



FUNDAÇÃO EDSON QUEIROZ  
UNIVERSIDADE DE FORTALEZA  
ENSINANDO E APRENDENDO

## T566 –SISTEMAS DIGITAIS AVANÇADOS

---

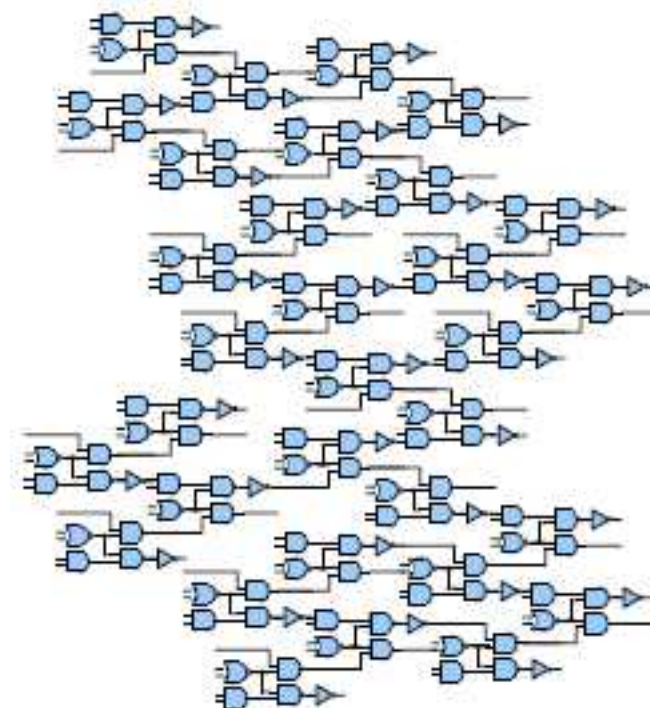
# Aula 8 - Linguagens de Descrição de Hardware

Prof. Danilo Reis



## Por que HDL?

- Projetos com esquemáticos
  - Ferramentas não são compatíveis entre si;
  - Difícil de compartilhar
  - Cada ferramenta tem processos diferentes logo exige aprendizado
  - Consomem muito tempo para projetos com milhares de portas
- São escalável para grandes projetos  
100.000 > gates?





## Solução

- Descrever o projeto em uma linguagem de texto (Hardware Description Language)
- Descrever apenas o comportamento não o detalhamento a nível de portas (gate level)
- Descrição a nível ser gerada automaticamente por ferramentas de sínteses

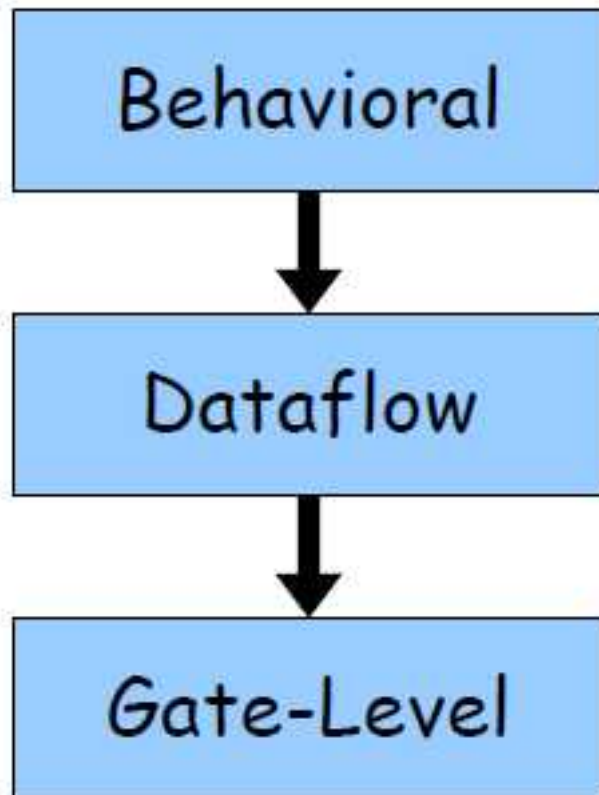


## Vantagens de HDLs

- Projetistas descrevem o comportamento do hardware sem se preocupar com projeto do mesmo;
- Codificação HDL permitem aos projetistas separar entre comportamento (behavior) e implementação (synthesis);
- Os projetistas desenvolvem uma especificação funcional e executável de todos os componentes e suas interfaces;
- Projetistas podem decidir sobre custo, performance, potência e área nas fases iniciais do projeto;
- Algumas ferramentas podem manipular o projeto para gerar automaticamente fazer verificação, síntese e otimização do projeto



## Níveis de Abstrações



- Define funcionalidade como uma caixa preta. Não sendo importante como será implementado;
- Implementa o módulo definindo como será o fluxo de dados dentro dos registradores;
- O módulo é implementado em lógica concreta



## HDL não é linguagem de Programação

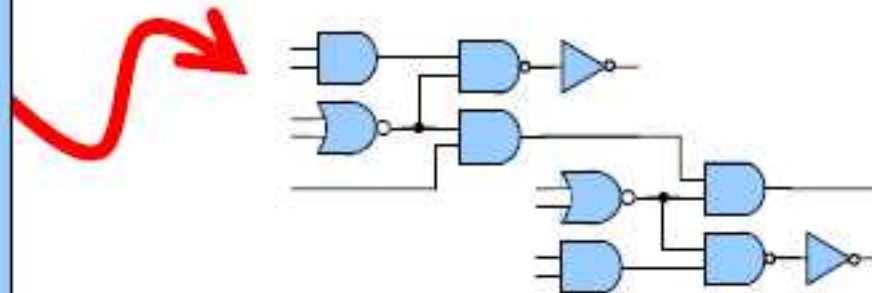
### Linguagens de Programação

Podem ser traduzidas em instruções de máquinas que podem ser executadas em um computador.

### Linguagens de Descrição de Hardware (HDL)

São linguagens com sintaxe e semântica com suporte para modelar temporalmente, comportamentalmente e estruturalmente estruturas de hardware.

```
module foo(clk,xi,yi,done):  
  input [15:0] xi,yi;  
  output done;  
  
  always @(posedge clk)  
  begin:  
    if (!done) begin  
      if (x == y) cd <= x;  
      else (x > y) x <= x - y;  
    end  
  end  
endmodule
```







## Exemplo Gate-Level

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    wire [1:0] sel_b;
    not not0( sel_b[0], sel[0] );
    not not1( sel_b[1], sel[1] );

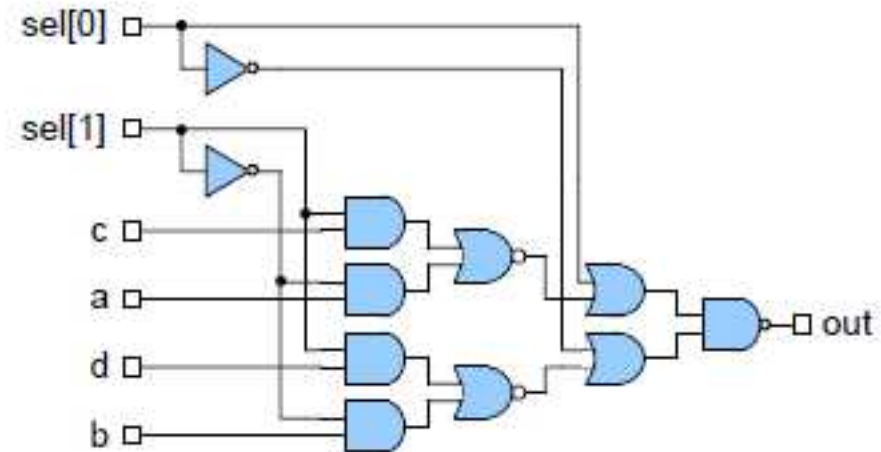
    wire n0, n1, n2, n3;
    and and0( n0, c, sel[1] );
    and and1( n1, a, sel_b[1] );
    and and2( n2, d, sel[1] );
    and and3( n3, b, sel_b[1] );

    wire x0, x1;
    nor nor0( x0, n0, n1 );
    nor nor1( x1, n2, n3 );

    wire y0, y1;
    or or0( y0, x0, sel[0] );
    or or1( y1, x1, sel_b[0] );

    nand nand0( out, y0, y1 );

endmodule
```



Gate-Level é elaborado com primitivas básicas não sendo necessário definir módulos




## Exemplo DataFlow

```
module mux4( input  a, b, c, d
              input [1:0] sel,
              output out );

  wire out, t0, t1;
  assign t0 = ~( (sel[1] & c) | (~sel[1] & a) );
  assign t1 = ~( (sel[1] & d) | (~sel[1] & b) );
  assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```



**Chama-se** continuous  
assignments





## Exemplo DataFlow

```
module mux4( input  a, b, c, d
              input [1:0] sel,
              output out );

  wire t0 = ~( (sel[1] & c) | (~sel[1] & a) );
  wire t1 = ~( (sel[1] & d) | (~sel[1] & b) );
  wire out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```

Forma mais suscinta



## Exemplo Behavior

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    reg out;

    always @( a or b or c or d or sel )
    begin
        if ( sel == 0 )
            out = a;
        else if ( sel == 1 )
            out = b;
        else if ( sel == 2 )
            out = c;
        else if ( sel == 3 )
            out = d;
    end

endmodule
```

Um bloco always é um behavioral block que contém uma lista de expressões que normalmente são executadas sequencialmente. Este código é sempre bem abstrato



## Pontos Importantes da em codificação com HDL

- Nem toda construção HDL é sintetizável;
- Síntese depende da ferramenta;
- É bom utilizar poucos comandos(case, if else, atribuições concorrentes e sequenciais);
- Procure manter na cabeça o circuito que se deseja sintetizar;
- Se o estilo de programação for C like, normalmente aumenta a área no silício;
- Conversões e estímulos de testes não são sintetizáveis;
- Use e abuse de comentários (mais que em assembler);
- Explique o funcionamento operacional dos módulos;
- Descreva a arquitetura claramente;
- Código pequeno HDL não implica em área de silício pequena;
- Procure cobrir com tratamentos todos os possíveis estados de if else, case;
- Não use loops encadeados para descrição de circuitos;
-



## Linguagens HDL

- **Verilog**
- **VHDL** ([VHSIC Hardware Description Language](#))(VHSIC - [Very High Speed Integrated Circuits](#));
  - System C;
  - AHDL ( Analog Hardware Description Language)
  - ABEL (Advanced Boolean Expression Language);
  - Bluespec(high-level HDL originally based on [Haskell](#), now with a[SystemVerilog](#) syntax);
  - JHDL;(Baseada em Java)
  - SystemVerilog;



## Comparação Verilog x VHDL

VERILOG	VHDL
Criada em em 1985	Criada 1981 DoD
Sintaxe similar a C	Sintaxe similar a ADA
Tipos pré-definidos e representações Lógicas	Tipos extensíveis
Módulos tem única implementação	Entidade tem múltiplas arquiteturas
Modelagem a nível de portas (RTL) e a nível comportamental	Modelagem a nível de portas (RTL) e a nível comportamental
Fácil de aprender	Difícil de aprender



## Por que vamos adotar Verilog

### ● Vantagens

- Mais utilizada por projetistas nos USA, adotada em grandes universidades como MIT ;
- Mais fácil de aprender;
- Sintaxe similar a C;
- Boa para gerar síntese e verificação;
- Padrão IEEE;

### ● Desvantagens

- Fácil de criar códigos horríveis! Estilo na codificação é importante!
- Principiantes assumem que a similaridade com a sintaxe C signifique a semântica C;
- Principiantes sempre erram os ';'.



## Como transformar código HDL em um circuito?

### ● Problemas

- Considerando que um engenheiro projetasse 150 gates/ dia uma equipe 10 engenheiro levaria 20 anos para projetar um Circuito Integrado de 10 milhões de gates;
- Complexidade cresce com tamanho;

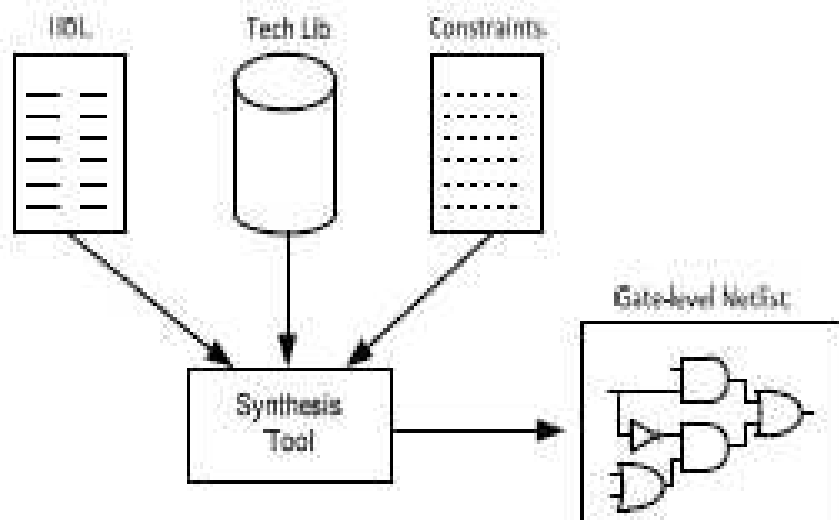
### ● Solução

- Utilizar linguagens HDL e criar uma ferramenta automática que a partir da linguagem HDL gerar o circuito semelhante a compiladores que geram códigos de máquina a partir de linguagens de programação.
- Ferramentas de Síntese.





## Ferramentas de Síntese



### • Entrada

- Código HDL
- Restrições
- Biblioteca de células

### • Saída

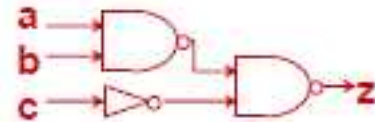
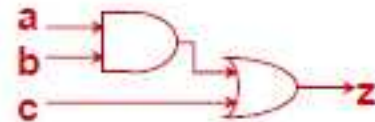
- Gate level netlist

1. Inferir elementos de Lógica e Estados
2. Fazer otimizações independentes da tecnologia;
3. Mapear elementos para tecnologia alvo;
4. Fazer otimizações dependentes da tecnologia (ajustar larguras de gates

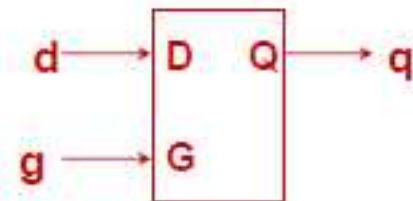


## Mapear Lógica e Estados

```
assign z = (a & b) | c;
```



```
reg q;  
  
// D-latch  
always @(g or d) begin  
    if (g) q <= d;  
end
```





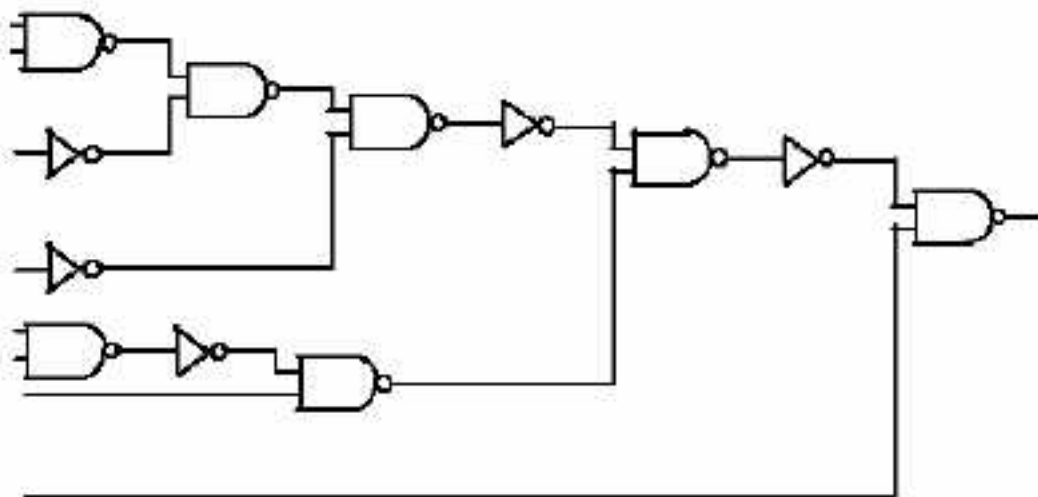
## Otimizações Independentes da Tecnologia

- Dois níveis de minimização booleana: baseada na hipótese que reduzindo o número de termos de produtos e o tamanho destes termos em uma equação booleana o circuito a ser implementado será menor e mais rápido; (método Quine-McCluskey)
- Otimizar FSM (Finite State Machines) buscando encontrar FSMs equivalentes com menos estados que produzam a mesmas saídas dadas a mesma sequência de entradas;
- Escolher codificações de estados das FSM que minimizem área;

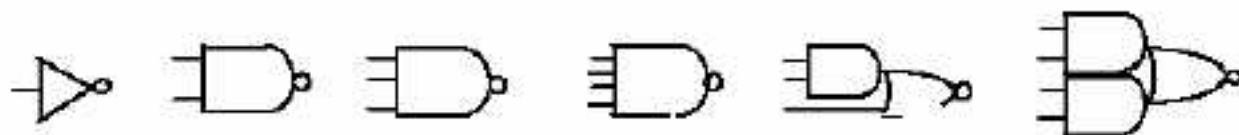


## Mapear para Tecnologia Alvo

### Circuito a ser Implementado



### Biblioteca








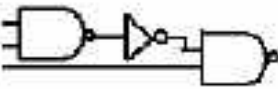

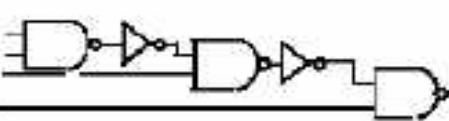
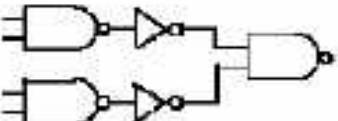

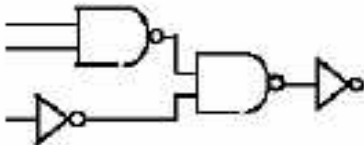
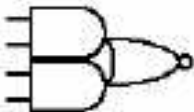
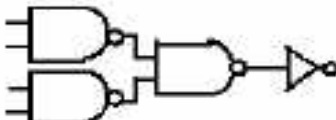


## Cobertura DAG

1. Colocar o circuito alvo na forma normal (NANDs + Inversores);
2. Colocar a biblioteca na forma normal
3. Problema encontrar a combinação ótima de implementação do circuito alvo com os componentes da biblioteca tendo o menor custo de componentes/ área/ ligação (Problema de programação dinâmica)

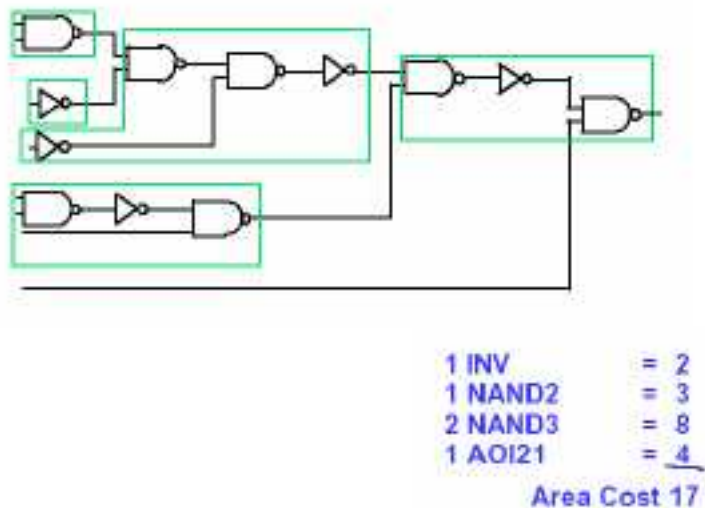
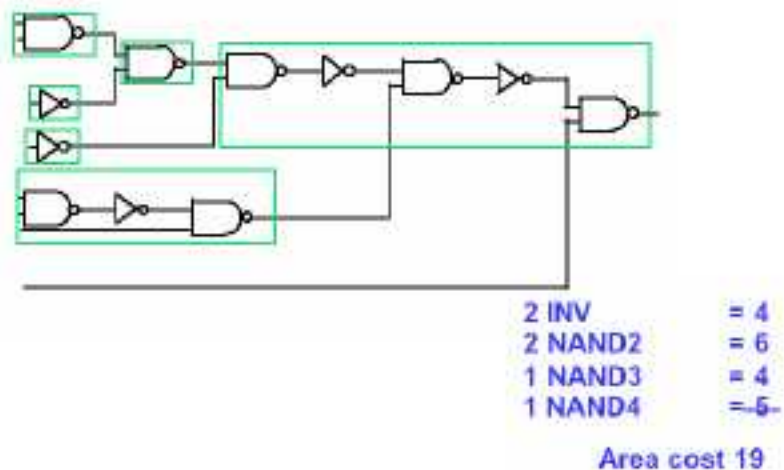
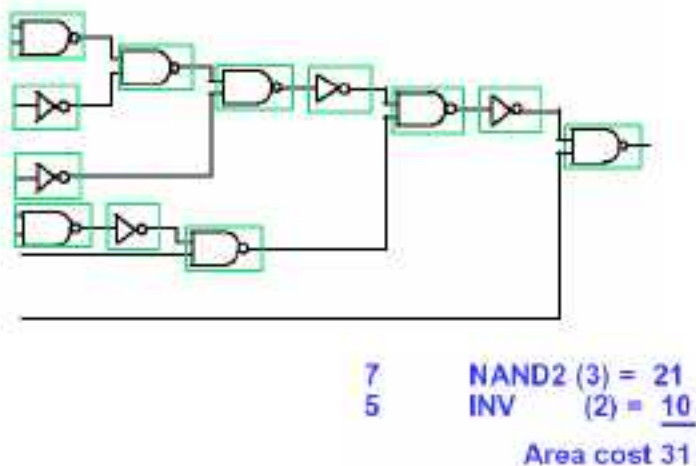


## Primitive DAGs for library gates

	Element/Area Cost		Tree Representation (normal form)	
INVERTER	2			
NAND2	3			
NAND3	4			8
NAND4	5			13
				13
AOI21	4			10
AOI22	5			11



## Mapear para Tecnologia Alvo







## Otimização dependente da Tecnologia

- Adicionar na biblioteca de componentes células mais complexas, reduzindo área nos caminhos críticos;
- Adicionar buffer inversores para melhorar delays nos caminhos críticos;
- Redimensionar transistores nos caminhos críticos;
- Trocar posicionamento de latches e registradores para otimizar o clock do sistema;
- Otimizar o roteamento



## Referências

- <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-884-complex-digital-systems-spring-2005/index.htm>
- [http://www.utdallas.edu/~kad056000/index\\_files/verilog/default.htm](http://www.utdallas.edu/~kad056000/index_files/verilog/default.htm);
- <http://ee.sharif.edu/~asic/>