

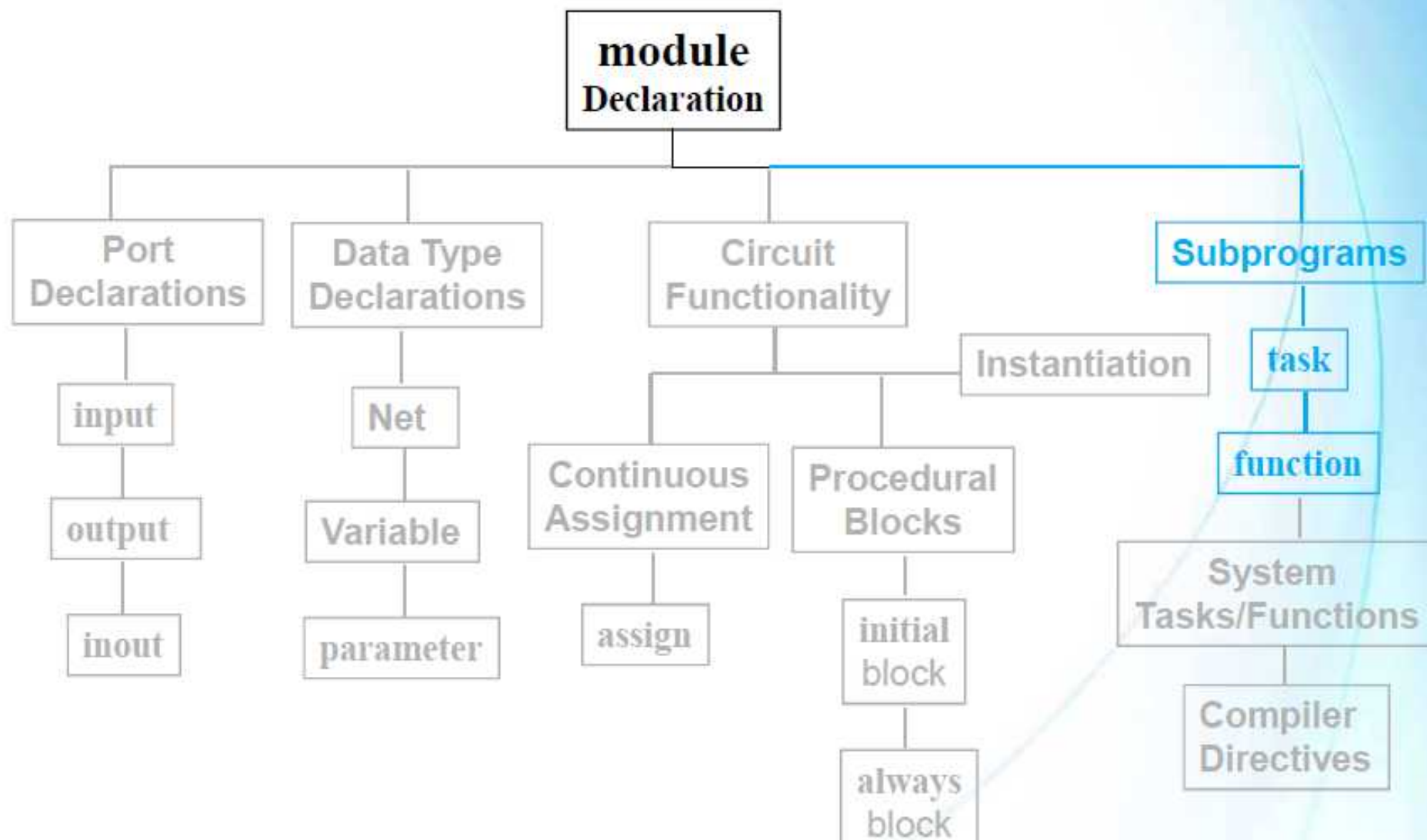


FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA
ENSINANDO E APRENDENDO

T566 –SISTEMAS DIGITAIS AVANÇADOS

Aula 13- Verilog

Prof. Danilo Reis





Verilog Functions e Tasks

Function e **Tasks** são subprogramas, constituídos de conjunto de instruções de comportamento. Usadas para:

- Substituir código repetitivo
- Melhorar a legibilidade do código

Function

- Retorna um valor baseado nas entradas;
- Produz lógica combinacional;
- Usada em expressões.Ex:

assign mult_out = mult (ina, inb);

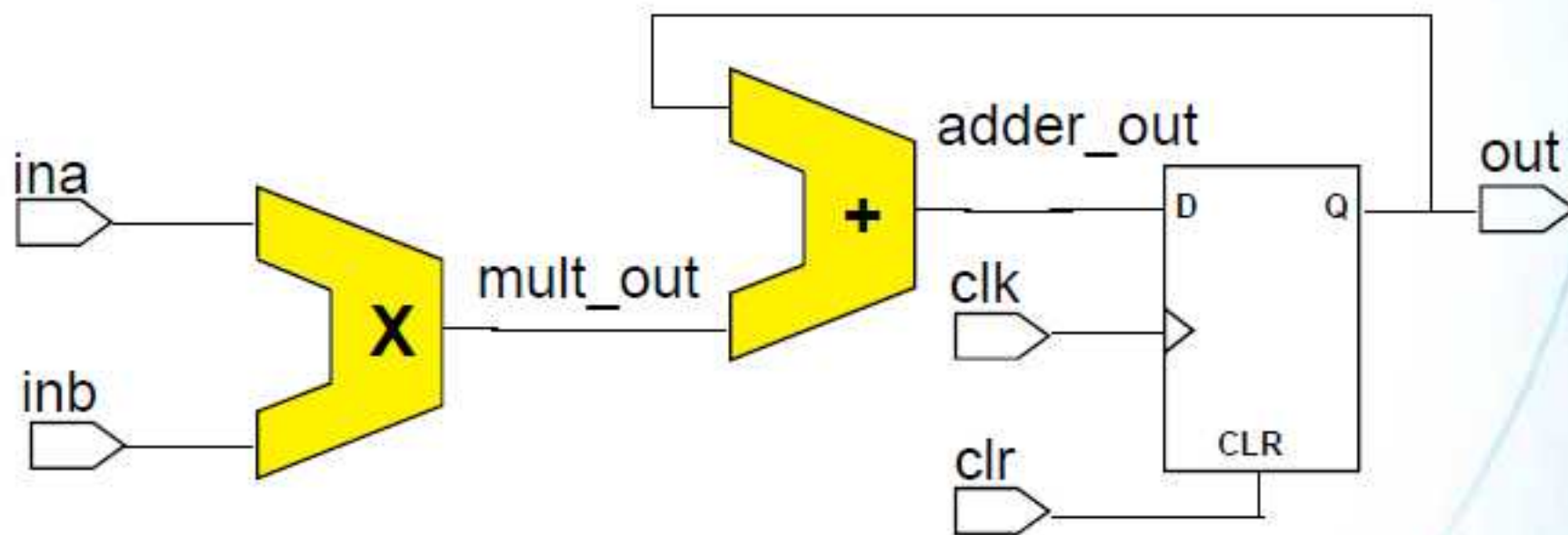
Tasks

- Similar as procedures em outras linguagens
- Pode ser combinacional ou registros
- Task são chamadas como instruções:

stm_out (nxt, first, sel, filter);



Criando um function para o MAC





Definição da Function

Function Definition - Multiplier

Function Definition:

```
function [15:0] mult;  
  input [7:0] a, b;  
  reg [15:0] r;  
  integer i;  
begin  
  if (a[0] == 1)  
    r = b;  
  else  
    r = 0;  
    for (i = 1; i <= 7; i = i + 1) begin  
      if (a[i] == 1)  
        r = r + b << i;  
    end  
    mult = r;  
end  
endfunction
```



Example Function Invocation

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter set = 10;
    parameter hld = 20;
```

```
    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

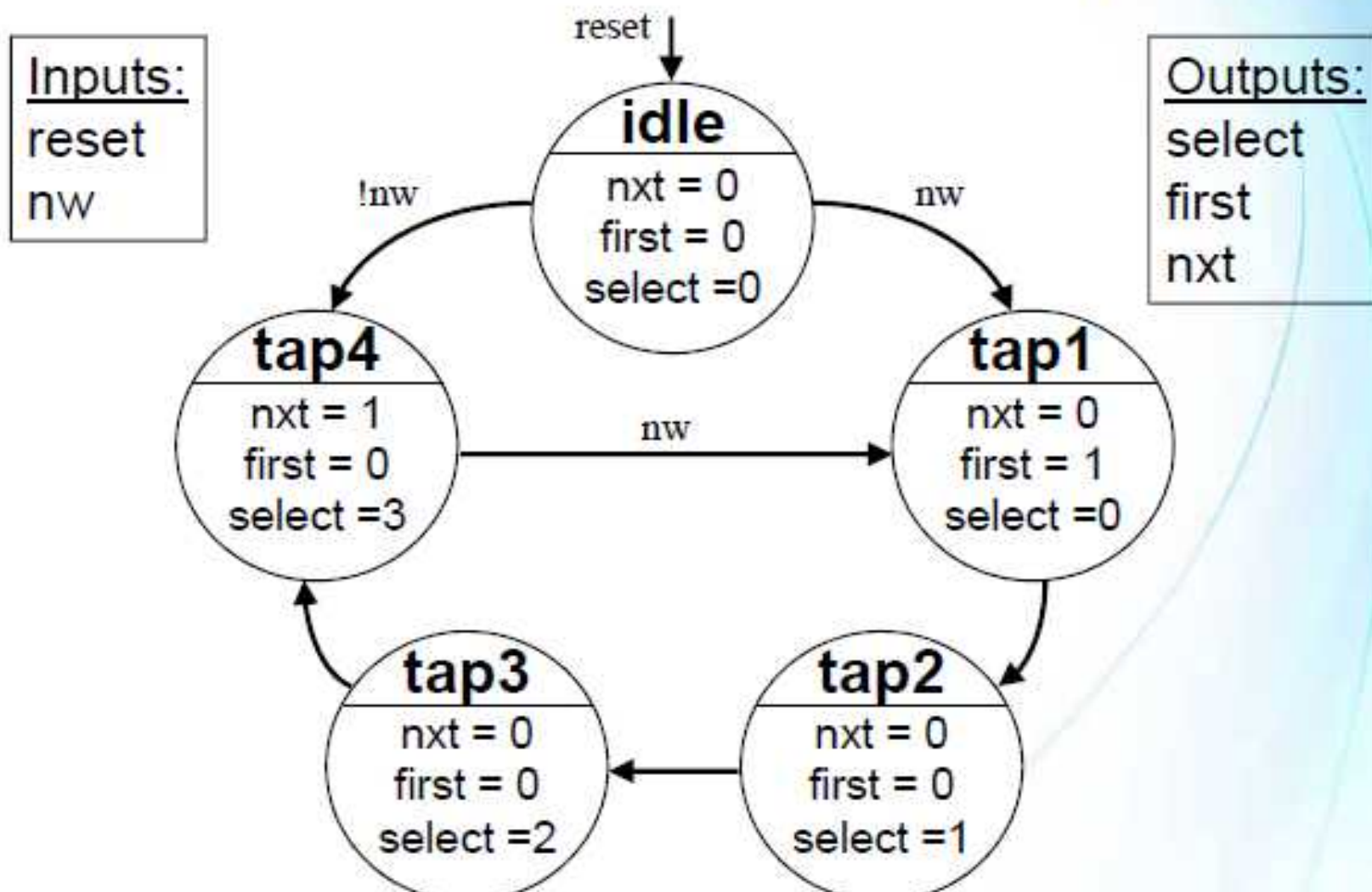
    assign mult_out = mult (dataa, datab)

    specify
        $setup (dataa, posedge clk, set);
        $hold (posedge clk, dataa, hld);
        $setup (datab, posedge clk, set);
        $hold (posedge clk, datab, hld);
    endspecify

endmodule
```



Create Task for State Machine Output





Task Definition – State Machine Output

```
task stm_out;  
  input [2:0] filter;  
  output reg nxt, first;  
  output reg [1:0] sel;  
  parameter idle=0, tap1=1, tap2=2, tap3=3, tap4=4;  
begin  
  nxt = 0;  
  first = 0;  
  case (filter)  
    tap1: begin sel = 0; first = 1; end  
    tap2: sel = 1;  
    tap3: sel = 2;  
    tap4: begin sel = 3; nxt = 1; end  
    default: begin nxt = 0; first = 0; sel = 0; end  
  endcase  
end  
endtask
```




Task Invocation – State Machine

```
module stm_fir (  
    input clk, reset, nw,  
    output reg nxt, first,  
    output reg [1:0] sel  
);  
  
    reg [2:0] filter;  
    parameter idle=0, tap1=1, tap2=2, tap3=3, tap4=4;  
    always @(posedge clk or posedge reset) begin  
        if (reset)  
            filter = idle;  
        else  
            case (filter)  
                idle: if (nw==1) filter = tap1;  
                tap1: filter = tap2;  
                tap2: filter = tap3;  
                tap3: filter = tap4;  
                tap4: if (nw==1) filter = tap1;  
                    else filter = idle;  
            endcase  
        end  
  
        always @(filter)  
            // Task Invocation  
            stm_out (nxt, first, sel, filter);  
    endmodule
```



Functions vs. Task

Function

- Sempre executam no time zero
 - Não podem ter execução pausada;
 - Podem não conter qualquer instrução de delay, evento ou controle de tempo;
- Deve ter pelo menos um argumento
 - Entradas podem não serem afetadas pela função;
- Argumentos não podem ser do tipo outputs e inouts;
- Sempre retorna um valor simples
- Podem chamar uma outra função, mas não pode chamar uma task.

Task

- Podem executar em time diferente de zero em tempo de simulação
 - Podem conter instrução de delay, evento ou controle de tempo;
- Podem ter nenhum ou várias argumentos aceitando também tipos output, e inout arguments;
- Modificam nenhum ou vários valores;
- Podem chamar functions ou outras tasks



Revisão - Modelo comportamental

Continuous Assignment

```
module full_adder4 (  
    output [3:0] fsum,  
    output fco,  
    input [3:0] a, b,  
    input cin  
);  
  
assign {fco, fsum} = cin + a + b;  
  
endmodule
```

Procedural Block

```
module fl_add4 (  
    output reg [3:0] fsum,  
    output reg fco,  
    input [3:0] a, b,  
    input cin  
);  
  
always @(cin or a or b)  
    {fco, fsum} = cin + a + b;  
  
endmodule
```



2 Tipo de processos RTL

Processo Combinacional

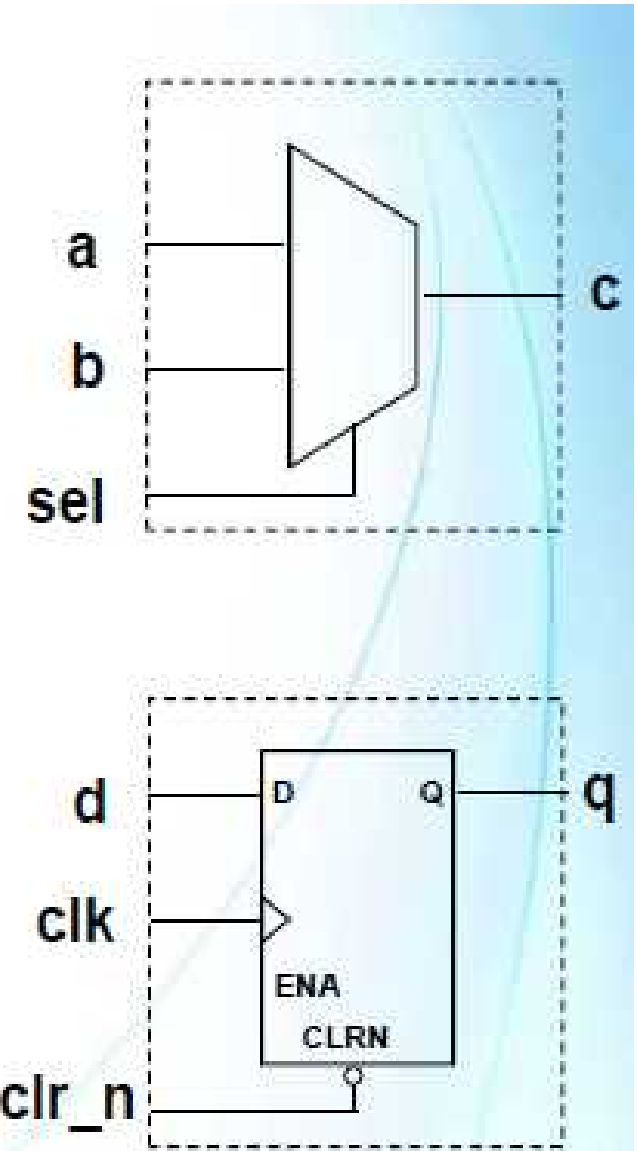
- Sensível a todas entradas usadas em lógica combinacional

the combinatorial logic

```
always @ (a, b, sel)  
always @ *
```

*Sensitivity list includes
all inputs used in the
combinatorial logic.*

** is a Verilog shortcut to manually having to add all inputs*



Processos Clocked

- Sensível a transição do clock e/ou sinais de controle

```
always @(posedge clk, negedge clr_n)
```



Funcional Latch vs. Flip-flop

Level-Sensitive Latch

```
module latch (  
    input d, gate,  
    output reg q  
);
```

```
    always @(d, gate)  
        if (gate)  
            q = d ;
```

```
endmodule
```

Edge-Triggered Flipflop

```
module dff (  
    input d, clk,  
    output reg q  
);
```

```
    always @(posedge clk)  
        q <= d ;
```

```
endmodule
```



Síncrono vs. Assíncrono

Synchronous Preset & Clear

```
module dff_sync (  
    input d, clk, sclr, spre,  
    output reg q  
);  
  
    always @(posedge clk) begin  
        if (sclr)  
            q <= 1'b0;  
        else if (spre)  
            q <= 1'b1;  
        else  
            q <= d;  
        end  
  
endmodule
```

Asynchronous Clear

```
module dff_async (  
    input d, clk, aclr,  
    output reg q  
);  
  
    always @(posedge clk,  
        posedge aclr) begin  
        if (aclr)  
            q <= 1'b0;  
        else  
            q <= d;  
        end  
  
endmodule
```



Clock Enable

Clock Enable

```
module dff_ena (  
    input d, enable, clk;  
    output reg q  
);  
  
/* If clock enable port does not exist in  
target technology, then a mux in  
front of the d input is generated */  
  
    always @( posedge clk )  
        if (enable)  
            q <= d;  
  
endmodule
```




Functional Counter

```
module cntr (  
    input aclr, clk,  
    input [7:0] d,  
    input [1:0] func, // Controls functionality  
    output reg [7:0] q,  
);  
  
    always @ (posedge clk, posedge aclr) begin  
        if (aclr)  
            q <= 8'h00;  
        else  
            case (func)  
                2'b00: q <= d; // Loads counter  
                2'b01: q <= q + 1; // Counts up  
                2'b10: q <= q - 1; // Counts down  
            endcase  
        end  
    end  
endmodule
```



Princípio Básico Blocking/Nonblocking

- Use operador blocking (=) para lógica combinacional;
- Use operador nonblocking(<=) para lógica sequencial

Isto evita confusões e hardware desnecessários na implementação do RTL durante a síntese.



MAC (Behavioral Modeling)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
        end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab), .mult_out(mult_out));

endmodule
```



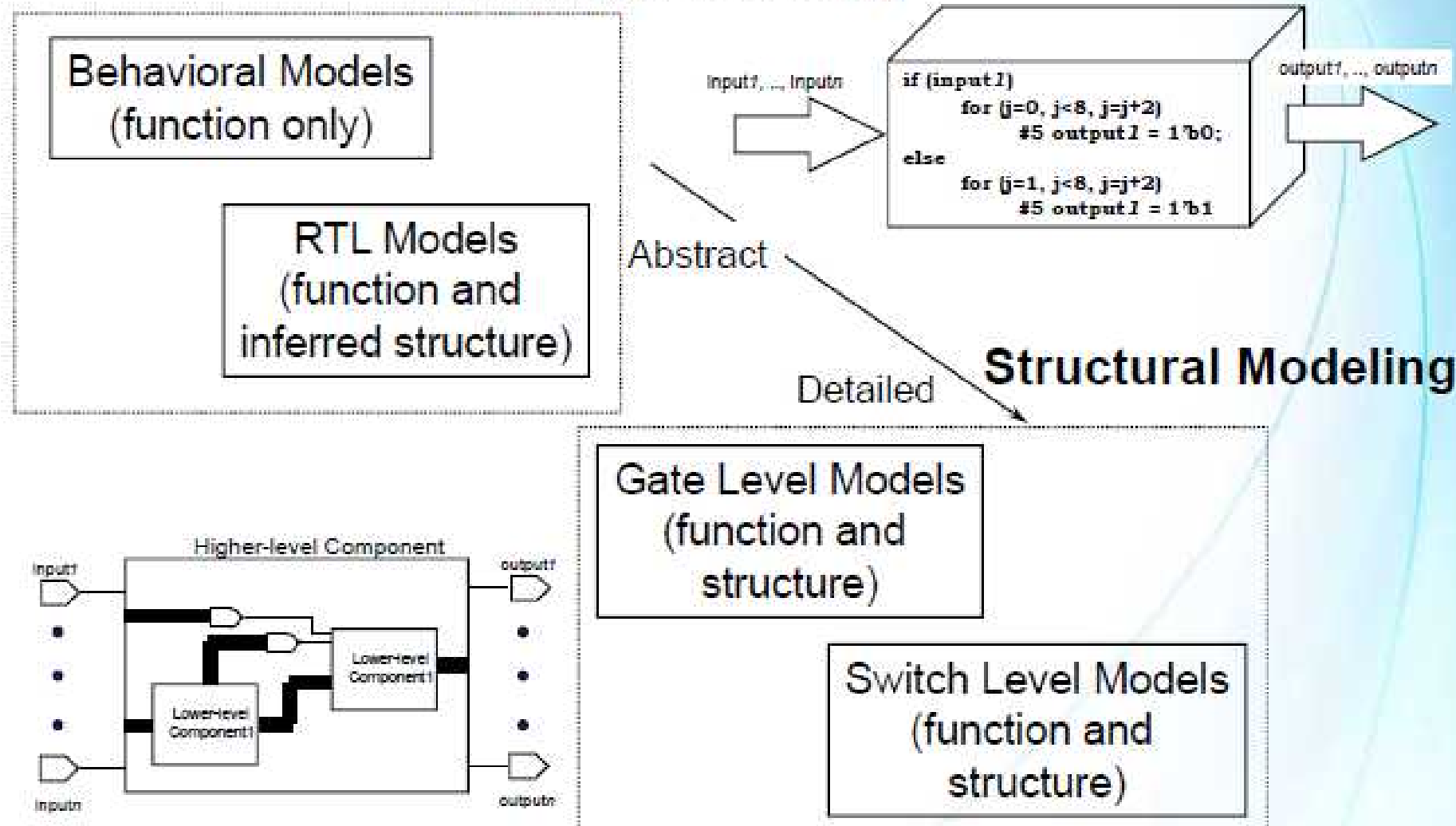
Modelo Estrutural

- Define funções e estruturas do circuito digital
- Adiciona Hierarquia



Levels of Abstraction

Behavioral Modeling





Verilog Modelo Estrutural

Modelo Gate-level - instanciação com portas primitivas nativas do Verilog.

- and, nand, or, nor, xor, xnor
- buf, bufif0, bufif1, not, notif0, notif1

Primitivas User-defined – instanciação de primitivas criadas pelo projetista

Instanciação de Módulos - instanciação de componentes baixo nível criados pelo usuário

designs (components)













Modelo Switch Level - instanciação de switch primitivas nativas do Verilog

- nmos, rnmos, pmos, rpmos, cmos, rcmos
- tran, rtran, tranif0, rtranif0, tranif1, rtrainif1, pullup, pulldown



Verilog Modelo Gate-Level

- Verilog has predefined gate primitives

Primitive	Name	Function	Primitive	Name	Function
	and	n-input AND gate		buf	n-output buffer
	nand	n-input NAND gate		not	n-output buffer
	or	n-input OR gate		bufif0	tristate buffer lo enable
	nor	n-input NOR gate		bufif1	tristate buffer hi enable
	xor	n-input XOR gate		notif0	tristate inverter lo enable
	xnor	n-input XNOR gate		notif1	tristate inverter hi enable



Formato de instanciação portas primitivas

```
<gate_name> #<delay> <instance_name> (port_list);
```

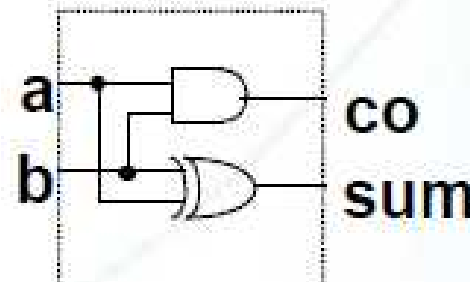
- **<gate_name>**
 - The name of gate (e.g. AND, NOR, BUFIF0...)
- **#delay**
 - Delay through gate
 - Optional
- **<instance_name>**
 - Unique name applied to individual gate instance
 - Optional
- **(port_list)**
 - List of signals to connect to gate primitive



Formato de instanciação

- Portas primitivas do Verilog a primeira porta na port list é a saída seguida das entradas
 - <gate_name>
 - and
 - xor
 - #delay (optional)
 - 2 unidades de time unit para a porta and
 - 4 unidades de time para porta xor
 - - <instance_name> (optional)
 - u1 para porta and
 - u2 para porta xor
 - - (port_list)
 - (co, a, b) - (output, input, input)
 - (sum, a, b) - (output, input, input)

```
module half_adder (  
    output co, sum,  
    input a, b  
);  
  
    parameter and_delay = 2;  
    parameter xor_delay = 4;  
  
    and #and_delay u1(co, a, b);  
    xor #xor_delay u2(sum, a, b);  
  
endmodule
```





Formato de instanciação módulos

```
<component_name> #<delay> <instance_name> (port_list);
```

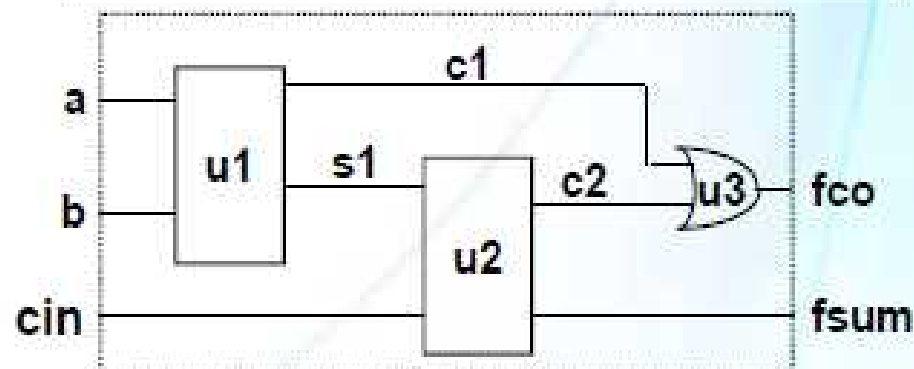
- **<component_name>**
 - The module name of your lower-level component
- **#<delay>**
 - Delay through component
 - Optional
- **<instance_name>**
 - Unique name applied to individual component instance
- **(port_list)**
 - List of signals to connect to component



Formato de instanciação

- Dois métodos de definir conexões das portas
 - Por lista ordenada
 - Por nomes
- Por lista ordenada(1st half adder*)
 - Conexões das portas definidas pela ordem da port list na declaração do módulo
 - `module half_adder (co, sum, a, b);`
 - Ordem das conexões das portas não importa
 - `co -> c1, sum -> s1, a -> a, b -> b`
- Por nome (2nd half_adder*)
 - Conexões das Portas são definidas por nome
 - Método recomendado
 - Ordem das conexões das porta não importa
 - `a -> s1, b -> cin, sum -> fsum, co ->c2`

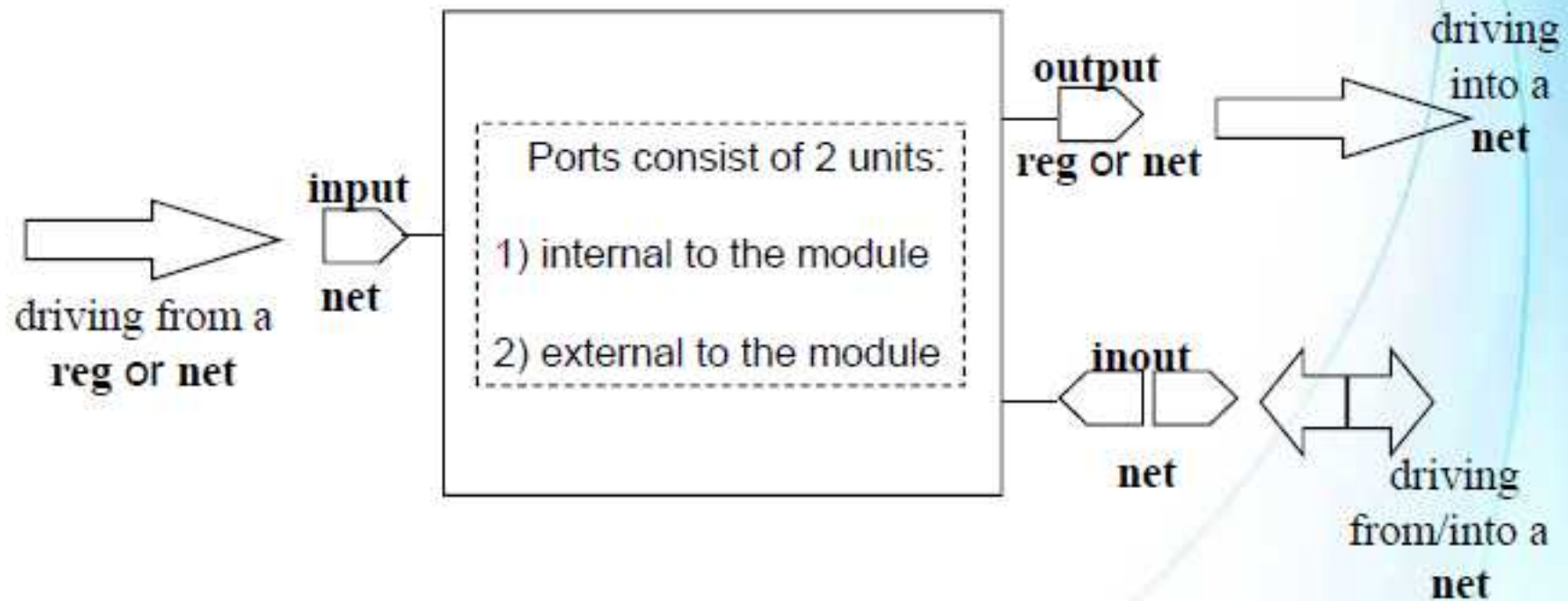
```
module full_adder (  
    output fco, fsum,  
    input cin, a, b  
);  
    wire c1, s1, c2;  
  
    half_adder u1 (c1, s1, a, b);  
    half_adder u2 (.a(s1), .b(cin),  
        .sum(fsum), .co(c2));  
    or u3(fco, c1, c2);  
  
endmodule
```





Regras de Conexões de Portas

These are the requirements for port connections when modules are instantiated within other modules





Sobrescrevendo Parameters

- Se um módulo lower-level contém parâmetros, existem duas formas de sobrescrever-los durante a instanciação
 - Feito no tempo de compilação, O parâmetro é resolvido como uma constante
- defparams
- Atribuição do parameter na instanciação



Defparam

- Use instrução **defparam** e nome hierárquico para sobrescrever parameters na instanciação do módulo;
- Pode ser colocado fora do módulo verilog, isto é, em outro arquivo;

```
module full_adder (  
    output fco, fsum,  
    input cin, a, b  
);  
  
    wire c1, s1, c2;  
  
    defparam u1.and_delay = 4, u1.xor_delay = 6;  
    defparam u2.and_delay = 3, u2.xor_delay = 5;  
  
    half_adder u1 (c1, s1, a, b);  
    half_adder u2 (.a(s1), .b(cin),  
        .sum(fsum), .co(fco));  
    or u3(fco, c1, c2);  
  
endmodule
```




Atribuição durante instanciação

- Definição do Parameter ocorre durante a instanciação do módulo
- Introduzido no Verilog '2001
- Método recomendado por facilitar a legibilidade

```
module full_adder (  
    output fco, fsum,  
    input cin, a, b  
);  
  
    wire c1, s1, c2;  
  
    half_adder #(4, 6)  
        u1 (c1, s1, a, b);  
    half_adder #(.and_delay(3), .xor_delay(5))  
        u2 (.a(s1), .b(cin), .sum(fsum), .co(fco));  
    or u3(fco, c1, c2);  
  
endmodule
```

Ordered list method

Name method (recommended)

To just use the (default) value defined in the lower-level module, in the name method, use empty value (e.g. `.and_delay()`) or leave parameter assignment out entirely



Atribuição durante instanciacao MAC (Module Instantiation)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
        end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab), .mult_out(mult_out));

endmodule
```



Atribuição durante instanciação

MAC (Module Instantiation & Local Parameter)

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    localparam mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
        end

    multa #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab), .mult_out(mult_out));

endmodule
```

Name **mult_size** defined
as local parameter so it
cannot accidentally be
overwritten at compile time



Diretivas de compilação & System Tasks

- Diretivas de compilação
 - São comandos direcionados para controlar o comportamento do compilador
 - Indicados pelo caracter ` (aspa simples)
 - Tecla a esquerda da tecla 1 no teclado, não é apostrofe!
 - Algumas vezes pode ser colocado dentro do módulo Verilog, outras vezes deve ser colocado fora do módulo
- System tasks & functions
 - Built-in Verilog tasks e functions
 - Começam com o caracter \$
 - Prover uma variedade de funcionalidades uteis
- Cobriremos poucas diretivas de compilação e task e functions do sistema para dar uma idéia das funcionalidades
 - Você pode encontrar muitas outras na web



Exemplos de diretivas de compilação

- ``timescale`
- ``include`
- ``define / `undef`
- ``ifdef / `ifndef / `elsif / `else / `endif`



Compiler Directives 'timescale

Defines module timing using two values

- <reference_time_unit>: specifies the unit of measurement for

times and delays

- <time_precision>: specifies the precision to which the delays are

rounded off during simulation

Only 1, 10, and 100 are supported integers

Must be placed outside of module definition

Ex: ``timescale 1 ns / 10 ps`



Compiler Directives `define and `undef

Define and undefine macros that perform text substitution

Use macro by typing
macro_name

Anything can be substituted

- Numbers
- Characters and strings
- Comments
- Keywords
- Operators

Can be placed outside or inside of module definition

Can pass arguments

Have global visibility

- Once defined, can be used within any Verilog file read afterwards

```
`define HADD_DELAY_VAL1 #(4,6)
`define HADD_DELAY_VAL2 #(3,5)
`define MY_OR(or_out, or_ina, or_inb) \
    or #5 (or_out, or_ina, or_inb)

module full_adder (
    output fco, fsum,
    input cin, a, b
);
    wire c1, s1, c2;

    half_adder `HADD_DELAY_VAL1
        u1 (c1, s1, a, b);
    half_adder `HADD_DELAY_VAL2
        u2 (.a(s1), .b(cin), .sum(fsum), .co(c2));
    `MY_OR (fco, c1, c2);

endmodule

`undef MY_OR
```




Compiler Directives `ifdef / `ifndef / `elsif / `else / `endif

Provide support for conditional compilation

- Ex. Support different variations of a module
- Ex. Choose different sets of stimulus

Allow designer to compile Verilog statements based on whether macros have been defined

- Any valid Verilog statements can be placed within

Definitions

- `ifdef <macro_name> : code compiled if macro defined
- `ifndef <macro_name> : code compiled if macro not defined
- `elsif <macro_name> : code compiled if macro defined and previous `ifdef/`ifndef not satisfied
- `else : code compiled if previous `ifdef/`ifndef not satisfied
- `endif : end code compilation region (one per `ifdef/`ifndef)

Can be placed outside or inside of module definition

Can be nested



Exemplo de compilação condicional

```
// Conditional Compilation

`ifdef TEST // Compile counter_test only if macro TEST has been defined
  module counter_test;
  ...
endmodule

`else // Compile the module counter as default
  module counter;
  ...
endmodule

`endif
```



System Tasks and Functions

- Simulation control & time
- Display control
- Math functions
 - Not discussed
- Conversion
 - Not discussed
- File I/O
 - Not discussed



Simulação e Control & Time

- \$stop task
 - - Pauses simulation
 - - Simulator still running
 - - Add argument 1 or 2 to print message about simulator state
- \$finish task
 - - Stop simulation
 - - Shut down simulator
 - - Add argument 1 or 2 to print message about simulator state
- \$time function
 - - Returns current time in simulation



Controle de Display

All display controls ignored for synthesis

`$display(...)` task

- Writes formatted message to simulator display whenever task is called
- Example

```
$display("%0d : \n Calculated %0d, Expected %0d", $time, sum, calc_cum);
```

`$monitor(..., ...)`

- Writes formatted message to simulator display whenever any monitor inputs change in value (except `$time`)

```
$monitor("%0d : \n Calculated %0d, Expected %0d", $time, sum, calc_cum);
```

monitor inputs





Referências

- Curso oficial da Altera "Quartus II Design Foundations"
- Aula do curso "ECE 448 - FPGA and ASIC" Design with HDL" George Manson University;
- <http://www.fpgadeveloper.com/2011/07/list-and-comparison-of-fpga-companies.html>