Введение в проектирование RTL цифровых систем средствами Chisel/Scala

Денис Муратов
Ведущий инженер по разработке СнК

2024

Введение

Основы Chisel

Функциональное программирование

Повторное использование

Итоги

System Verilog

1. синтезируемое подмножество ограничено
2. слабая портируемость кода
3. нет ООП и других техник современных ЯП
4. нет единой точки входа для IP
5. избыточность языковых конструкций

Chisel

Chisel – это не HLS (High Level Synthesis)

Chisel

Constructing Hardware in Scala Embedded Language

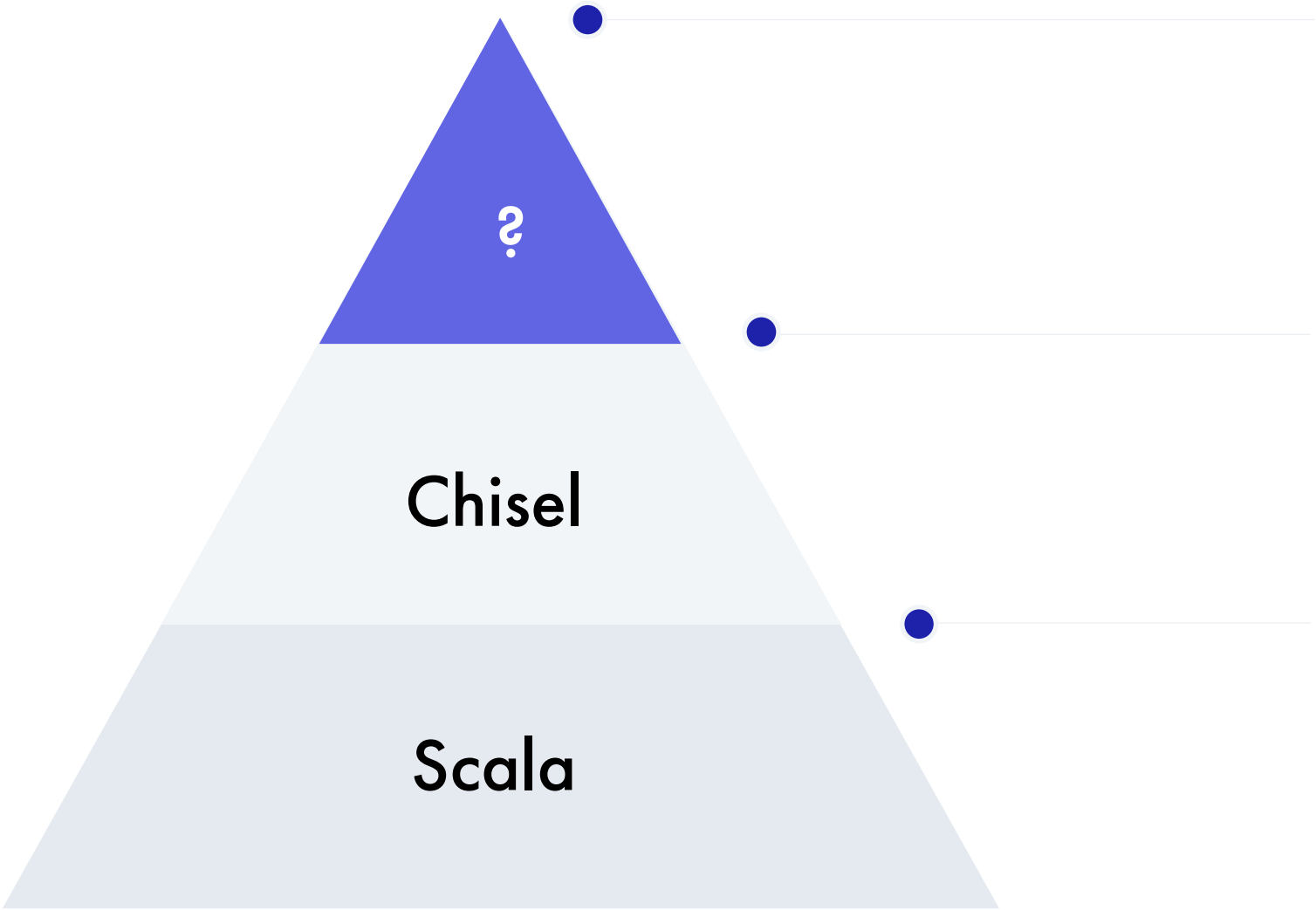
- Open source hardware description language (HDL) used to describe digital electronics and circuits at the register-transfer level

Wikipedia (c)

Smaller design teams create larger designs

DARPA (c)

Scala – scalable language



HLS

Ориентирован под предметную область

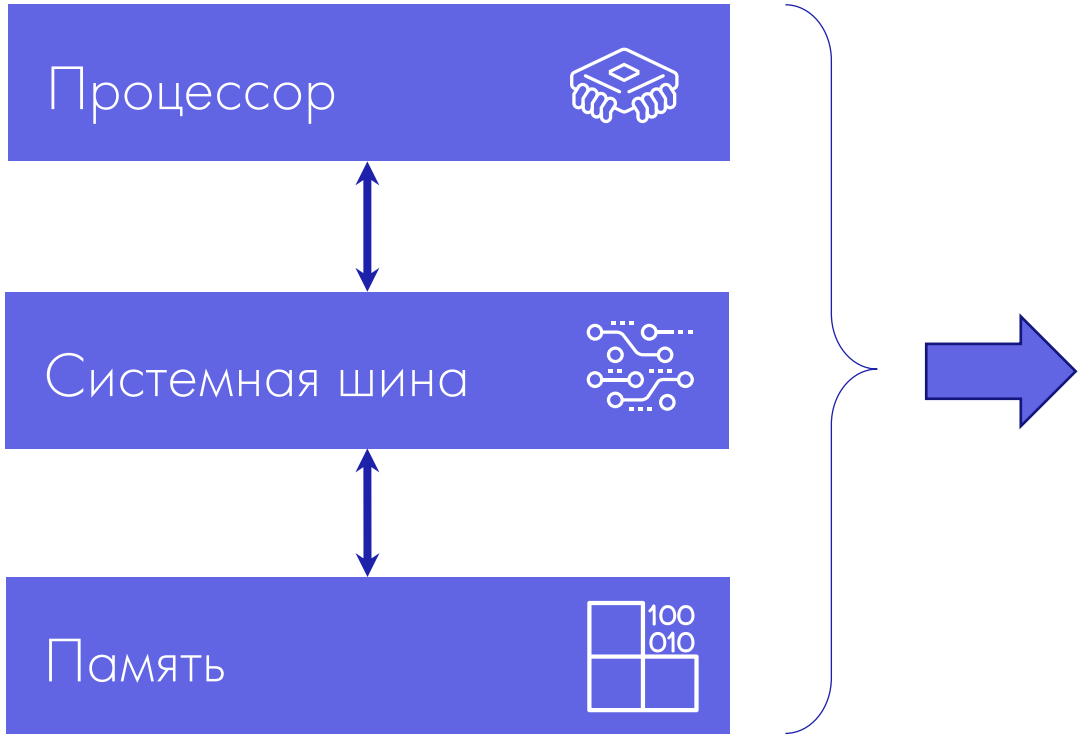
Библиотека

- генерация RTL описаний
- симуляция RTL

ЯП

- исполняется на JVM
- ФП, библиотеки Java

Простая цифровая система (System Verilog)

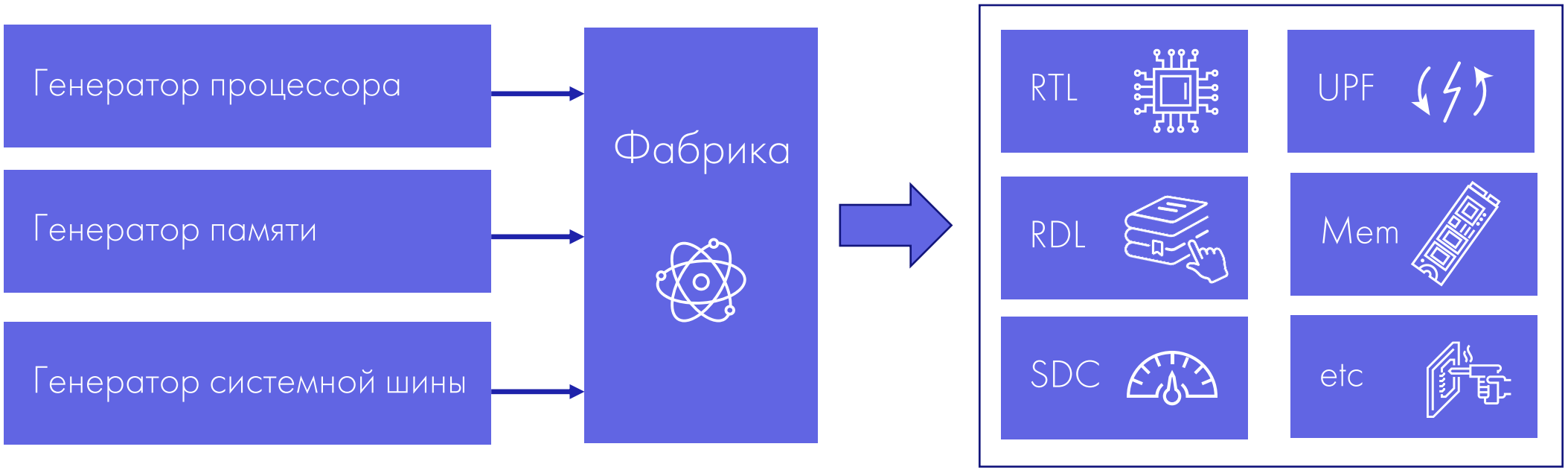


Простая цифровая система (Chisel)

```
(f << c(cp).* << b(bp).* << m(mp).*).!
```

Простая цифровая система (Chisel)

```
1: (fabric <<
2:   CpuGen(cpuParam).getAll <<
3:   SysBusGen(sysBusParam).getAll <<
4:   SysMemGen(sysMemParam).getAll
5: ).generate
```



Введение

Основы Chisel

Функциональное программирование

Повторное использование

Итоги

Основы Chisel: одна шина

System Verilog

```
logic [WIDTH-1:0] data;
```

Chisel

```
val data = Wire(UInt(Width.W))
```

Основы Chisel: две шины

System Verilog

```
1: logic [WIDTH-1:0]     data;
2: logic [(WIDTH+1)-1:0] data_inc;

3: assign data_inc = data + 1'b1;
```

Chisel

```
4: val data     = Wire(UInt(Width.W))
5: val data_inc = WireInit(data +& 1.U)
```

Основы Chisel: D-триггер

System Verilog

```
logic my_ff;

always_ff @(negedge rst_n or posedge clk)
  if (!rst_n)
    my_ff <= 1'b0;
  else
    my_ff <= my_ff_next;
```

Chisel

```
val my_ff = RegNext(my_ff_next, false.B)
```

Основы Chisel: D-триггер

System Verilog

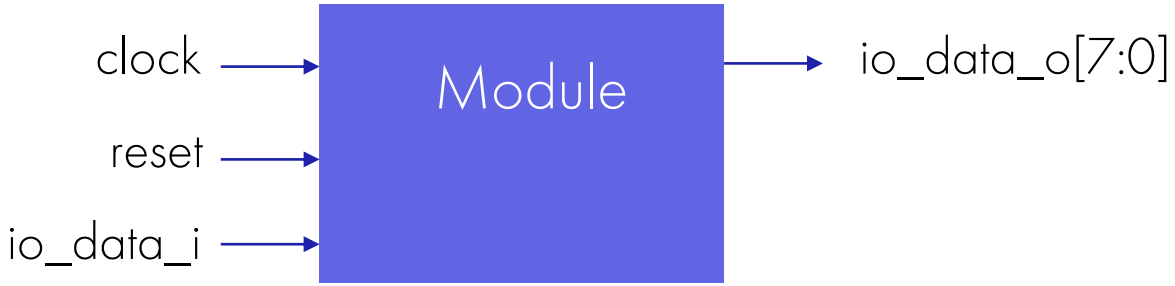
```
logic my_ff;

always_ff @(negedge rst_n or posedge clk)
  if (!rst_n)
    my_ff <= 1'b0;
  else
    my_ff <= my_ff_next;
```

Chisel

```
val my_ff = RegNext(my_ff_next, false.B)
```

Где clock?

Основы Chisel: модуль и интерфейс



System Verilog

```
module my_module #(
  parameter WIDTH = 8
) (
  input                 clock,
  input                 reset,
  input                 io_data_i,
  output [WIDTH-1:0] io_data_o
);
   ...
endmodule
```

Chisel

```
1: class my_module (dataWidth: Int = 8)
2:     extends Module {

3:   val io = IO(new Bundle {
4:     val data_i  = Input(Bool())
5:     val data_o  = Output(UInt(dataWidth.W))
6:   })

     ...
7: }
```

Введение

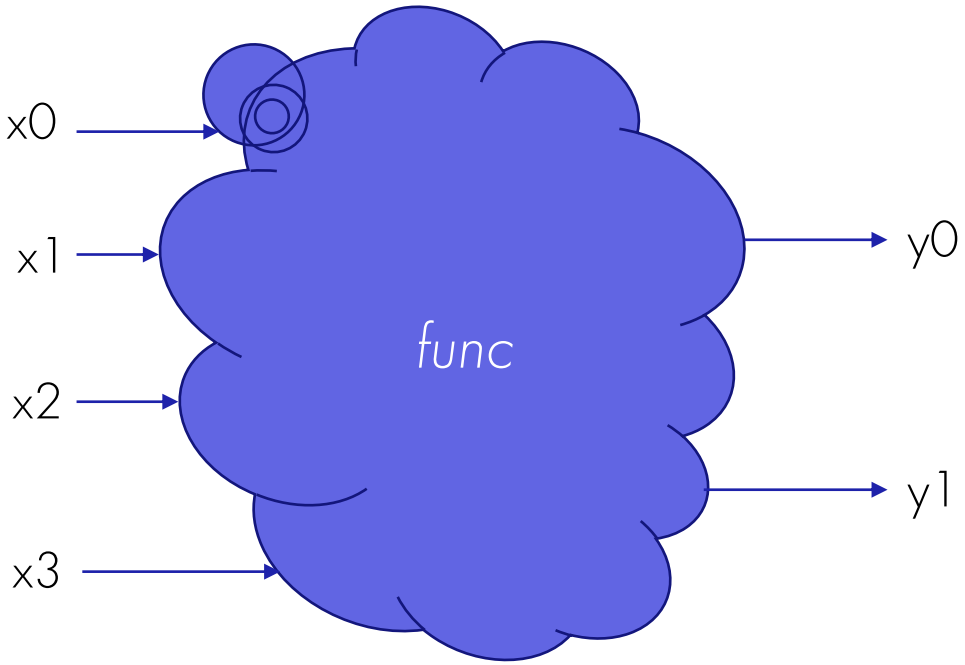
Основы Chisel

Функциональное программирование

Повторное использование

Итоги

Функциональное программирование



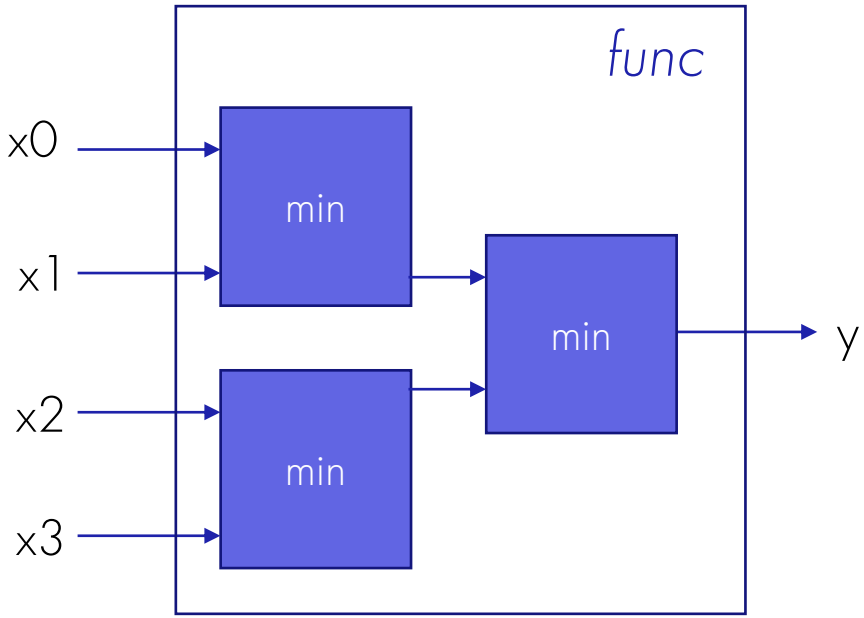
Парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних.

Wikipedia (c)

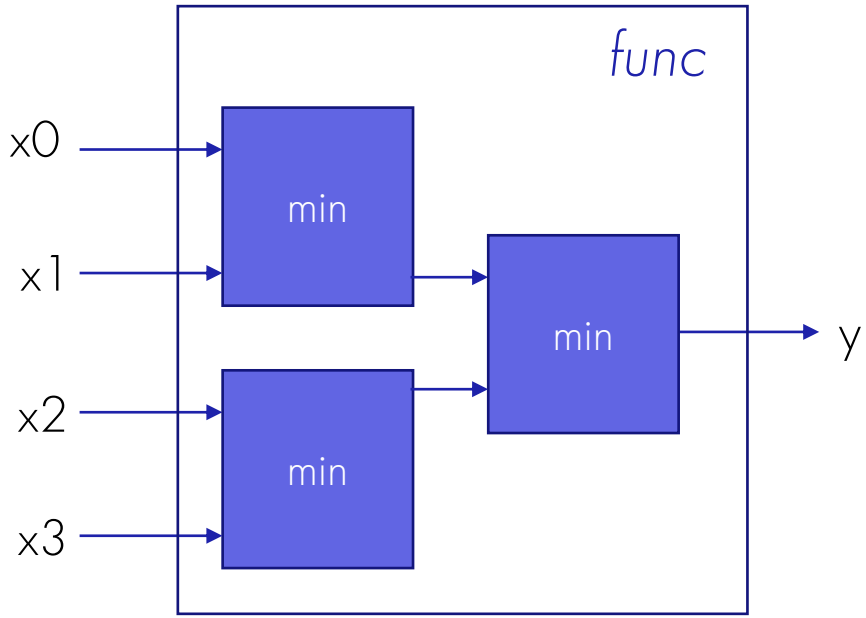
Scala

```
val (y0, y1) = func(x0, x1, x2, x3)
```

Функциональное программирование

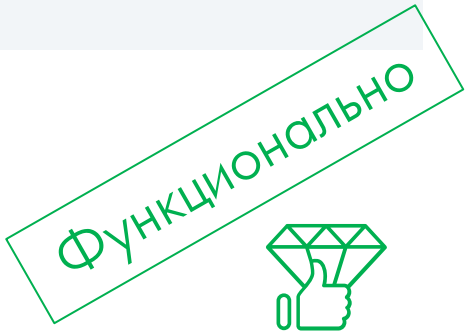


Функциональное программирование

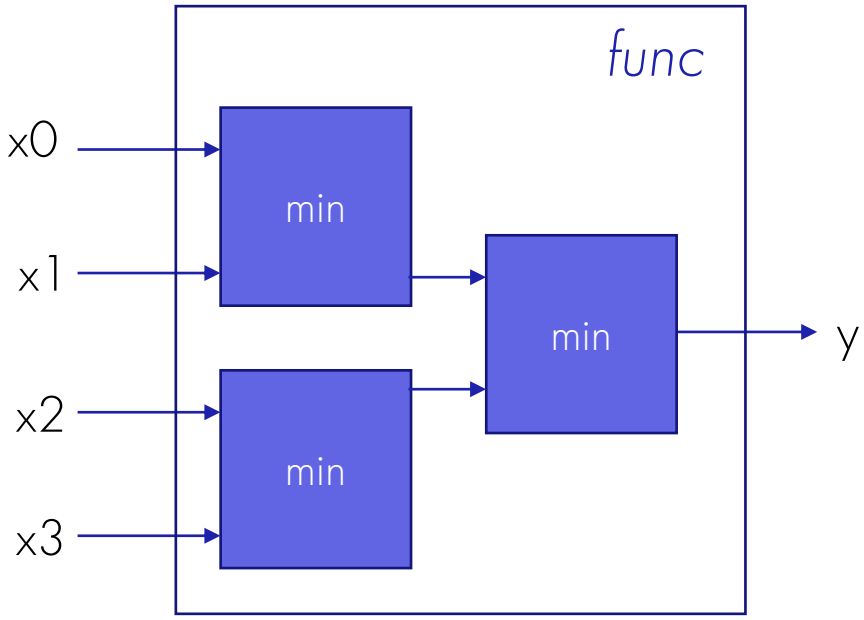


System Verilog

```
assign y0 = (x0 < x1) ? x0 : x1;
assign y1 = (x2 < x3) ? x2 : x3;
assign y  = (y0 < y1) ? y0 : y1;
```



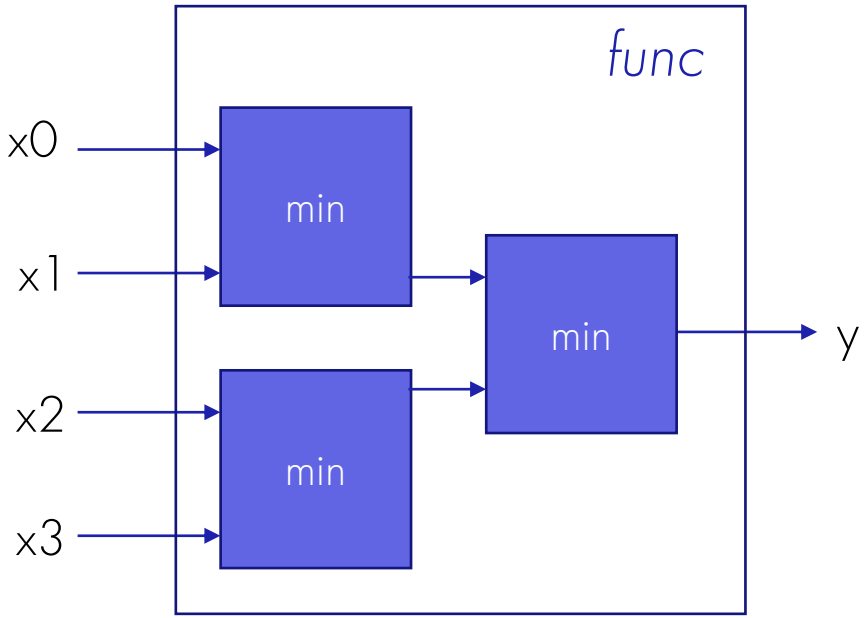
Функциональное программирование



System Verilog

```
1:  always_comb begin
2:    logic [WIDTH-1:0] tmp;
3:    int i;
4:    tmp = 2 ** WIDTH-1;
5:    for (i = 0; i < LEN; i = i + 1) begin
6:      if (tmp > x[i]) begin
7:        tmp = x[i];
8:      end
9:    end
10:   y = tmp;
11: end
```

Функциональное программирование



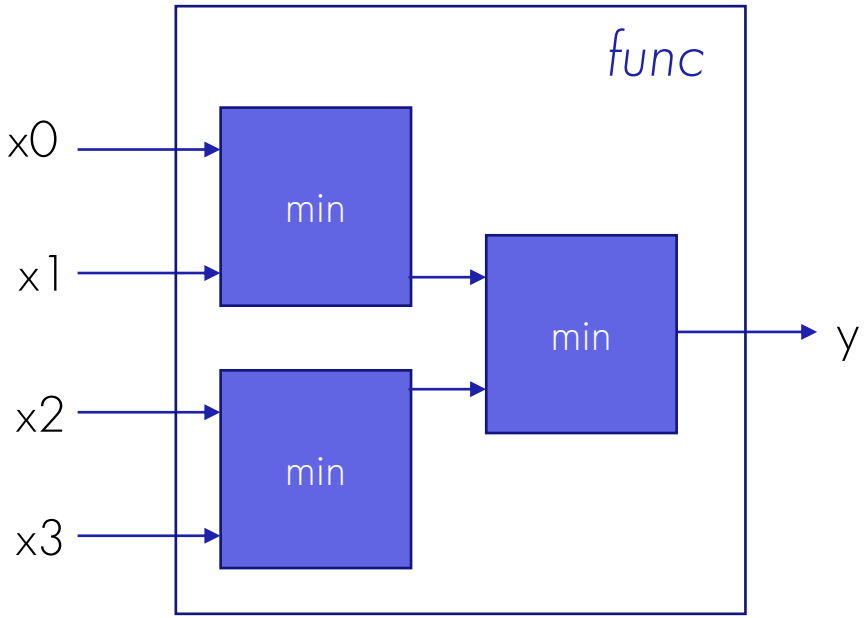
System Verilog

```
always_comb begin
  logic [WIDTH-1:0] tmp;
  int i;
  tmp = 2 ** WIDTH-1;
  for (i = 0; i < LEN; i = i + 1) begin
    if (tmp > x[i]) begin
      tmp = x[i];
    end
  end
  y = tmp;
end
```

Chisel

```
val y = x.reduce((n, k) => Mux(n < k, n, k))
```

Функциональное программирование



System Verilog

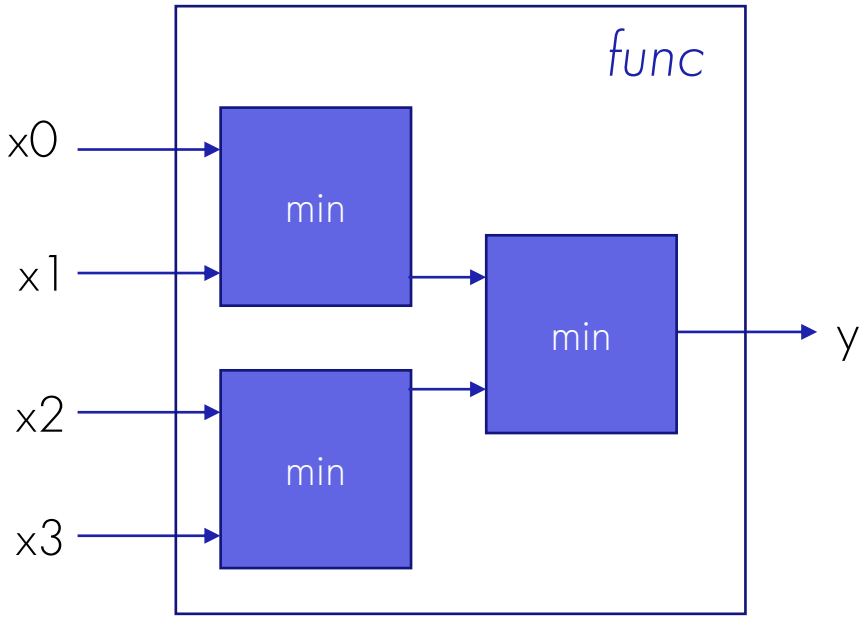
```
always_comb begin
  logic [WIDTH-1:0] tmp;
  int i;
  tmp = 2 ** WIDTH-1;
  for (i = 0; i < LEN; i = i + 1) begin
    if (tmp > x[i]) begin
      tmp = x[i];
    end
  end
  y = tmp;
end
```

Это не дерево!

Chisel

```
val y = x.reduce((n, k) => Mux(n < k, n, k))
```

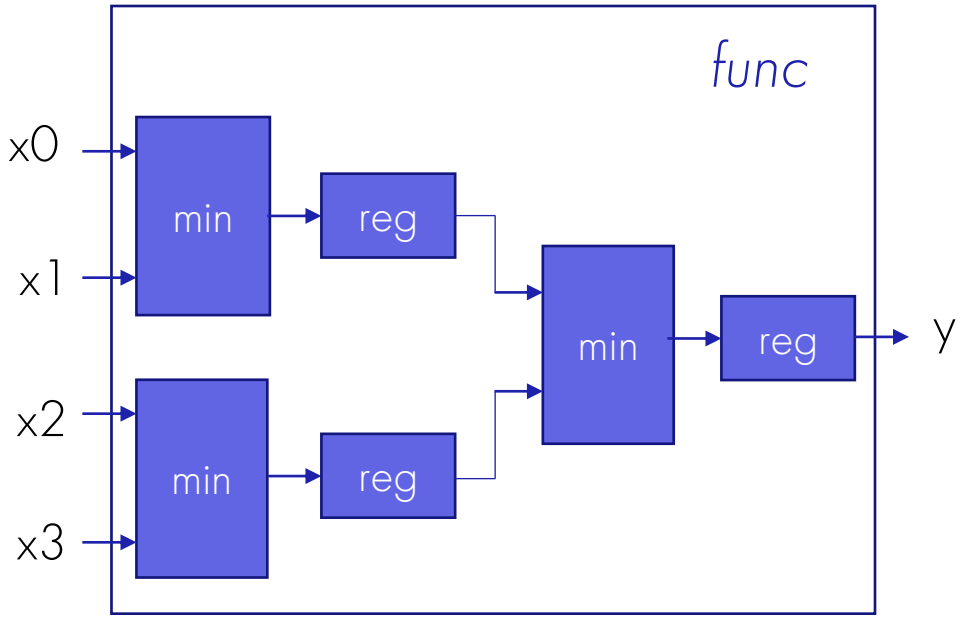
Функциональное программирование



Chisel

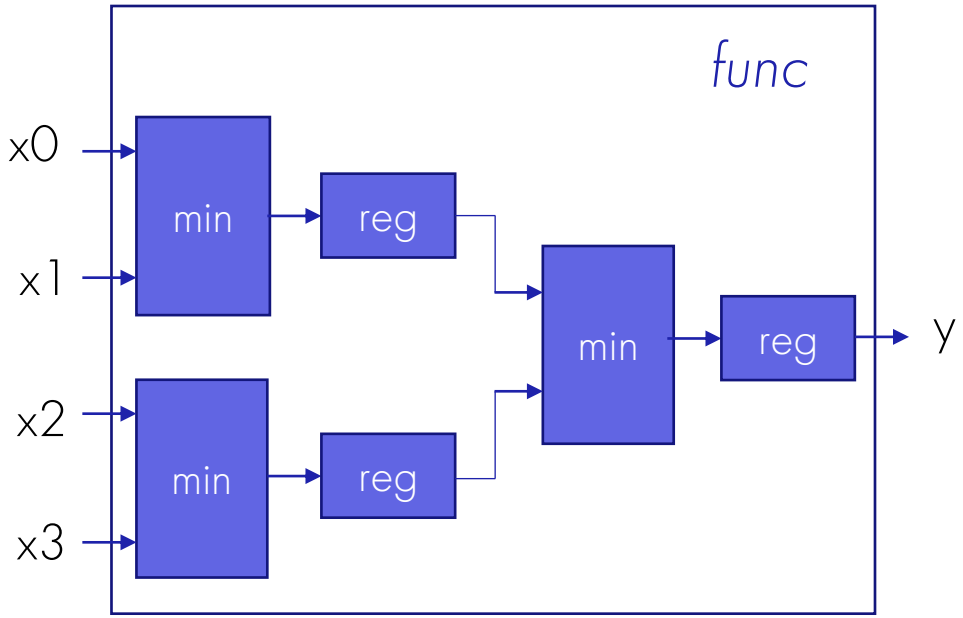
```
val y = x.reduceTree((n, k) =>
  Mux(n < k, n, k)
)
```

Функциональное программирование



?

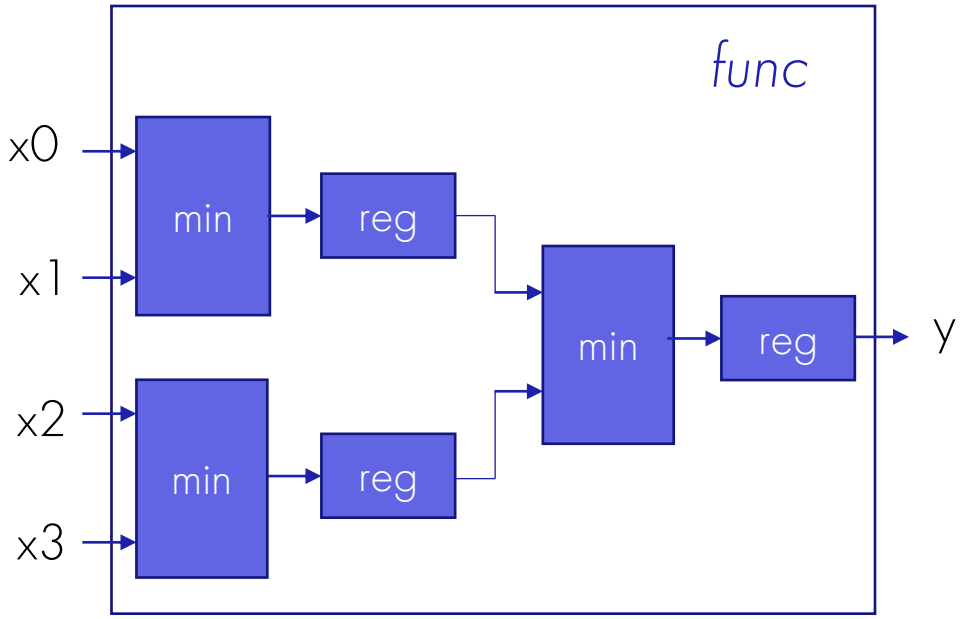
Функциональное программирование



Chisel

```
1:  val y = x.reduceTree((x, y) => {
2:    val reg = RegNext(Mux(x < y, x, y))
3:    reg
4:  })
```

Функциональное программирование



Chisel

```
1:  val y = x.reduceTree((x, y) => {
2:    val reg = RegNext(Mux(x < y, x, y))
3:    reg
4:  })
```

Сложно

Результаты генерации

(1)

```
module my_min(
  input          clock,
                 reset,
  input  [15:0] io_x_0,
                 io_x_1,
                 io_x_2,
                 ...
  output [15:0] io_y
);
```

(2)

```
  reg [15:0] io_y_reg;
  reg [15:0] io_y_reg_1;
  reg [15:0] io_y_reg_2;
  reg [15:0] io_y_reg_3;
  reg [15:0] io_y_reg_4;
  reg [15:0] io_y_reg_5;
  reg [15:0] io_y_reg_6;
```

(3)

```
  always @(posedge clock) begin
    io_y_reg <= io_x_0 < io_x_1 ? io_x_0 : io_x_1;
    io_y_reg_1 <= io_x_2 < io_x_3 ? io_x_2 : io_x_3;
    io_y_reg_2 <= io_x_4 < io_x_5 ? io_x_4 : io_x_5;
    io_y_reg_3 <= io_x_6 < io_x_7 ? io_x_6 : io_x_7;
    io_y_reg_4 <= io_y_reg < io_y_reg_1 ? io_y_reg : io_y_reg_1;
    io_y_reg_5 <= io_y_reg_2 < io_y_reg_3 ? io_y_reg_2 : io_y_reg_3;
    io_y_reg_6 <= io_y_reg_4 < io_y_reg_5 ? io_y_reg_4 : io_y_reg_5;
  end
  assign io_y = io_y_reg_6;
endmodule
```

Введение

Основы Chisel

Функциональное программирование

Повторное использование

Итоги

Повторное использование

System Verilog:

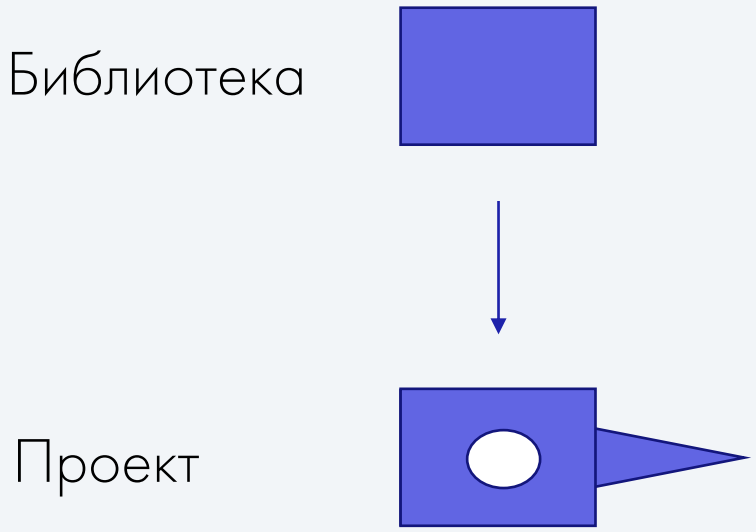
- параметры модулей
- глобальные макросы

Проблема:

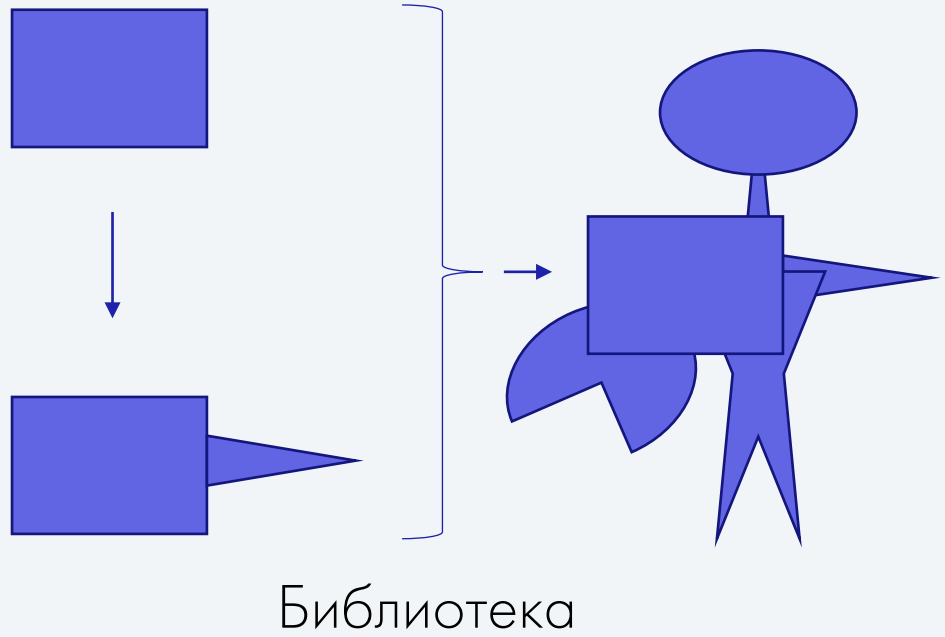
- нужен такой же модуль, но другой

Повторное использование

Дублирование:



Изменения в общем коде:

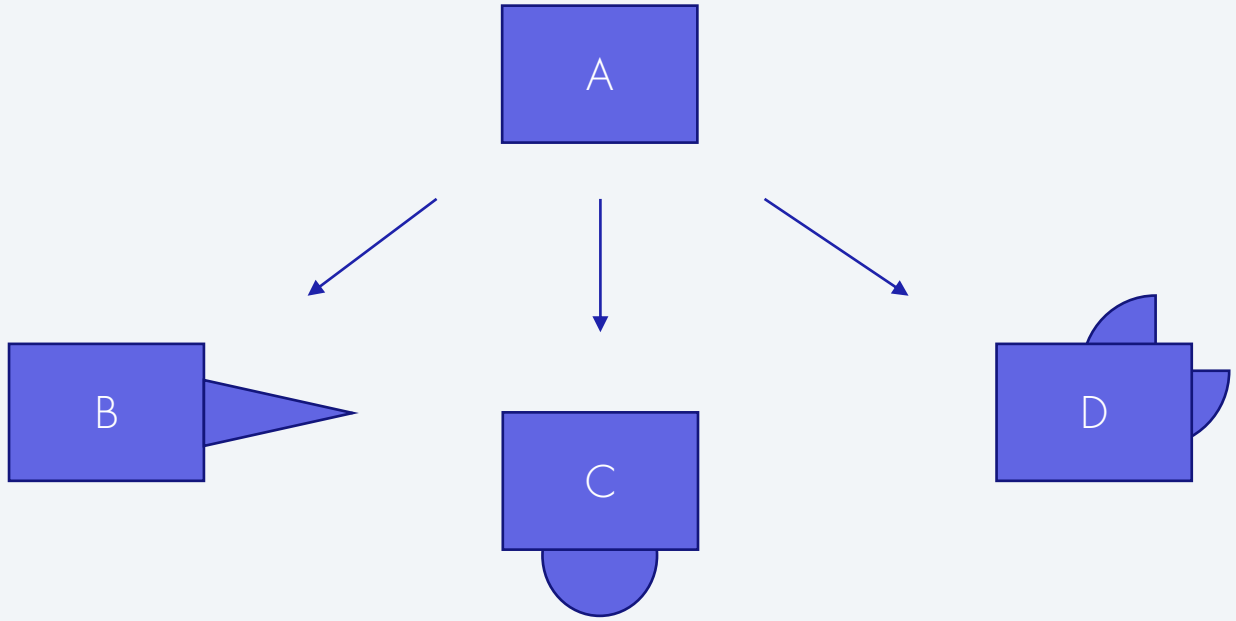


Повторное использование

Наследование:

Библиотека

Проекты



Повторное использование

System Verilog:

Как сделать модуль для решения всех задач?

Chisel:

Как сделать простой и легко расширяемый модуль?

Введение

Основы Chisel

Функциональное программирование

Повторное использование

Итоги

Выводы

Плюсы:

- простое описание сложных вещей
- код менее подвержен случайным ошибкам
- простой рефакторинг
- высокая портируемость кода

Минусы:

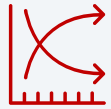
- тяжело и непонятно
- размер артефактов генерации

Итоги

Полезно

- если IP в целом очень сложен
- если используются самописные генераторы

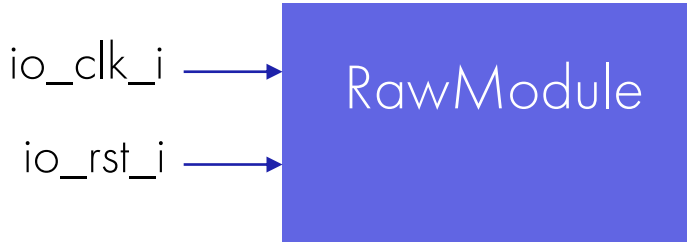
Бесполезно



- простые IP
- отсутствие желания к изучению

Спасибо за внимание!

Еще clock



Chisel

```
1: class my_module_cdc extends RawModule
2:       with RequireAsyncReset {

3:     val io = IO(new Bundle {
4:     val clk_i = Input(Clock())
5:     val rst_i = Input(AsyncReset())
     ...
   })
   ...
}
```

Chisel

```
1: withClockAndReset(io.clk_i, io.rst_i)
2: {

3:   val u_sync = Module(new cdc_ndff)
4:     u_sync.io.in  <> io.data_i
5:     u_sync.io.out <> io.data_o

6: }
```

Оставшееся позади

- библиотека Chisel (Queue, DecoupledIO, ValidIO и т. п.)
- богатые библиотеки Scala (Map, map, filter, zip и т. п.)
- строгая типизация
- инфраструктура для unit tests
- автоматизация процесса разработки