# 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 ore 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:

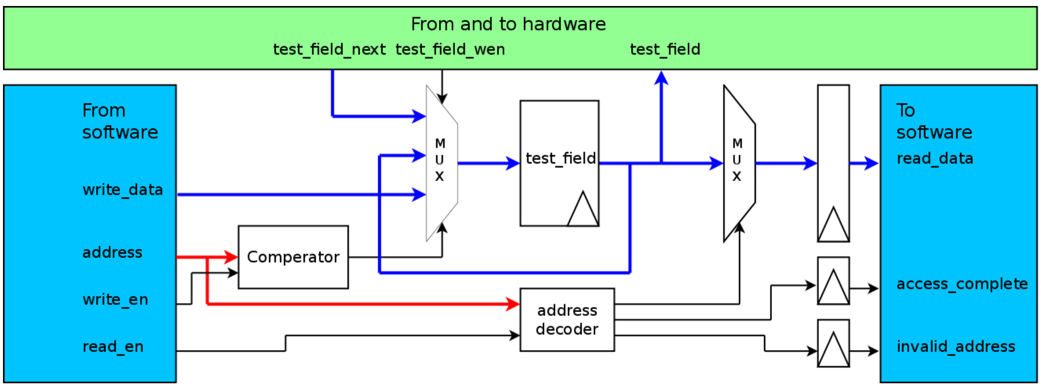
| attribute name | description |
|---|---|
| ro | read only |
| wo | write only |
| rw | read and write |

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:

```
module reg_hrw_srw_nhwen
(
    //Software Interface
    input wire res_n,
    input wire clk,
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    input wire test_test_field_wen,
    output reg[31:0] test_test_field
);

    /* register test */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            test_test_field <= 32'h0;
        end
        else
        begin
            if((address[3:3]== 0) && write_en)
            begin
                test_test_field <= write_data[31:0];
            end
            else if(test_test_field_wen)
            begin
                test_test_field <= test_test_field_next;
            end
        end
    end

    always @(posedge clk)
    begin
        if (!res_n)
        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.

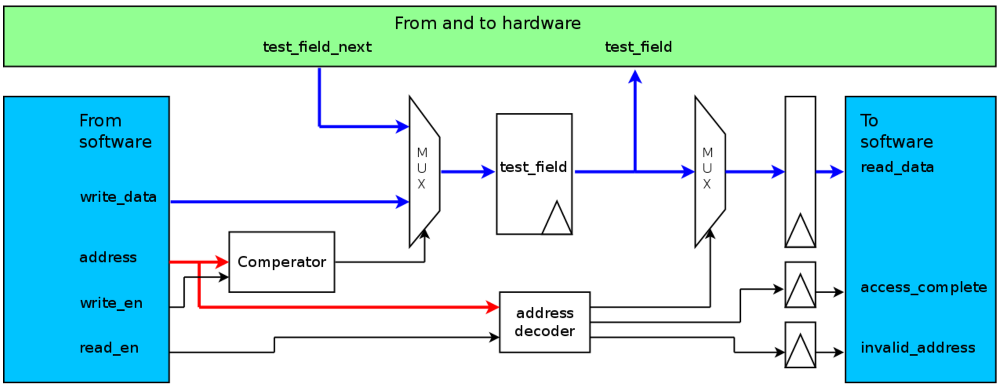
### 4.2.2 no_hardware_wen

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:

```
module reg_hrw_srw_nhwen
(
    // Software Interface
    input wire res_n,
    input wire clk,
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    output reg[31:0] test_test_field

);

    /* register test */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            test_test_field <= 32'h0;
        end
        else
        begin

            if((address[3:3]== 0) && write_en)
            begin
                test_test_field <= write_data[31:0];
            end
            else
            begin
                test_test_field <= test_test_field_next;
            end
        end
    end

    always @(posedge clk)
    begin
        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.

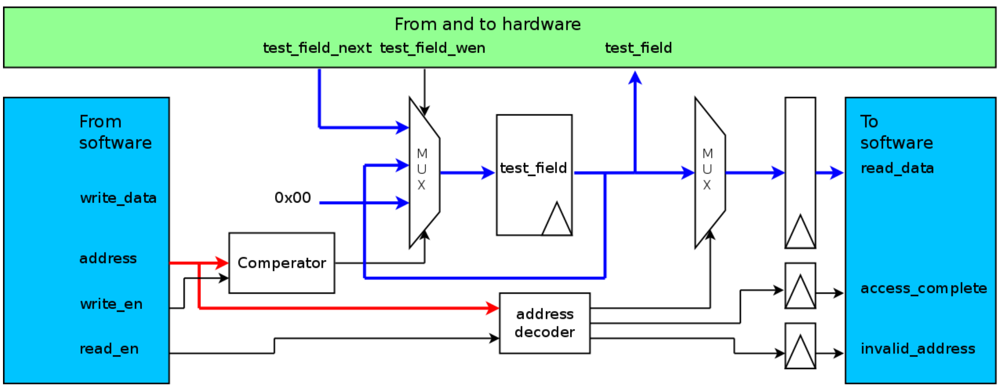
### 4.2.3 software_write_clear

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

RFG Descpription:

```
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:

```
module reg_hrw_srw_swrite_clear
(
    input wire res_n,
    input wire clk,
    // Software Interface
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    input wire test_test_field_wen,
    output reg[31:0] test_test_field
);

    /* register test */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            test_test_field <= 32'h0;
        end
        else
        begin
            if((address[3:3]== 0) && write_en)
            begin
                test_test_field <= 32'h0;
            end
            else if(test_test_field_wen)
            begin
                test_test_field <= test_test_field_next;
            end
        end
    end

    always @(posedge clk)
    begin
        if (!res_n)
        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.

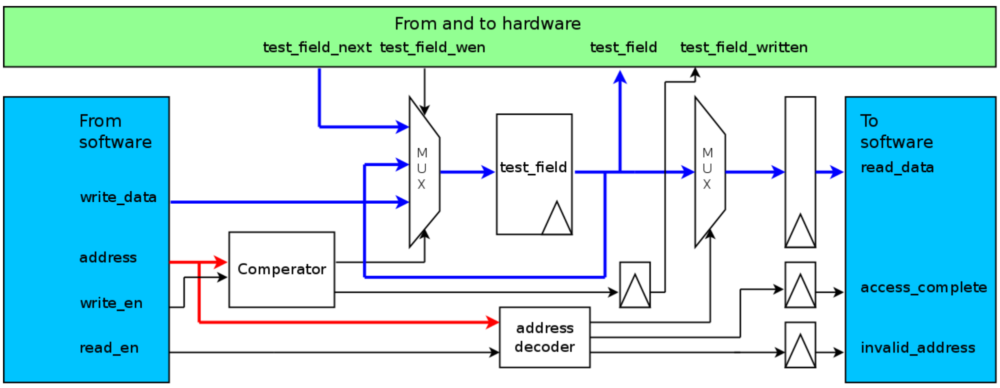
### 4.2.4 software_written

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 reseted. 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:

```
module reg_hrw_srw_swritten
(
    input wire res_n,
    input wire clk,
    // Software Interface
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    input wire test_test_field_wen,
    output reg[31:0] test_test_field,
    output reg test_test_field_written

);

    /* register test */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            test_test_field <= 32'h0;
            test_test_field_written <= 1'b0;
        end
        else
        begin

            if((address[3:3]== 0) && write_en)
            begin
                test_test_field <= write_data[31:0];
                test_test_field_written <= 1'b1;
            end
            else if(test_test_field_wen)
            begin
                test_test_field <= test_test_field_next;
                test_test_field_written <= 1'b0;
            end
            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.

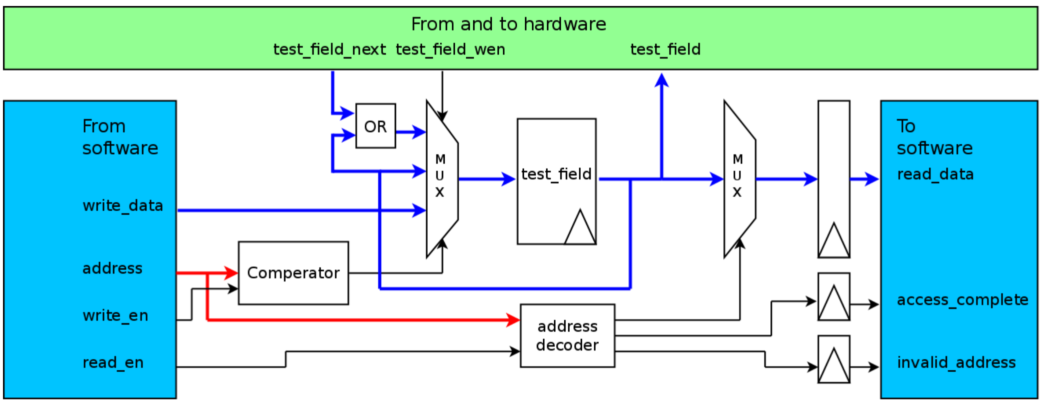
### 4.2.5 sticky

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 reseted 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:

```verilog
module reg_hrw_srw_sticky
(
    input wire res_n,
    input wire clk,
    // Software Interface
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    input wire test_test_field_wen,
    output reg[31:0] test_test_field

);

    /* register test */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            test_test_field <= 32'h0;
        end
        else
        begin

            if((address[3:3]== 0) && write_en)
            begin
                test_test_field <= write_data[31:0];
            end
            else if(test_test_field_wen)
            begin
                test_test_field <= test_test_field_next |
                    test_test_field;
            end
        end
    end

    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.

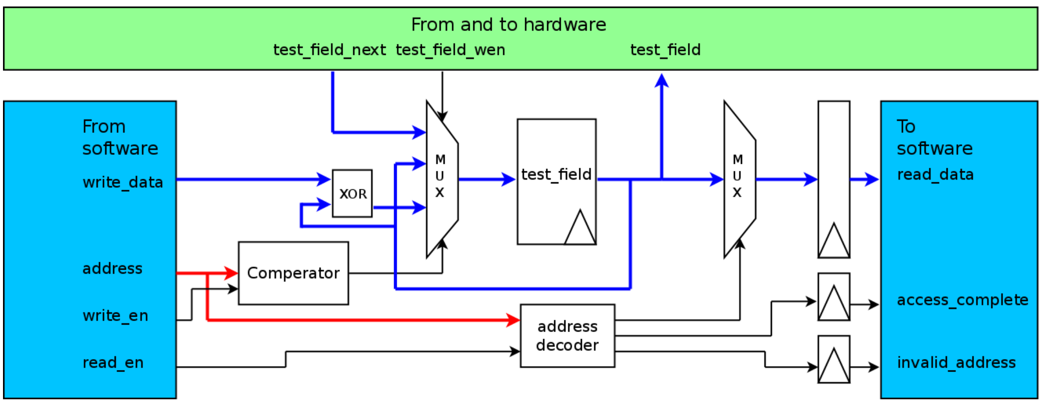
### 4.2.6 software_write_xor

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:

```
module reg_hrw_srw_swrite_xor
(
    input wire res_n,
    input wire clk,
    // Software Interface
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    input wire test_test_field_wen,
    output reg[31:0] test_test_field

);

    /* register test */
    always @(posedge clk) `endif
    begin
        if (!res_n)
        begin
            test_test_field <= 32'h0;
        end
        else
        begin

            if((address[3:3]== 0) && write_en)
            begin
                test_test_field <= (write_data[31:0] ^ test_test_field);
            end
            else if(test_test_field_wen)
            begin
                test_test_field <= test_test_field_next;
            end
        end
    end

    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.

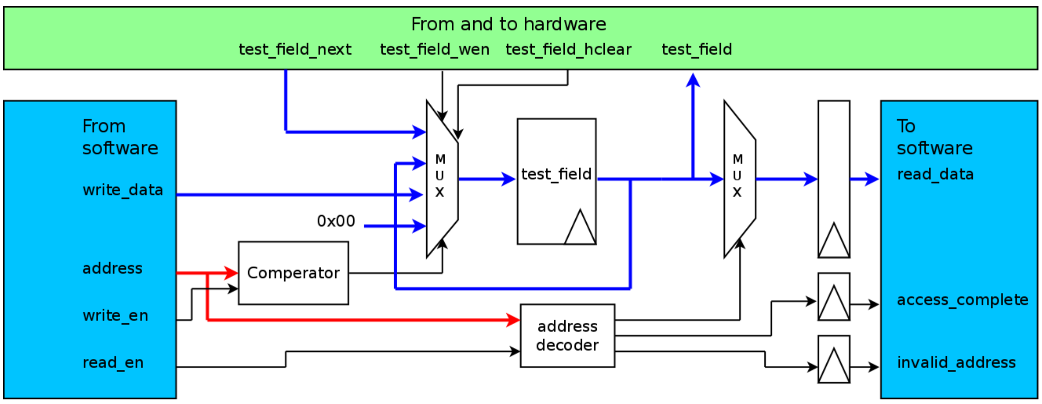
### 4.2.7 hardware_clear

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:

```
module reg_hrw_srw_hclear
(
    input wire res_n,
    input wire clk,
    // Software Interface
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    input wire test_test_field_wen,
    output reg[31:0] test_test_field,
    input wire test_test_field_clear

);

    /* register test */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            test_test_field <= 32'h0;
        end
        else
        begin
            if((address[3:3]== 0) && write_en)
            begin
                test_test_field <= write_data[31:0];
            end
            else if(test_test_field_clear)
            begin
                test_test_field <= 32'h0;
            end
            else if(test_test_wen)
            begin
                test_test_field <= test_test_field_next;
            end
        end
```

```
    end

    always @(posedge clk)
    begin
        if (!res_n)
        begin
            invalid_address <= 1'b0;
            access_complete <= 1'b0;
        end
        else
        begin
            casex(address[3:3])
                1'h0:
                begin
                    read_data[31:0] <= test_test_field;
                    invalid_address <= 1'b0;
                    access_complete <= write_en || read_en;
                end
                default:
                begin
                    invalid_address <= read_en || write_en;
                    access_complete <= read_en || write_en;
                end
            endcase
        end
    end
endmodule
```

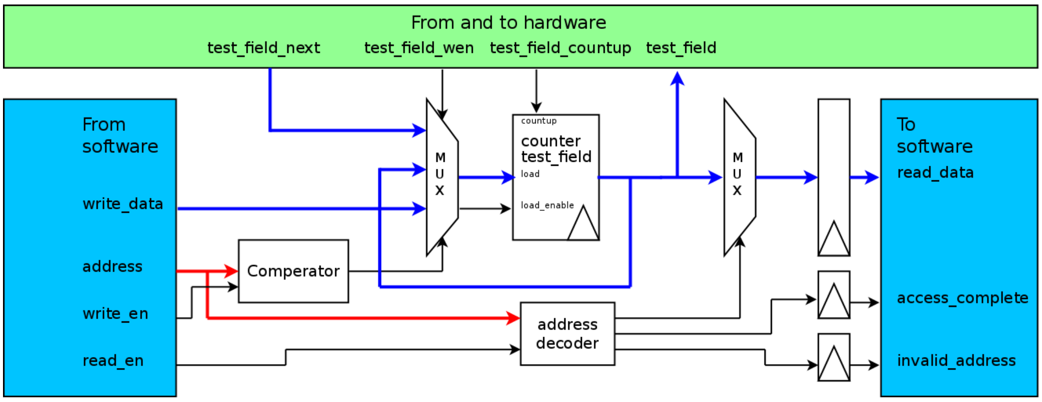
### 4.2.8 counter

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:

```
module reg_hrw_srw_counter
(
    input wire clk,
    input wire res_n,
    // Software Interface
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    output wire[31:0] test_test_field,
    input wire test_test_field_wen,
    input wire test_test_field_countup
);

    reg test_test_field_load_enable;
    reg[31:0] test_test_field_load_value;

    counter48 #(
        .DATASIZE(32)
    ) test_test_field_I (
        .clk(clk),
        .res_n(res_n),
        .increment(test_test_field_countup),
        .load(test_test_field_load_value),
        .load_enable(test_test_field_load_enable),
        .value(test_test_field)
    );

    /* register test */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            test_test_field_load_enable <= 1'b0;
        end
        else
        begin
```

```

            if((address[3:3]== 0) && write_en)
            begin
                test_test_field_load_enable <= 1'b1;
                test_test_field_load_value <= write_data[31:0];
            end
            else if(test_test_field_wen)
            begin
                test_test_field_load_value <= test_test_field_next;
                test_test_field_load_enable <= 1'b1;
            end
            else
            begin
                test_test_field_load_enable <= 1'b0;
                test_test_field_load_value <= 32'b0;
            end
        end
    end

    always @(posedge clk)
    begin
        if (!res_n)
        begin
            invalid_address <= 1'b0;
            access_complete <= 1'b0;
        end
        else
        begin

            casex(address[3:3])
                1'h0:
                begin
                    read_data[31:0] <= test_test_field;
                    invalid_address <= 1'b0;
                    access_complete <= write_en || read_en;
                end
                default:
                begin
                    invalid_address <= read_en || write_en;
                    access_complete <= read_en || write_en;
                end
```

```
            endcase
        end
    end
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:

```
module reg_hrw_srw_rreinit_source
(
    input wire res_n,
    input wire clk,
    // Software Interface
    input wire[3:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] example_test_field_next,
    output wire[31:0] example_test_field,
    input wire example_test_field_wen,
    input wire example_test_field_countup

);

    reg rreinit;
    reg example_test_field_load_enable;
    reg[31:0] example_test_field_load_value;

    counter48 #(
        .DATASIZE(32)
    ) example_test_field_I (
        .clk(clk),
        .res_n(res_n),
        .increment(example_test_field_countup),
        .load(example_test_field_load_value),
        .load_enable(rreinit || example_test_field_load_enable),
        .value(example_test_field)
    );

    /* register counter_rreinit */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            rreinit <= 1'b0;
```

```
        end
        else
        begin

            if((address[3:3]== 0) && write_en)
            begin
                rreinit <= 1'b1;
            end
            else
            begin
                rreinit <= 1'b0;
            end
        end
    end

    /* register example */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            example_test_field_load_enable <= 1'b0;
        end
        else
        begin

            if((address[3:3]== 1) && write_en)
            begin
                example_test_field_load_enable <= 1'b1;
                example_test_field_load_value <= write_data[31:0];
            end
            else if(example_test_field_wen)
            begin
                example_test_field_load_value <= example_test_field_next;
                example_test_field_load_enable <= 1'b1;
            end
            else
            begin
                example_test_field_load_enable <= 1'b0;
                example_test_field_load_value <= 32'b0;
            end
        end
```

```
    end

    always @(posedge clk)
    begin
        if (!res_n)
        begin
            invalid_address <= 1'b0;
            access_complete <= 1'b0;
        end
        else
        begin

            casex(address[3:3])
                1'h1:
                begin
                    read_data[31:0] <= example_test_field;
                    invalid_address <= 1'b0;
                    access_complete <= write_en || read_en;
                end
                default:
                begin
                    invalid_address <= read_en || write_en;
                    access_complete <= read_en || write_en;
                end
            endcase
        end
    end
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:

```
module RamBlock
(
    input wire clk,
    input wire res_n,
    // Software Interface
    input wire[10:3] address,
    output reg[31:0] read_data,
    output reg invalid_address,
    output reg access_complete,
    input wire read_en,
    input wire write_en,
    input wire[31:0] write_data,
    // Hardware Interface
    input wire[31:0] test_test_field_next,
    output reg[31:0] test_test_field,
    input wire[6:0] test_ram_addr,
    input wire test_ram_ren,
    output wire[31:0] test_ram_rdata,
    input wire test_ram_wen,
    input wire[31:0] test_ram_wdata

);

    reg[6:0] test_ram_rf_addr;
    reg test_ram_rf_ren;
    wire[31:0] test_ram_rf_rdata;
    reg test_ram_rf_wen;
    reg[31:0] test_ram_rf_wdata;
    reg read_en_dly0;
    reg read_en_dly1;
    reg read_en_dly2;

    ram_2rw_1c #(
        .DATASIZE(32),
        .ADDRSIZE(7),
        .PIPELINED(0)
    ) test_ram (
        .clk(clk),
        .wen_a(test_ram_rf_wen),
        .ren_a(test_ram_rf_ren),
        .addr_a(test_ram_rf_addr),
        .wdata_a(test_ram_rf_wdata),
```

```
        .rdata_a(test_ram_rf_rdata),
        .wen_b(test_ram_wen),
        .ren_b(test_ram_ren),
        .addr_b(test_ram_addr),
        .wdata_b(test_ram_wdata),
        .rdata_b(test_ram_rdata)
    );


    /* register test */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            test_test_field <= 0;
        end
        else
        begin

            if((address[10:3]== 0) && write_en)
            begin
                test_test_field <= write_data[31:0];
            end
            else
            begin
                test_test_field <= test_test_field_next;
            end
        end
    end

    /* RamBlock test_ram */
    always @(posedge clk)
    begin
        if (!res_n)
        begin
            `ifdef ASIC
            test_ram_rf_addr <= 7'b0;
            test_ram_rf_wdata  <= 32'b0;
            `endif
            test_ram_rf_wen <= 1'b0;
            test_ram_rf_ren <= 1'b0;
        end
        else
```

```
        begin
            if (address[10:10] == 1)
            begin
                test_ram_rf_addr <= address[9:3];
                test_ram_rf_wdata <= write_data[31:0];
                test_ram_rf_wen <= write_en;
                test_ram_rf_ren <= read_en;
            end
        end
    end

    always @(posedge clk)
    begin
        if (!res_n)
        begin
            invalid_address <= 1'b0;
            access_complete <= 1'b0;

            read_en_dly0 <= 1'b0;
            read_en_dly1 <= 1'b0;
            read_en_dly2 <= 1'b0;
        end
        else
        begin
            read_en_dly0 <= read_en;
            read_en_dly1 <= read_en_dly0;
            read_en_dly2 <= read_en_dly1;

            casex(address[10:3])
                8'h0:
                begin
                    read_data[31:0] <= test_test_field;
                    invalid_address <= 1'b0;
                    access_complete <= write_en || read_en;
                end
                {1'h1,7'bxxxxxxx}:
                begin
                    read_data[31:0] <= test_ram_rf_rdata;
                    invalid_address <= 1'b0;
                    access_complete <= write_en || read_en_dly2;
                end
                default:
```

```
                    begin
                        invalid_address <= read_en || write_en;
                        access_complete <= read_en || write_en;
                    end
                endcase
            end
        end
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

A registerfile can be included in another one. There are two different constructs to include the registerfile internal or external.

RFG Description:

```
registerFile RF {
    external RamBlock.rf RamBlockRF_external
    internal RamBlock.rf RamBlockRF_internal
}
```

Generated verilog from RFG description: