Specification RFG V1.1

Tobias Markus

24.02.2015

Contents

1 RFG Hardware Generation

1.1 Overview

1.2 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 of generated hardware depending on the attributes is described 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
    }
}
```

1.2.1 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 defined with the attributes shown in the table below:

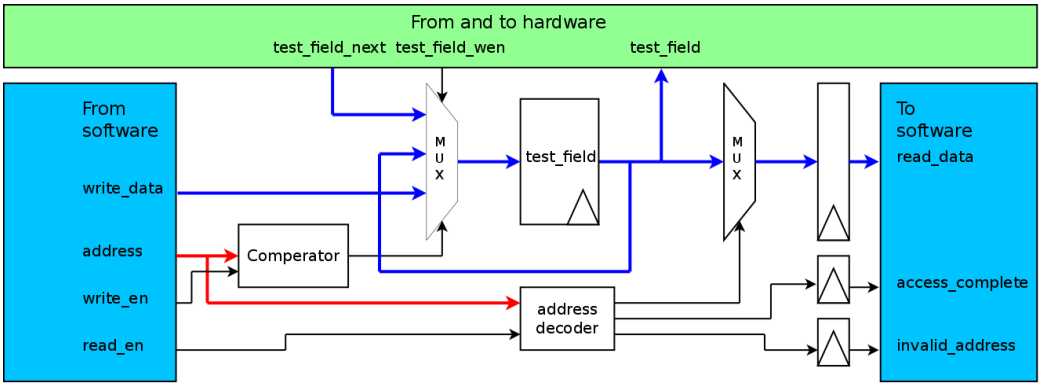
attribute name	description
ro	read only
wo	write only
rw	read and write

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 functionallaty 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.

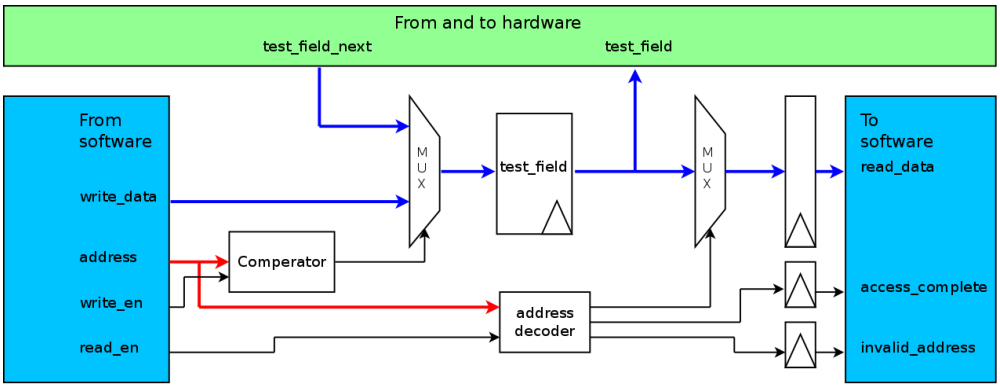
1.2.2 no_hardware_wen

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.

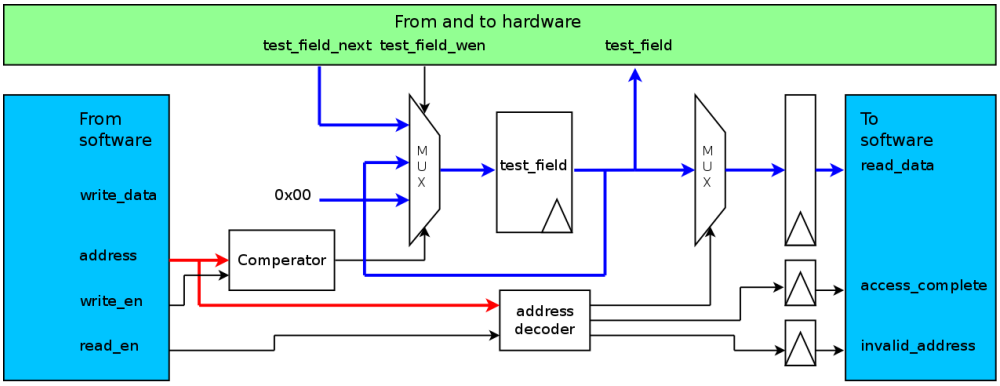
1.2.3 software_write_clear

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:

```
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.

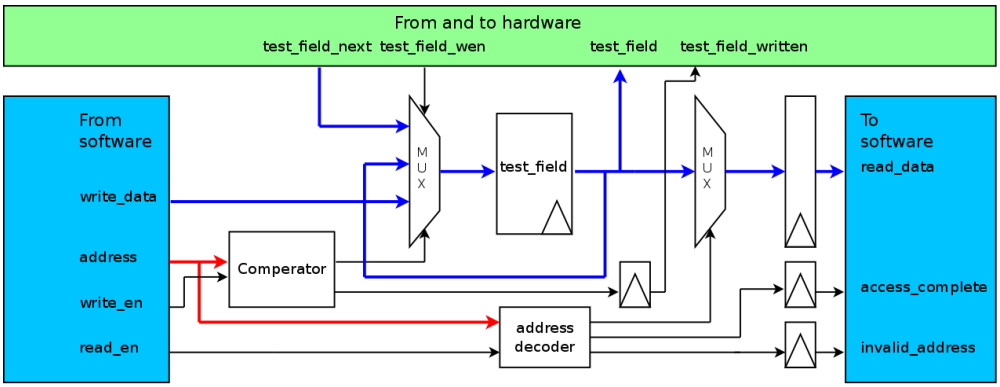
1.2.4 software_written

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 resetted. 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.

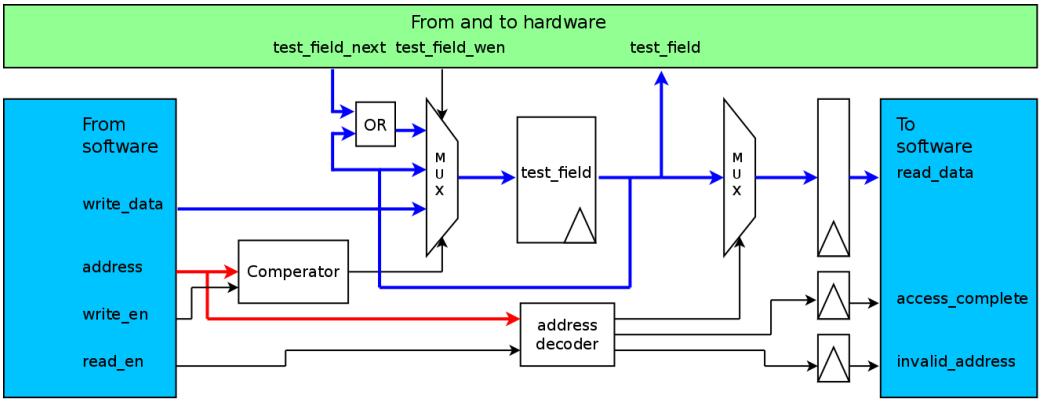
1.2.5 sticky

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 resetted 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 |
                       test_test_field;
37             end
38         end
39     end
40
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 diffirence 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.

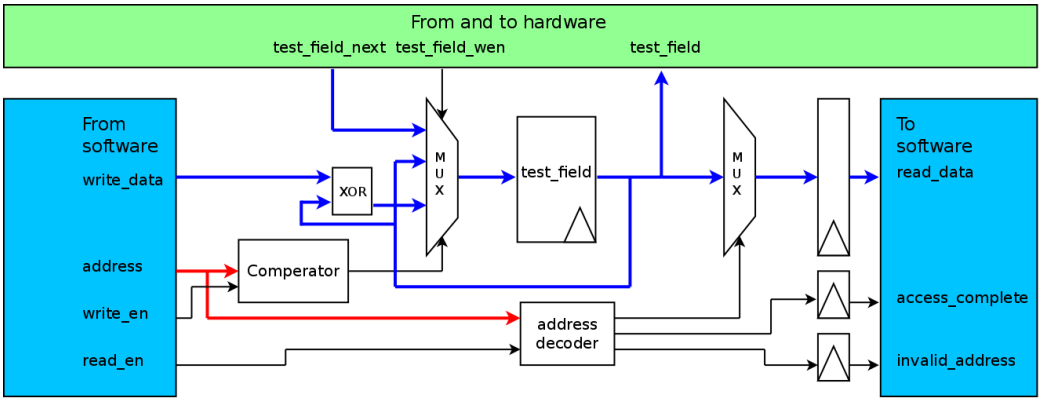
1.2.6 software_write_xor

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] ^
                       test_test_field);
33             end
34             else if(test_test_field_wen)
35             begin
36                 test_test_field <= test_test_field_next;
37             end
38         end
39     end
40
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.

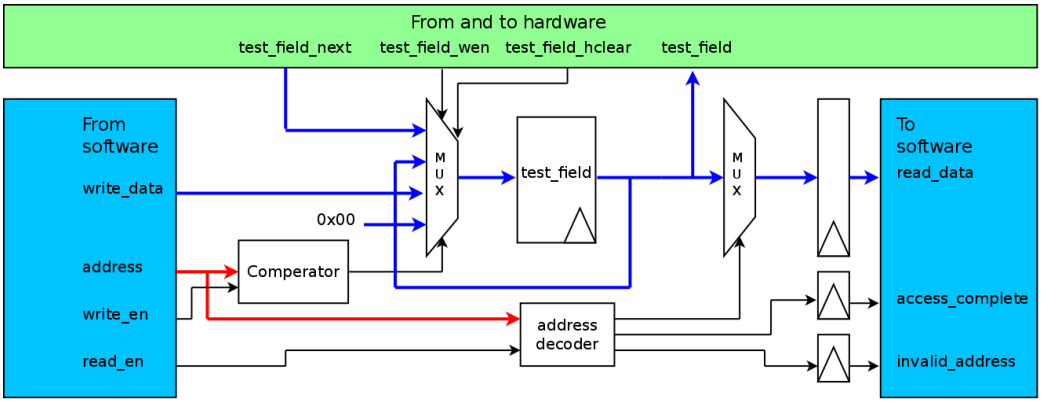
1.2.7 hardware_clear

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
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
```

1.2.8 counter

1.2.9 rreinit_source, rreinit

1.3 RamBlock

1.4 external/internal RegisterFiles