
10    input wire read_en ,
11    input wire write_en ,
12    input wire[31:0] write_data ,
13    // Hardware Interface
14    input wire[31:0] test_test_field_next ,
15    input wire test_test_field_wen ,
16    output reg[31:0] test_test_field
17 );
18
19 /* register test */
20 always @(posedge clk)
21 begin
22     if (!res_n)
23     begin
24         test_test_field <= 32'h0;
25     end
26     else
27     begin
28         if ((address[3:3]== 0) && write_en)
29         begin
30             test_test_field <= 32'h0;
31         end
32         else if(test_test_field_wen)
33         begin
34             test_test_field <= test_test_field_next;
35         end
36     end
37 end
38
39 always @(posedge clk)
40 begin
41     if (!res_n)
42     begin
```

```

43         invalid_address <= 1'b0;
44         access_complete <= 1'b0;
45     end
46     else
47     begin
48         casex(address[3:3])
49             1'h0:
50             begin
51                 read_data[31:0] <= test_test_field;
52                 invalid_address <= 1'b0;
53                 access_complete <= write_en || read_en;
54             end
55             default:
56             begin
57                 invalid_address <= read_en || write_en;
58                 access_complete <= read_en || write_en;
59             end
60         endcase
61     end
62 end
63 endmodule

```

In line 30 we can see that now the register is cleared when the register is written from the software.

4.2.4 software_written

With the software_written signal an additional hardware output is generated which is high when the software writes the register and depending on its value also when the register is reset. Otherwise the software_written signal is low.

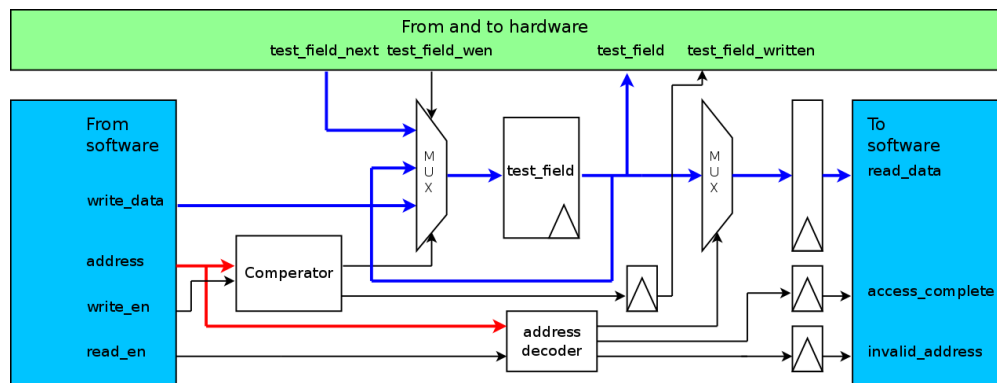
RFG Description:

```

registerFile reg_hrw_srw_swritten {
    register test {
        field test_field {
            width 32
            reset 32'h0
            software rw
            hardware {
                rw
                software_written
            }
        }
    }
}

```

Block Diagramm:



Generated verilog from RFG description:

```
1  module reg_hrw_srw_swritten
2  (
3      input wire res_n ,
4      input wire clk ,
5      // Software Interface
6      input wire[3:3] address ,
7      output reg[31:0] read_data ,
8      output reg invalid_address ,
9      output reg access_complete ,
10     input wire read_en ,
11     input wire write_en ,
12     input wire[31:0] write_data ,
13     // Hardware Interface
14     input wire[31:0] test_test_field_next ,
15     input wire test_test_field_wen ,
16     output reg[31:0] test_test_field ,
17     output reg test_test_field_written
18 );
19
20
21     /* register test */
22     always @(posedge clk)
23     begin
24         if (!res_n)
25         begin
26             test_test_field <= 32'h0;
27             test_test_field_written <= 1'b0;
28         end
29         else
30         begin
31
32             if((address[3:3]== 0) && write_en)
33             begin
34                 test_test_field <= write_data[31:0];
35                 test_test_field_written <= 1'b1;
36             end
37             else if(test_test_field_wen)
38             begin
39                 test_test_field <= test_test_field_next;
40                 test_test_field_written <= 1'b0;
41             end
42             else
```

```

43         begin
44             test_test_field_written <= 1'b0;
45         end
46
47     end
48 end
49
50 always @(posedge clk)
51 begin
52     if (!res_n)
53     begin
54         invalid_address <= 1'b0;
55         access_complete <= 1'b0;
56     end
57     else
58     begin
59
60         casex(address[3:3])
61             1'h0:
62             begin
63                 read_data[31:0] <= test_test_field;
64                 invalid_address <= 1'b0;
65                 access_complete <= write_en || read_en;
66             end
67             default:
68             begin
69                 invalid_address <= read_en || write_en;
70                 access_complete <= read_en || write_en;
71             end
72         endcase
73     end
74 end
75 endmodule

```

In this verilog output an additional hardware output signal is added (line 17). It is set when the software interface writes the field, line 32 to 36. It is reset on every cycle in which the software interface does not do a write operation. In this example the software_written attribute is configured to output a zero, when the register is resetted. It can also be configured to output a one.

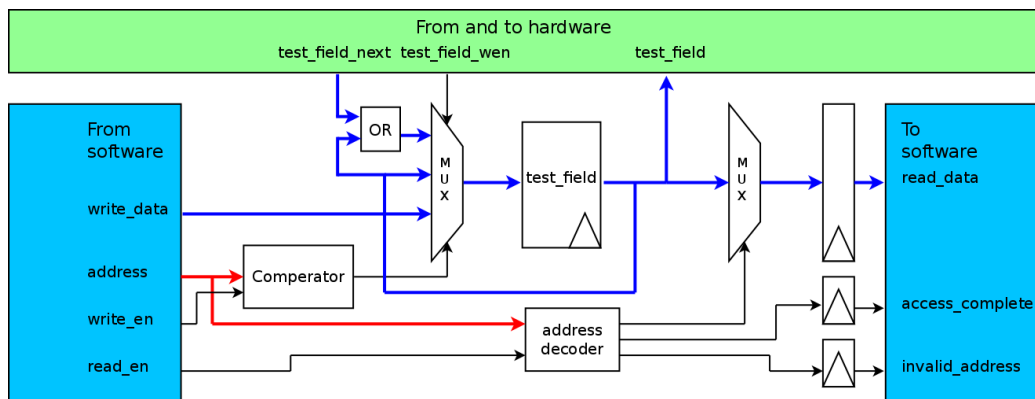
4.2.5 sticky

With the sticky feature a field is generated in which the bits in the field can only be set from the hardware. The field can only be reset from the software interface or with a reset.

RFG Description:

```
registerFile reg_hrw_srw_sticky {  
    register test {  
        field test_field {  
            width 32  
            reset 32'h0  
            software rw  
            hardware {  
                rw  
                sticky  
            }  
        }  
    }  
}
```

Block Diagramm:



Generated verilog from RFG description:

```
1  module reg_hrw_srw_sticky
2  (
3      input wire res_n ,
4      input wire clk ,
5      // Software Interface
6      input wire[3:3] address ,
7      output reg[31:0] read_data ,
8      output reg invalid_address ,
9      output reg access_complete ,
10     input wire read_en ,
11     input wire write_en ,
12     input wire[31:0] write_data ,
13     // Hardware Interface
14     input wire[31:0] test_test_field_next ,
15     input wire test_test_field_wen ,
16     output reg[31:0] test_test_field
17 );
18
19
20     /* register test */
21     always @(posedge clk)
22     begin
23         if (!res_n)
24         begin
25             test_test_field <= 32'h0;
26         end
27         else
28         begin
29
30             if((address[3:3]== 0) && write_en)
31             begin
32                 test_test_field <= write_data[31:0];
33             end
34             else if(test_test_field_wen)
35             begin
36                 test_test_field <= test_test_field_next |
37                     test_test_field;
38             end
39         end
40     end
41     always @(posedge clk)
```

```

42     begin
43         if (!res_n)
44             begin
45                 invalid_address <= 1'b0;
46                 access_complete <= 1'b0;
47             end
48         else
49             begin
50                 casex(address[3:3])
51                     1'h0:
52                         begin
53                             read_data[31:0] <= test_test_field;
54                             invalid_address <= 1'b0;
55                             access_complete <= write_en || read_en;
56                         end
57                     default:
58                         begin
59                             invalid_address <= read_en || write_en;
60                             access_complete <= read_en || write_en;
61                         end
62                     endcase
63             end
64         end
65     endmodule

```

The difference in the verilog output with the sticky attribute is that the new hardware value is ored on write with the register value itself line 36 to line 37.

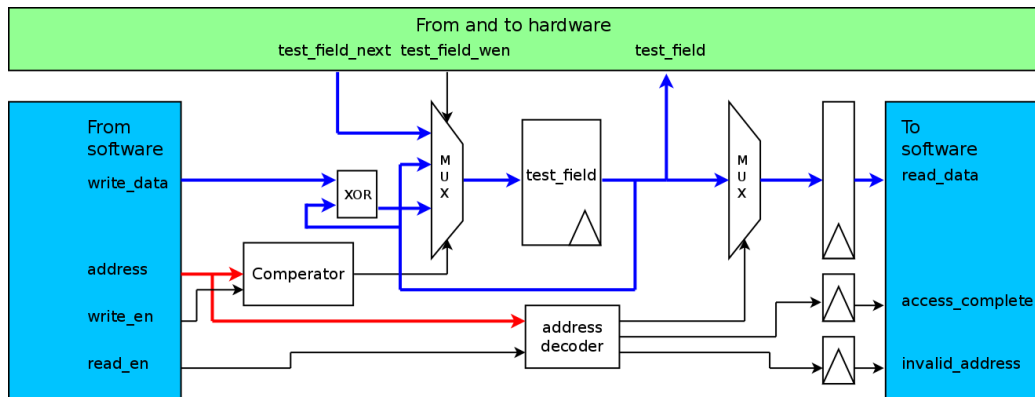
4.2.6 software_write_xor

The software_write_xor attributes writes on a software write the new value xored with the register value.

RFG Description:

```
registerFile reg_hrw_srw_swrite_xor {  
    register test {  
        field test_field {  
            width 32  
            reset 32'h0  
            software rw  
            hardware {  
                rw  
                software_write_xor  
            }  
        }  
    }  
}
```

Block Diagramm:



Generated verilog from RFG description:

```
1  module reg_hrw_srw_swrite_xor
2  (
3      input wire res_n ,
4      input wire clk ,
5      // Software Interface
6      input wire[3:3] address ,
7      output reg[31:0] read_data ,
8      output reg invalid_address ,
9      output reg access_complete ,
10     input wire read_en ,
11     input wire write_en ,
12     input wire[31:0] write_data ,
13     // Hardware Interface
14     input wire[31:0] test_test_field_next ,
15     input wire test_test_field_wen ,
16     output reg[31:0] test_test_field
17 );
18
19
20     /* register test */
21     always @(posedge clk) 'endif
22     begin
23         if (!res_n)
24         begin
25             test_test_field <= 32'h0;
26         end
27         else
28         begin
29
30             if((address[3:3]== 0) && write_en)
31             begin
32                 test_test_field <= (write_data[31:0] ^
33                                     test_test_field);
34             end
35             else if(test_test_field_wen)
36             begin
37                 test_test_field <= test_test_field_next;
38             end
39         end
40     end
41     always @(posedge clk)
```

```

42     begin
43         if (!res_n)
44             begin
45                 invalid_address <= 1'b0;
46                 access_complete <= 1'b0;
47             end
48         else
49             begin
50
51                 casex(address[3:3])
52                     1'h0:
53                         begin
54                             read_data[31:0] <= test_test_field;
55                             invalid_address <= 1'b0;
56                             access_complete <= write_en || read_en;
57                         end
58                     default:
59                         begin
60                             invalid_address <= read_en || write_en;
61                             access_complete <= read_en || write_en;
62                         end
63                 endcase
64             end
65         end
66     endmodule

```

The `software_write_xor` attribute does also just do small change. When the register is written form the software interface it gets xored with itself line 32.

4.2.7 hardware_clear

The hardware_clear attribute adds an additional signal to the hardware interface. This signal clears the register when the hardware_clear signal is asserted.

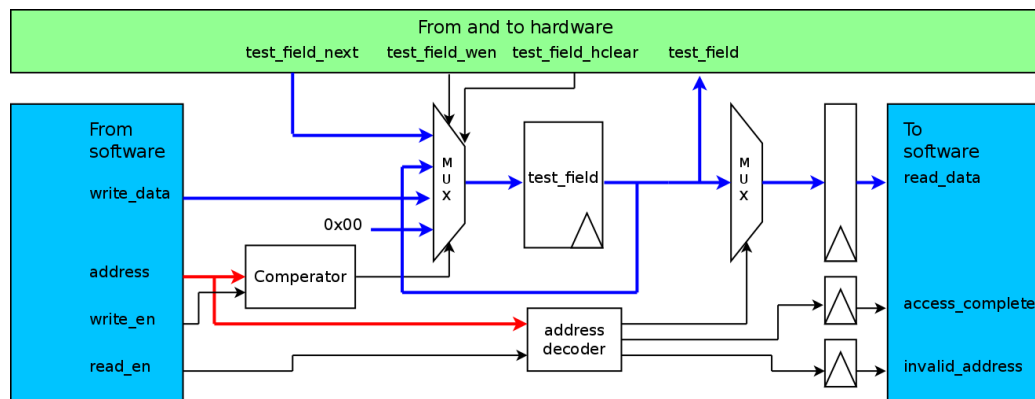
RFG Description:

```

registerFile reg_hrw_srw_hclear {
    register test {
        field test_field {
            width 32
            reset 32'h0
            software rw
            hardware {
                rw
                hardware_clear
            }
        }
    }
}

```

Block Diagramm:



Generated verilog from RFG description:

```
1  module reg_hrw_srw_hclear
2  (
3      input wire res_n ,
4      input wire clk ,
5      // Software Interface
6      input wire[3:3] address ,
7      output reg[31:0] read_data ,
8      output reg invalid_address ,
9      output reg access_complete ,
10     input wire read_en ,
11     input wire write_en ,
12     input wire[31:0] write_data ,
13     // Hardware Interface
14     input wire[31:0] test_test_field_next ,
15     input wire test_test_field_wen ,
16     output reg[31:0] test_test_field ,
17     input wire test_test_field_clear
18 );
19 );
20
21     /* register test */
22     always @(posedge clk)
23     begin
24         if (!res_n)
25         begin
26             test_test_field <= 32'h0;
27         end
28         else
29         begin
30             if((address[3:3]== 0) && write_en)
31             begin
32                 test_test_field <= write_data[31:0];
33             end
34             else if(test_test_field_clear)
35             begin
36                 test_test_field <= 32'h0;
37             end
38             else if(test_test_wen)
39             begin
40                 test_test_field <= test_test_field_next;
41             end
42         end
43     end
```

```

43     end
44
45     always @(posedge clk)
46     begin
47         if (!res_n)
48         begin
49             invalid_address <= 1'b0;
50             access_complete <= 1'b0;
51         end
52         else
53         begin
54             casex(address[3:3])
55             1'h0:
56             begin
57                 read_data[31:0] <= test_test_field;
58                 invalid_address <= 1'b0;
59                 access_complete <= write_en || read_en;
60             end
61             default:
62             begin
63                 invalid_address <= read_en || write_en;
64                 access_complete <= read_en || write_en;
65             end
66         endcase
67     end
68 end
69 endmodule

```


4.2.8 counter

The counter attribute transforms the internal register into a counter with count up signal for the hardware interface.

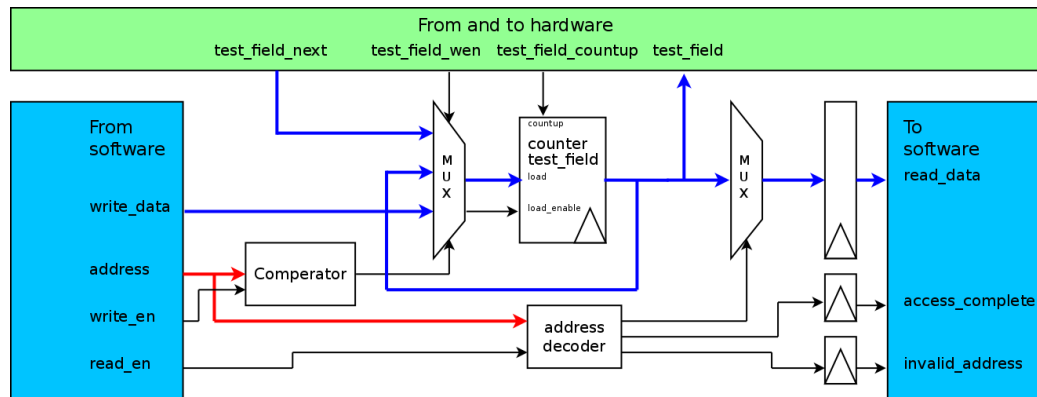
RFG Description:

```

registerFile reg_hrw_srw_counter {
    register test {
        field test_field {
            width 32
            reset 32'h0
            software rw
            hardware {
                rw
                counter
            }
        }
    }
}

```

Block Diagramm:



Generated verilog from RFG description:

```
1 module reg_hrw_srw_counter
2 (
3     input wire clk ,
4     input wire res_n ,
5     // Software Interface
6     input wire[3:3] address ,
7     output reg[31:0] read_data ,
8     output reg invalid_address ,
9     output reg access_complete ,
10    input wire read_en ,
11    input wire write_en ,
12    input wire[31:0] write_data ,
13    // Hardware Interface
14    input wire[31:0] test_test_field_next ,
15    output wire[31:0] test_test_field ,
16    input wire test_test_field_wen ,
17    input wire test_test_field_countup
18 );
19
20    reg test_test_field_load_enable;
21    reg[31:0] test_test_field_load_value;
22
23    counter48 #(
24        .DATASIZE(32)
25    ) test_test_field_I (
26        .clk(clk) ,
27        .res_n(res_n) ,
28        .increment(test_test_field_countup) ,
29        .load(test_test_field_load_value) ,
30        .load_enable(test_test_field_load_enable) ,
31        .value(test_test_field)
32    );
33
34    /* register test */
35    always @(posedge clk)
36    begin
37        if (!res_n)
38        begin
39            test_test_field_load_enable <= 1'b0;
40        end
41        else
42        begin
```

```

43
44         if((address[3:3]== 0) && write_en)
45         begin
46             test_test_field_load_enable <= 1'b1;
47             test_test_field_load_value <= write_data
               [31:0];
48         end
49         else if(test_test_field_wen)
50         begin
51             test_test_field_load_value <=
               test_test_field_next;
52             test_test_field_load_enable <= 1'b1;
53         end
54         else
55         begin
56             test_test_field_load_enable <= 1'b0;
57             test_test_field_load_value <= 32'b0;
58         end
59     end
60 end
61
62 always @(posedge clk)
63 begin
64     if (!res_n)
65     begin
66         invalid_address <= 1'b0;
67         access_complete <= 1'b0;
68     end
69     else
70     begin
71
72         casex(address[3:3])
73         1'h0:
74         begin
75             read_data[31:0] <= test_test_field;
76             invalid_address <= 1'b0;
77             access_complete <= write_en || read_en;
78         end
79         default:
80         begin
81             invalid_address <= read_en || write_en;
82             access_complete <= read_en || write_en;
83         end

```

```
84         endcase
85     end
86 end
87 endmodule
```

4.2.9 rreinit_source, rreinit

With the rreinit_source and rreinit combination a register with the rreinit_source attribute can be generated which resets a counter field marked with the rreinit attribute.

RFG Description:

```
registerFile reg_hrw_srw_rreinit_source {  
  
    register counter_rreinit {  
        hardware {  
            rreinit_source  
        }  
    }  
  
    register example {  
        field test_field {  
            width 32  
            reset 32'h0  
            software rw  
            hardware {  
                rw  
                counter  
                rreinit  
            }  
        }  
    }  
}
```

Generated verilog from RFG description:

```
1  module reg_hrw_srw_rreinit_source
2  (
3      input wire res_n ,
4      input wire clk ,
5      // Software Interface
6      input wire[3:3] address ,
7      output reg[31:0] read_data ,
8      output reg invalid_address ,
9      output reg access_complete ,
10     input wire read_en ,
11     input wire write_en ,
12     input wire[31:0] write_data ,
13     // Hardware Interface
14     input wire[31:0] example_test_field_next ,
15     output wire[31:0] example_test_field ,
16     input wire example_test_field_wen ,
17     input wire example_test_field_countup
18 );
19
20
21     reg rreinit;
22     reg example_test_field_load_enable;
23     reg[31:0] example_test_field_load_value;
24
25     counter48 #(
26         .DATASIZE(32)
27     ) example_test_field_I (
28         .clk(clk) ,
29         .res_n(res_n) ,
30         .increment(example_test_field_countup) ,
31         .load(example_test_field_load_value) ,
32         .load_enable(rreinit ||
33             example_test_field_load_enable) ,
34         .value(example_test_field)
35     );
36
37     /* register counter_rreinit */
38     always @(posedge clk)
39     begin
40         if (!res_n)
41             begin
42                 rreinit <= 1'b0;
```

```

42         end
43     else
44         begin
45
46             if ((address[3:3]== 0) && write_en)
47                 begin
48                     rreinit <= 1'b1;
49                 end
50             else
51                 begin
52                     rreinit <= 1'b0;
53                 end
54             end
55         end
56
57         /* register example */
58         always @(posedge clk)
59         begin
60             if (!res_n)
61                 begin
62                     example_test_field_load_enable <= 1'b0;
63                 end
64             else
65                 begin
66
67                     if ((address[3:3]== 1) && write_en)
68                         begin
69                             example_test_field_load_enable <= 1'b1;
70                             example_test_field_load_value <= write_data
71                                 [31:0];
72                         end
73                     else if (example_test_field_wen)
74                         begin
75                             example_test_field_load_value <=
76                                 example_test_field_next;
77                             example_test_field_load_enable <= 1'b1;
78                         end
79                     else
80                         begin
81                             example_test_field_load_enable <= 1'b0;
82                             example_test_field_load_value <= 32'b0;
83                         end
84                     end
85                 end
86             end
87         end
88     end
89 end

```

```

83     end
84
85     always @(posedge clk)
86     begin
87         if (!res_n)
88         begin
89             invalid_address <= 1'b0;
90             access_complete <= 1'b0;
91         end
92         else
93         begin
94
95             casex(address[3:3])
96             1'h1:
97             begin
98                 read_data[31:0] <= example_test_field;
99                 invalid_address <= 1'b0;
100                access_complete <= write_en || read_en;
101            end
102            default:
103            begin
104                invalid_address <= read_en || write_en;
105                access_complete <= read_en || write_en;
106            end
107        endcase
108    end
109 end
110 endmodule

```


4.3 RamBlock

A RamBlock is a construct which implements an addressspace as RAM inside the hardware. In different to registers, the hardware interface has now also address, data, and control lines. Depending on the read/write Permissions different RAMs are used. See the table below. 1w_1r_1c means a Ram with one write, one read interface. and 2rw_1c means a dual port Ram.

permissions	none	ro	wo	rw	hardware
none	none	none	none	1w_1r_1c	
ro	none	none	1w_1r_1c	2rw_1c	
wo	none	1w_1r_1c	none	2rw_1c	
rw	1w_1r_1c	2rw_1c	2rw_1c	2rw_1c	
software					

RFG Description:

```

registerFile RamBlock {
    register test {
        field test_field {
            width 32
            hardware rw
            software rw
        }
    }
    ramBlock test_ram {
        depth 128
        width 32
        hardware rw
        software rw
    }
}

```

Generated verilog from RFG description:

```
1  module RamBlock
2  (
3      input wire clk ,
4      input wire res_n ,
5      // Software Interface
6      input wire[10:3] address ,
7      output reg[31:0] read_data ,
8      output reg invalid_address ,
9      output reg access_complete ,
10     input wire read_en ,
11     input wire write_en ,
12     input wire[31:0] write_data ,
13     // Hardware Interface
14     input wire[31:0] test_test_field_next ,
15     output reg[31:0] test_test_field ,
16     input wire[6:0] test_ram_addr ,
17     input wire test_ram_ren ,
18     output wire[31:0] test_ram_rdata ,
19     input wire test_ram_wen ,
20     input wire[31:0] test_ram_wdata
21
22 );
23
24     reg[6:0] test_ram_rf_addr;
25     reg test_ram_rf_ren;
26     wire[31:0] test_ram_rf_rdata;
27     reg test_ram_rf_wen;
28     reg[31:0] test_ram_rf_wdata;
29     reg read_en_dly0;
30     reg read_en_dly1;
31     reg read_en_dly2;
32
33     ram_2rw_1c #(
34         .DATASIZE(32) ,
35         .ADDRSIZE(7) ,
36         .PIPELINED(0)
37     ) test_ram (
38         .clk(clk) ,
39         .wen_a(test_ram_rf_wen) ,
40         .ren_a(test_ram_rf_ren) ,
41         .addr_a(test_ram_rf_addr) ,
42         .wdata_a(test_ram_rf_wdata) ,
```

```

43         .rdata_a(test_ram_rf_rdata),
44         .wen_b(test_ram_wen),
45         .ren_b(test_ram_ren),
46         .addr_b(test_ram_addr),
47         .wdata_b(test_ram_wdata),
48         .rdata_b(test_ram_rdata)
49     );
50
51
52     /* register test */
53     always @(posedge clk)
54     begin
55         if (!res_n)
56         begin
57             test_test_field <= 0;
58         end
59         else
60         begin
61
62             if ((address[10:3]== 0) && write_en)
63             begin
64                 test_test_field <= write_data[31:0];
65             end
66             else
67             begin
68                 test_test_field <= test_test_field_next;
69             end
70         end
71     end
72
73     /* RamBlock test_ram */
74     always @(posedge clk)
75     begin
76         if (!res_n)
77         begin
78             `ifdef ASIC
79                 test_ram_rf_addr <= 7'b0;
80                 test_ram_rf_wdata <= 32'b0;
81             `endif
82                 test_ram_rf_wen <= 1'b0;
83                 test_ram_rf_ren <= 1'b0;
84         end
85         else

```

```

86         begin
87             if (address[10:10] == 1)
88                 begin
89                     test_ram_rf_addr <= address[9:3];
90                     test_ram_rf_wdata <= write_data[31:0];
91                     test_ram_rf_wen <= write_en;
92                     test_ram_rf_ren <= read_en;
93                 end
94             end
95         end
96
97         always @(posedge clk)
98         begin
99             if (!res_n)
100                 begin
101                     invalid_address <= 1'b0;
102                     access_complete <= 1'b0;
103
104                     read_en_dly0 <= 1'b0;
105                     read_en_dly1 <= 1'b0;
106                     read_en_dly2 <= 1'b0;
107                 end
108             else
109                 begin
110                     read_en_dly0 <= read_en;
111                     read_en_dly1 <= read_en_dly0;
112                     read_en_dly2 <= read_en_dly1;
113
114                     casex(address[10:3])
115                         8'h0:
116                             begin
117                                 read_data[31:0] <= test_test_field;
118                                 invalid_address <= 1'b0;
119                                 access_complete <= write_en || read_en;
120                             end
121                         {1'h1,7'bxxxxxxx}:
122                             begin
123                                 read_data[31:0] <= test_ram_rf_rdata;
124                                 invalid_address <= 1'b0;
125                                 access_complete <= write_en ||
126                                     read_en_dly2;
127                             end
128                         default:

```

```

128             begin
129                 invalid_address <= read_en || write_en;
130                 access_complete <= read_en || write_en;
131             end
132         endcase
133     end
134 end
135 endmodule

```

4.3.1 attributes RamBlock

external attribute:

With the external attribute there is just a RamBlock Interface generated to communicate with a RamBlock outside the registerfile.

address_shift attribute:

The address_shift attributes allows to address the Ram Block with a shift inside the registerfile address space. In example each ramblock entry each 4kB with (address_shift 12)

4.4 external/internal RegisterFiles