



Universidade Federal  
de Ouro Preto

Universidade Federal de Ouro Preto – UFOP  
Departamento de Computação e Sistemas - DECSI

# Linguagens de Descrição de Hardware

CSI509 – Organização e Arquitetura de Computadores II  
Prof. Samira Santos da Silva

# Sumário

- Introdução
- Verilog
  - Conceitos Básicos
  - Linguagem
    - Módulos, portas e tipos de dados.
    - Operadores.
    - Estruturas de Controle.
    - Estruturas de Repetição.
    - Atribuição de Variáveis.
    - Test benches.
    - Exemplos e Exercícios.
    - Considerações Gerais.

# Introdução

# Introdução

- Linguagens de Descrição de Hardware
  - Uma **alternativa** à necessidade de se criar **esquemas elétricos** para **descrever** um **círcuito** são as chamadas **linguagens de descrição de hardware** (Hardware Description Languages – HDLs).
  - Muito utilizadas pela **indústria** em **projetos de sistemas digitais**: microcontroladores, microprocessadores, e componentes eletrônicos em geral.
  - Visa prover uma maneira “formal” para se **descrever** um **componente de hardware**.

# Introdução

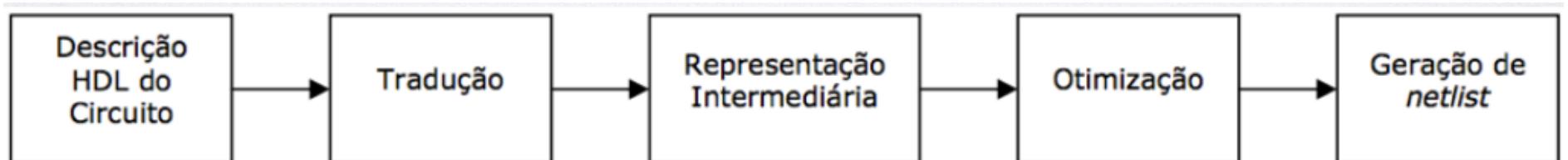
- Linguagens de Descrição de Hardware
  - Peculiaridade em relação a **linguagens de programação convencionais: paralelismo.**
  - Linguagens de descrição de hardware mais conhecidas:
    - Verilog;
    - VHDL (VHSIC - Very High Speed Integrated Circuit – Hardware Description Language);
    - AHDL;
    - etc

# Introdução

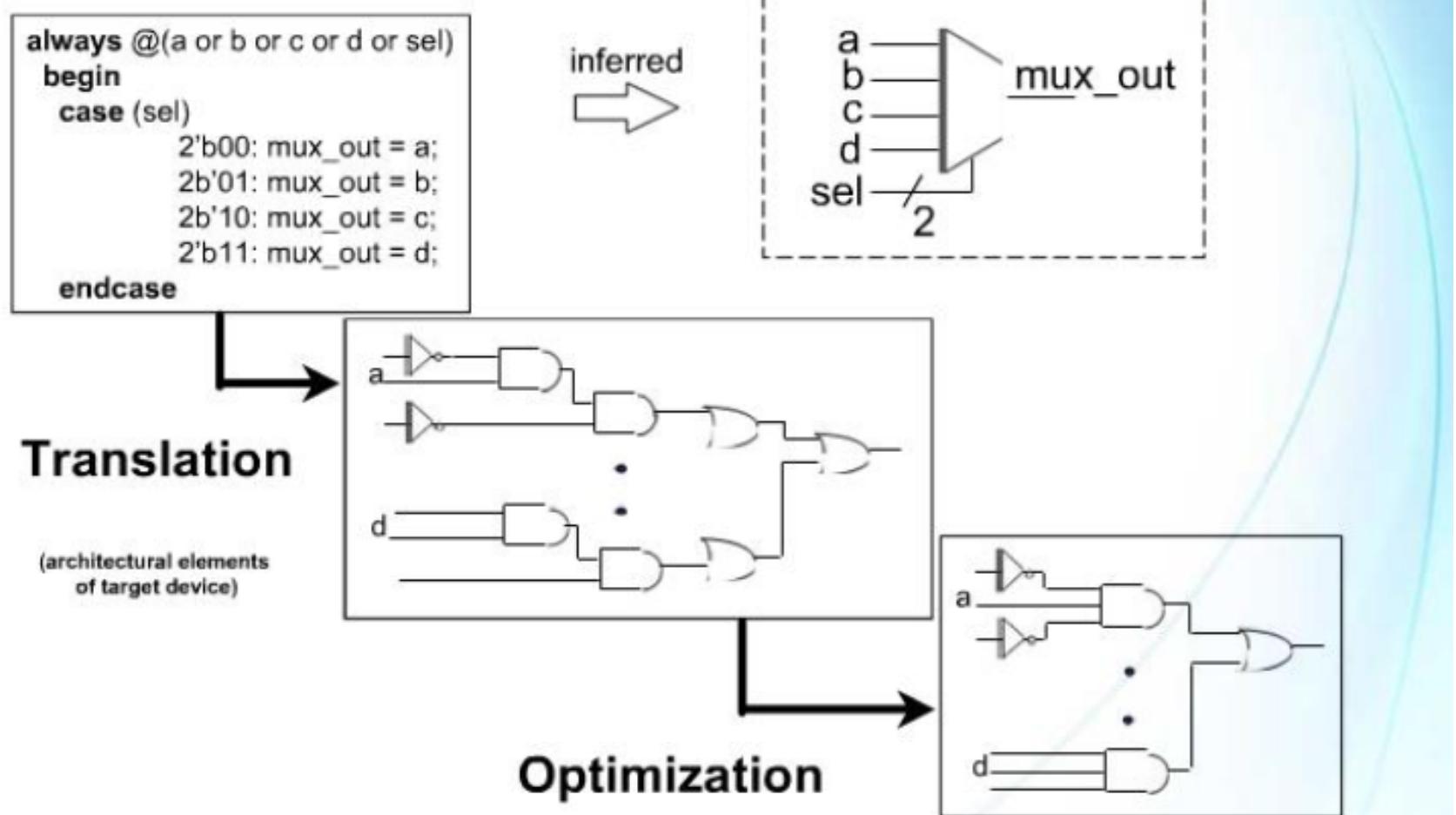
- Linguagens de Descrição de Hardware
  - **Análogo a linguagens de programação convencionais:** Programador cria **arquivo-fonte** utilizando uma **linguagem específica**.
  - Linguagens possuem um **conjunto de regras e sintaxes** de modo a serem compreendidas por um interpretador externo.
  - Semelhantes às linguagens de programação mas são orientadas à descrição das **estruturas e comportamento** do hardware.
  - Descrição **estrutural** descreve a interconexão entre os componentes que fazem parte do circuito.

# Introdução

- Linguagens de Descrição de Hardware
  - Descrição **comportamental** apresenta o funcionamento de cada um dos componentes do circuito.
  - Pode-se utilizar HDLs em conjunto com uma **biblioteca de componentes** que **interpretados** por uma **ferramenta de síntese lógica** gera automaticamente um **circuito digital**.
  - Principais etapas da síntese lógica:



# Introdução



# Introdução

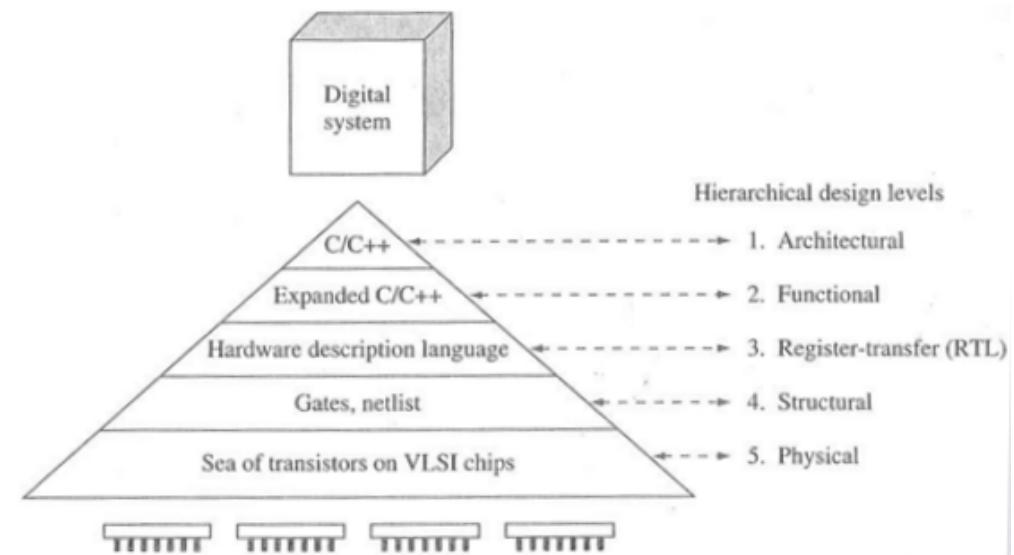
- Linguagens de Descrição de Hardware
  - **netlist:** Listagem das **estruturas internas de armazenamento, lógica combinatória** e estrutura de **conexão dos componentes.**
  - Vantagens das HDLs:
    - Menor espaço ocupado na placa final;
    - Menor consumo de energia;
    - Maior confiabilidade;
    - Menor complexidade de desenvolvimento;
    - Menor custo.

# Introdução

- Linguagens de Descrição de Hardware
  - Fluxo de projetos:
    - Assim como o desenvolvimento de software, a **Descrição do hardware** também deve passar por **determinadas etapas**:
    - Especificação em alto nível;
    - Codificação;
    - Verificação / testes;
    - Sintetização.

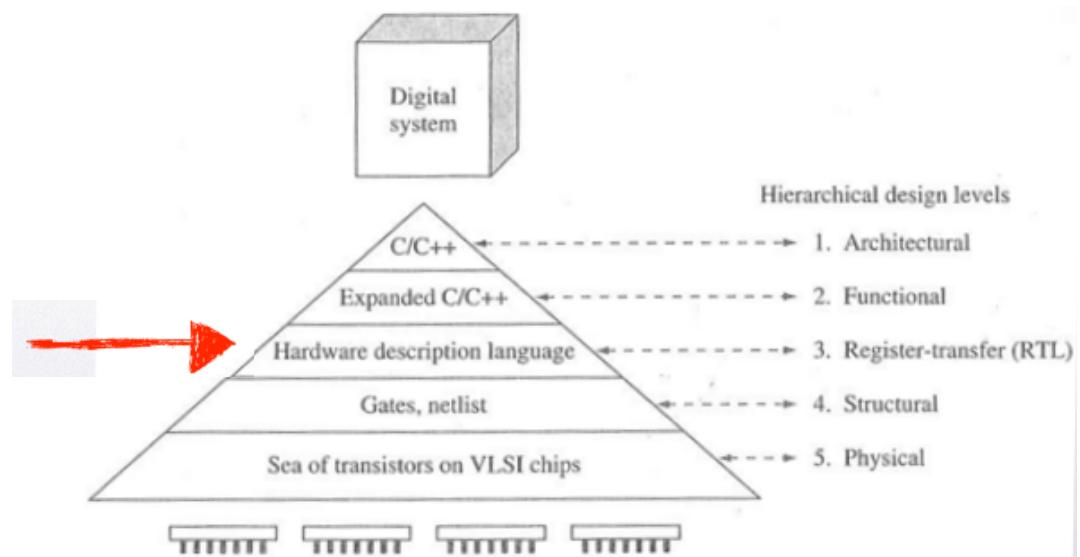
# Introdução

- Linguagens de Descrição de Hardware
  - Fluxo de projetos:
    - Assim como o desenvolvimento de software, a **Descrição do hardware** também deve passar por **determinadas etapas**:
    - Especificação em alto nível;
    - Codificação;
    - Verificação / testes;
    - Sintetização.



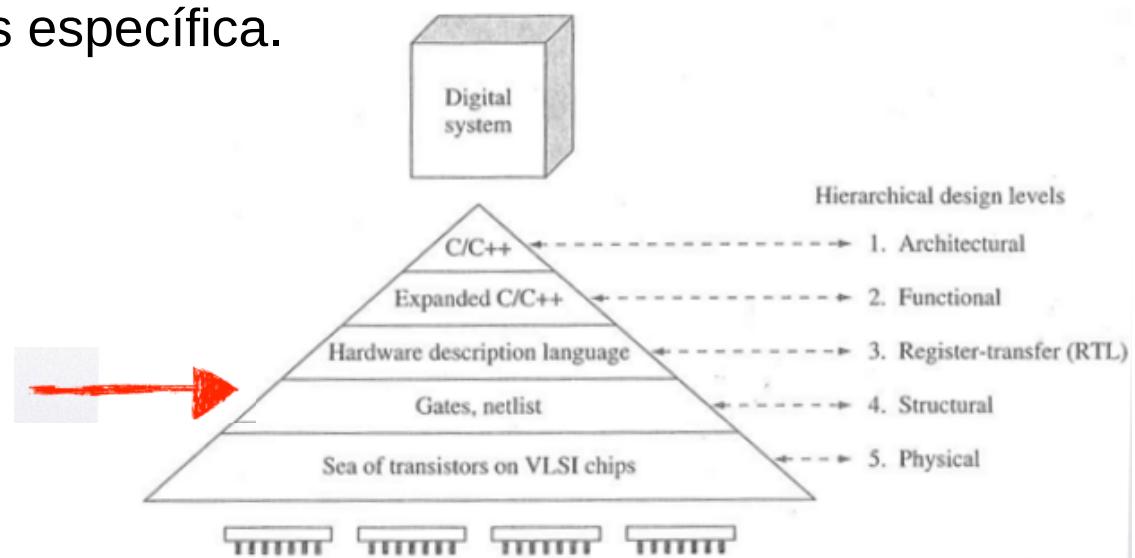
# Introdução

- Linguagens de Descrição de Hardware
  - Fluxo de projetos: **Nível Comportamental**
    - Refinamento lógico da especificação.
    - Fornece informações funcionais precisas sobre como o sistema reage a cada operação.



# Introdução

- Linguagens de Descrição de Hardware
  - Fluxo de projetos: **Nível Estrutural**
    - Descreve como funções são realmente implementadas.
    - Define número de ciclos de clock necessários para conclusão de cada operação.
    - Apresenta mapeamento do modelo comportamental para implementação mais específica.



# Introdução

- Linguagens de Descrição de Hardware
  - Fluxo de projetos: **Nível Físico**
    - Descreve detalhes a nível do chip, layout e transistores.
    - Mapeia o nível estrutural em uma tecnologia específica para conclusão de cada operação.



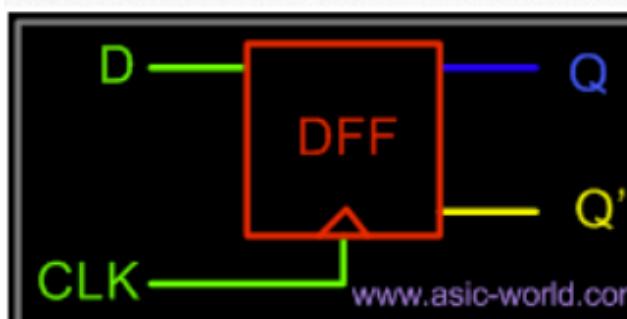
# Introdução

- Linguagens de Descrição de Hardware
  - Fluxo de projetos: **Boas práticas**
    - Quanto **mais bem escrito** é um código, **mais chances** o mesmo tem de poder ser **otimizado** durante a **síntese**.
      - Separação e particionamento do projeto em **blocos** e **módulos**.
      - **Convenção de nomes** utilizados facilita a leitura do código.

Verilog

# Verilog

- Conceitos Básicos
  - Verilog é uma das linguagens de descrição de hardware.
  - Com o Verilog é possível **descrever a organização e comportamento** de componentes de hardware.
  - Ex.:



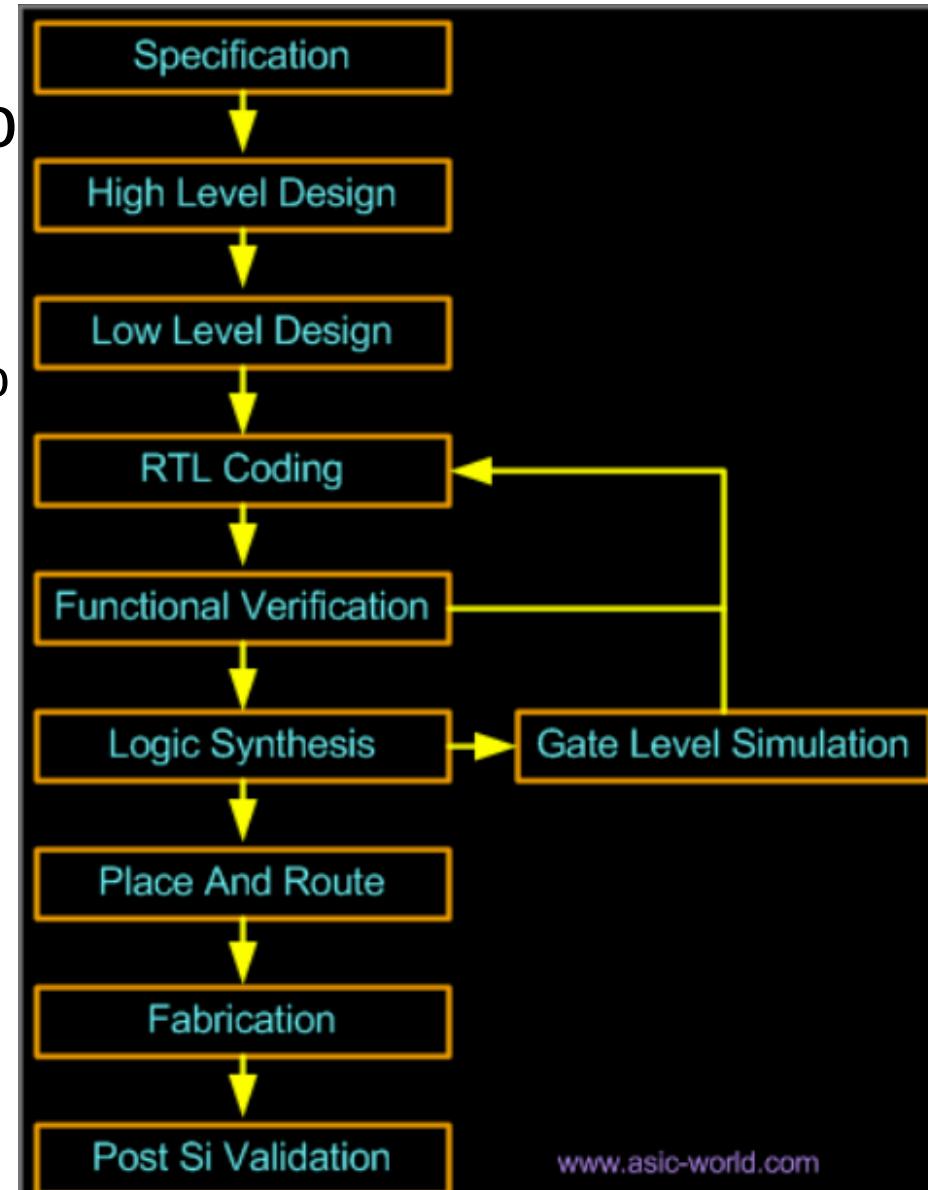
```
1 // D flip-flop Code
2 module d_ff ( d, clk, q, q_bar);
3   input d ,clk;
4   output q, q_bar;
5   wire d ,clk;
6   reg q, q_bar;
7
8   always @ (posedge clk)
9   begin
10     q <= d;
11     q_bar <= ! d;
12   end
13
14 endmodule
```

# Verilog

- Conceitos Básicos
  - Estilos de desenvolvimento
    - **Bottom-up**
      - Especificação inicia-se em **nível mais baixo** (nível de portas lógicas).
      - Modelo **impossível** de se manter atualmente devido a **complexidade dos processadores/microcontroladores**.
      - Recentemente tem dado lugar ao desenvolvimento hierárquico e estrutural.

# Verilog

- Conceitos Básicos
  - Estilos de desenvolvimento
    - **Top-down**
      - Desenvolvimento inicia-se no nível **mais abstrato** para o **menos abstrato**.



# Verilog

- Conceitos Básicos
  - Níveis da abstração do Verilog
    - **Comportamental**
      - Descreve um sistema à partir de **diversos algoritmos** que executam de forma **concorrente** entre si.
      - Entretanto, **cada** um dos **algoritmos** (internamente) funciona de forma **sequencial**.
      - Algoritmos consistem de um conjunto de instruções executadas uma após a outra.
      - Neste nível não existe **nenhuma especificação** de **como** a **estrutura** deve **funcionar**.

# Verilog

- Conceitos Básicos
  - Níveis da abstração do Verilog
    - **Transferência de registradores**
      - Especifica as **características** de um **circuito** através de **operações básicas**.
      - Considera ainda os **dados transferidos** entre **registradores**.
      - **Operações** são **escalonadas** para acontecer em determinados intervalos/tempos específicos.

# Verilog

- Conceitos Básicos
  - Níveis da abstração do Verilog
    - **Nível de portas lógicas**
      - Descreve as **características do hardware** em **nível lógico** através de **links e propriedades** relacionadas a **temporização**.
      - Operações são pré-definidas: AND, OR, NOT e outras portas lógicas conhecidas.
      - Devido a **complexidade**, normalmente **não se desenvolve** nesse nível. Deixa-se a geração do mesmo para **ferramentas**.

# Verilog

- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog
    - Desenvolvimento na **linguagem Verilog** se aproxima muito da **programação convencional** de sistemas que é focada mais na **execução sequencial**. Em Verilog é comum que **partes do código** executem **em paralelo**.
  - Passos:
    - Especificações;
    - Design de alto-nível;
    - Design de baixo-nível;
    - Código RTL (Register Transfer Level);
    - Verificações;
    - Síntese.

# Verilog

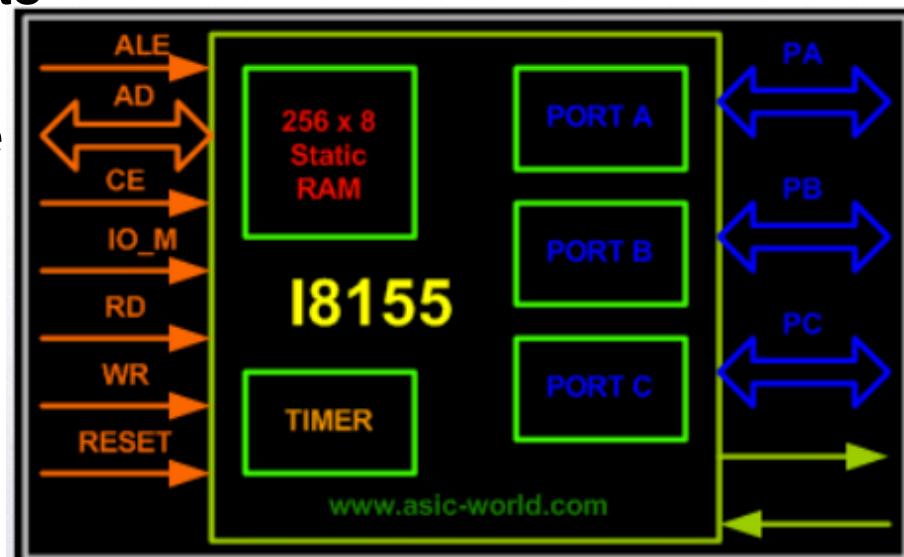
- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog
    - **Especificações:**
      - Estágio no qual os principais **requisitos** do sistema/design são **definidos**.
      - Quais são os requisitos e restrições do design? O que se deseja criar?
      - Ex.: Criação de um **contador** em hardware que suporte um **máximo de 4bits**; reset síncrono com ativação em high. Quando reset ativo contador deverá ir para “0”.

# Verilog

- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog

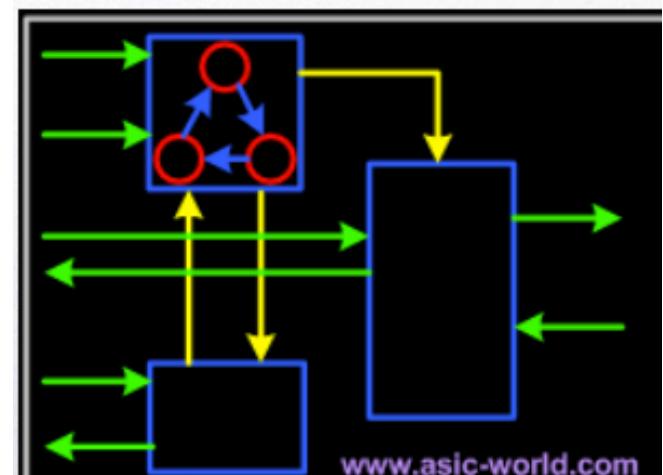
- **Design de alto nível:**

- Estágio no qual é **definido** os **vários blocos** componentes do design e a **forma** com a qual os mesmos se **comunicam**.
    - No projeto de um **microprocessador**, tal fase corresponde à **separação dos blocos** de acordo com suas **funcionalidades**: ALU, banco de registradores, memórias, etc.



# Verilog

- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog
    - **Design de baixo nível:**
      - Fase na qual se descreve **como cada um dos blocos é implementado.**
      - Contém detalhes das **máquinas de estados, contadores, multiplexadores, decodificadores, registradores internos**, etc.
      - Comumente é a fase na qual se **gasta mais tempo** durante o design de um hardware.



# Verilog

- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog
    - **Codificação RTL (Register Transfer Level):**
      - Design de baixo nível é convertido em **linguagem Verilog/VHDL** utilizando-se construções dessas linguagens.

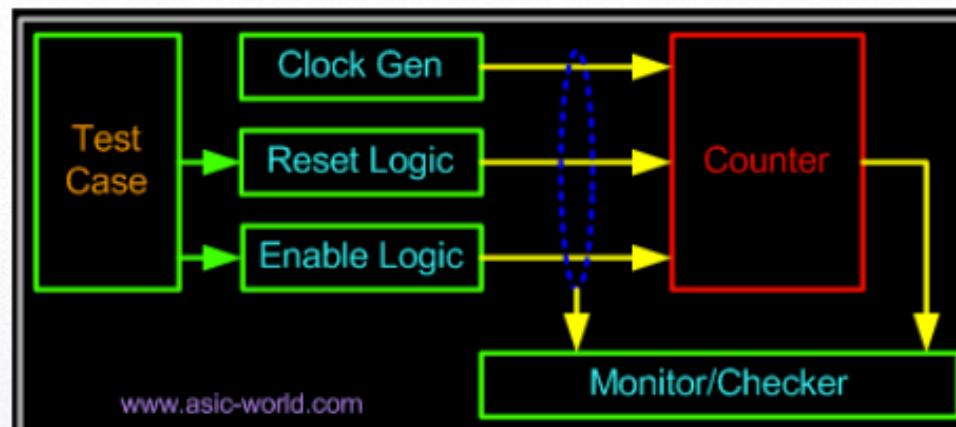
```
module addbit (
    a,          //entrada 1
    b,          //entrada 2
    ci,         //carry in
    sum,        //saída da soma
    co          //carry out
);
    //entradas
    input a;
    input b;
    input ci;
    //saídas
    output sum;
    output co;
    //tipos de dados das portas
    wire a;
    wire b;
    wire ci;
    wire sum;
    wire co;
    //código
    assign {co,sum} = a + b + ci;
endmodule
```

# Verilog

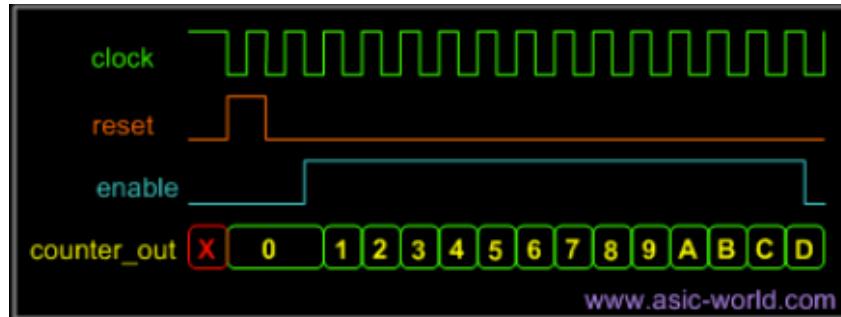
- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog
    - **Simulação:**
      - É o processo de **verificação** das **características funcionais** dos módulos desenvolvidos - em qualquer nível.
      - Simuladores são utilizados para **verificar o comportamento do hardware** projeto **sem que haja necessidade de implementá-lo**.
      - Para testar se o código cobre os requisitos é necessário se **gerar entradas e analisar as saídas** dos módulos.
      - 60% a 70% do tempo do projeto pode ser gasto na parte de simulação e verificação..

# Verilog

- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog
    - **Simulação:**
    - Típico ambiente de simulação:

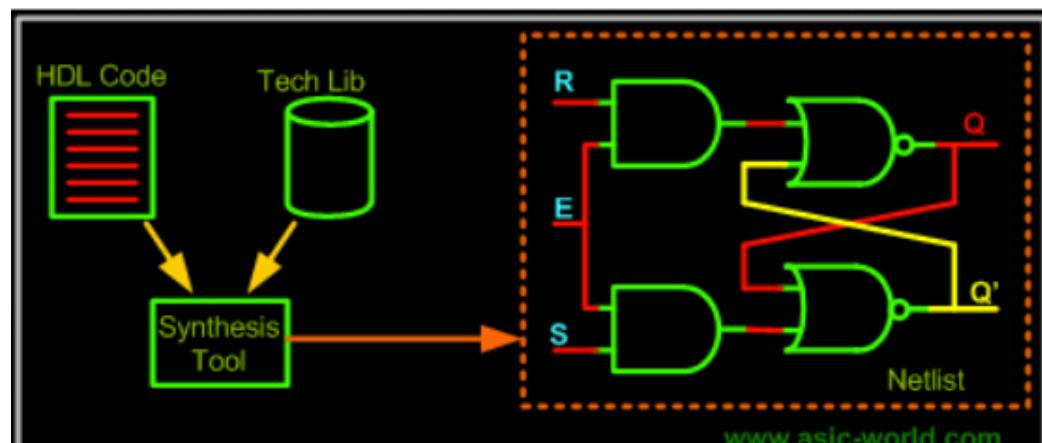


- Saída em forma de onda pode ser avaliada para uma validação adicional:



# Verilog

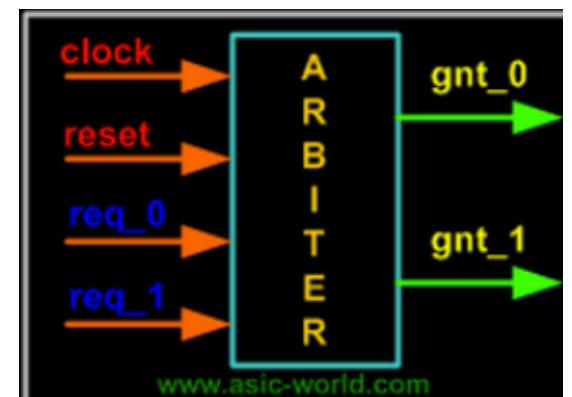
- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog
    - Síntese:
      - Processo no qual **ferramentas analisam as entradas** (código verilog, restrições, tecnologia alvo) e as **mapeiam em primitivas** para a **tecnologia alvo**.
      - Após mapear o código RTL para as portas (gates) também fazem testes mínimos de temporização.



# Verilog

- Conceitos Básicos
  - Curva esperada de aprendizado do Verilog
    - **Exemplo: Circuito arbitrador**
      - Como exemplo para os próximos passos utilizaremos o projeto de um circuito arbitrador.
      - Um arbitrador é um circuito que provê duas saídas, e é utilizado para determinar a ordem de chegada dos sinais de entrada.
    - Circuito arbitrador (Especificações):
      - Suporte a dois agentes;
      - Reset assíncrono ativo em “alto”;
      - Prioridade do agente 0 sobre o agente 1;
      - Acesso será garantido assim que condições forem satisfeitas.

Diagrama de Blocos

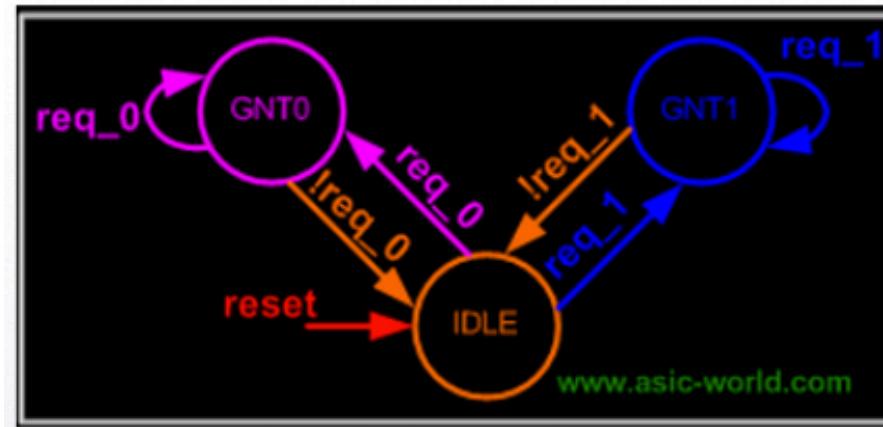


# Verilog

- Conceitos Básicos
  - Desenvolvimento **tradicional** do arbitrador:
    - Desenho do diagrama de blocos;
    - Desenho da máquina de estados;
    - Criação da tabela verdade com as possíveis transições para cada flip-flop;
    - Criação dos mapas de Karnaugh.
    - Desenvolvimento tradicional funciona bem para pequenos projetos. Projetos maiores se tornam complexos e sujeitos a erros.

# Verilog

- Conceitos Básicos
  - Arbitrador utilizando Verilog
    - Considerando o diagrama anterior, as entradas são tratadas da seguinte forma:



- Entradas e tratamento de cada uma podem ser mapeados em uma máquina de estados.

# Verilog

- Linguagem
  - Módulos
    - Cada pedaço componente (**caixa-preta**) é chamado em Verilog de **módulo**.
    - “*module*” é então utilizado como uma palavra reservada da linguagem.
    - Como o exemplo do arbitrador, cada módulo possui **entradas, saídas e lógica de funcionamento interno**.
    - É possível dizer que **internamente** um **módulo** é o **equivalente** a uma “**função**” em linguagens de programação convencionais.

# Verilog

- Linguagem
  - Portas
    - Cada módulo possui um **nome e portas** (de entrada e saída) que devem ser declaradas.
    - Ainda é possível se ter portas **bidirecionais** declaradas através da palavra reservada “**inout**”.

```
module arbiter (
    //Comentário
    clock, //Clock
    reset, //Reset
    req_0, //Request 0
    req_1, //Request 1
    gnt_0, //Grant 0
    gnt_1 //Grant 1
);

    //-- Portas de entrada --
    input clock;
    input reset;
    input req_0;
    input req_1;

    //-- Portas de saída --
    output gnt_0;
    output gnt_1;
endmodule
```

# Verilog

- Linguagem
  - Portas
    - Ex. porta bidirecional:

```
inout read_enable; //porta bidirecional de nome read_enable
```
    - Ex. vetor representando endereço de 8 bits (*little endian*):

```
inout[7:0] address; //porta bidirecional representando endereço
```
    - Ex. vetor representando endereço de 8 bits (*big endian*):

```
inout[0:7] address; //porta bidirecional representando endereço
```

# Verilog

- Linguagem
  - Tipos de Dados
    - Qual a relação de **tipos de dados** com o **hardware** em si?
    - Tipos de dados **não são tão comuns** aqui quanto em linguagens convencionais.
    - Entretanto, em Verilog temos os **drivers** com usos bem comuns:
      - Armazenamento de valores (ex.: *flip-flop*). Em Verilog: “**reg**”.
      - Conexão entre dois pontos (ex.: um fio/trilha). Em Verilog: “**wire**”.

```
wire and_gate_output; //fio somente saída
reg d_flip_flop_output; //registrador que armazena um sinal
reg [7:0] address_bus; //registrador de 8-bits
```

# Verilog

- Linguagem
  - Operadores
    - Em Verilog funcionam da **mesma forma** que em linguagens de programação convencionais.
    - Toma-se **dois valores** e se “opera” entre os mesmos, gerando um terceiro como **resultado**.
    - Praticamente **todos os operadores** utilizados na linguagem C são **suportados** em Verilog.
    - Ex.: Adição, subtração, and, or, etc.

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+ -	Unary plus Unary minus
	!	Logical negation
Logical	&&	Logical and
		Logical or
	> < >= <=	Greater than Less than Greater than or equal Less than or equal
Relational	==	Equality
	!=	Inequality
Reduction	~	Bitwise negation
	~&	nand
		or
	~	nor
	^	xor
	^~	xnor
	~~	xnor
	>> <<	Right shift Left shift
Concatenation	{ }	Concatenation
Conditional	?	conditional

# Verilog

- Linguagem
  - Estruturas de Controle
    - Utilizadas para **controle** e **alteração do fluxo** de execução em linguagens convencionais.
    - Em Verilog possuem a **mesma função** que nestas linguagens.
    - **Cuidado:** Verilog descreve o hardware. Determinadas construções da linguagem **podem não ser passíveis** de implementação.

# Verilog

- Linguagem
  - Estruturas de Controle

- If-else:

```
//begin e end são como { }
if (enable == 1'b1) begin
    data = 10; //decimal
    address = 16'hDEAD; //hexadecimal
    wr_enable = 1'b1; //binário
end else begin
    data = 32'b0;
    wr_enable = 1'b0;
    address = address + 1;
end
```

- Boa política: **Descrever sempre o caso “else”**, pois pode-se gerar um problema caso seja omitido e circuito termine em um *latch* (duas entradas set/reset e uma saída).

# Verilog

- Linguagem
  - Estruturas de Controle
    - **case:**
      - Utilizado quando se possui **uma variável** que deve ser verificada contra **múltiplos valores**.
      - **Boa política:** Sempre implementar o caso “default” para evitar que a FSM caia em um estado não previsto.

```
case (address)
  0 : $display("It is 11:40pm");
  1 : $display("I'm feeling sleepy");
  2 : $display("Very sleepy");
  default : $display("I already woke up!");
endcase
```

# Verilog

- Linguagem
  - Estruturas de Controle
    - **case:**
      - Normalmente são utilizados em **substituição** a vários *if-else's* aninhados.
      - Se nem todos os casos são cobertos (e/ou não existe o caso *default*), e se está escrevendo um circuito combinacional, a ferramenta de síntese pode inferir a ocorrência de um *latch* como solução.

# Verilog

- Linguagem
  - Estruturas de Repetição
    - **while:**
      - Assim como em linguagens convencionais, executa determinado trecho de código **enquanto a condição** especificada for **válida**.
      - Normalmente não são utilizados em códigos reais, mas sim em **conjuntos de testes**.

```
while (free_time) begin
    $display ("Keep teaching some classes!!");
end
```

# Verilog

- Linguagem
  - Estruturas de Repetição

- *always*:

- Considerando o módulo:
- Bloco “*always*” é executado de forma paralela sempre que a condição especificada é atingida.
- Pode se utilizar o comando “*disable*” para desativar um bloco quando condição é satisfeita.

```
module counter (clk, rst, enable, count);
    input clk, rst, enable;
    output [3:0] count;
    reg [3:0] count;

    always @ (posedge clk or posedge rst)
        if (rst) begin
            count <= 0;
        end else begin : COUNT
            while (enable) begin
                count <= count + 1;
                disable COUNT;
            end
        end
    endmodule
```

# Verilog

- Linguagem
  - Estruturas de Repetição
    - **for:**
      - Loops for são implementados em Verilog assim como em C/C++. Exceto pelo incremento (sem suporte a “++”/“--”).
      - **Cuidado:** Não implementar loops infinitos. Comportamento pode não ser esperado, causando um erro na descrição do hardware.

```
for (i = 0; i<16; i = i + 1) begin
    $display("Current value of i is %d",i);
end
```

# Verilog

- Linguagem
  - Estruturas de Repetição
    - repeat:
      - Muito parecido com a estrutura *for*, entretanto **não utiliza uma variável de controle explicitamente.**
      - Em projetos reais **não é muito comum** o uso das estruturas de repetição *for* e *repeat*.

```
repeat (16) begin
    $display("Current value of i is %d",i);
    i = i+1;
end
```

# Verilog

- Linguagem
  - Atribuição de variáveis
    - Em hardware, dois tipos de elementos: **combinacionais / sequenciais.**
    - Em Verilog existem **duas maneiras** de se modelar a **lógica combinacional** e **uma maneira** de se modelar a **lógica sequencial**.
      - **Combinacionais**: Modelados utilizando-se “**assign**” e “**always**”.
      - **Sequenciais**: Modelados utilizando-se “**always**”.
      - Testes: Pode-se utilizar blocos de inicialização (“**initial**”) quando tempo=0.

```
initial begin
    clk = 0;
    reset = 0;
    req_0 = 0;
    req_1 = 0;
end
```

# Verilog

- Linguagem
  - Atribuição de variáveis
    - Bloco **always**
      - Como o próprio nome diz, um bloco *always* é **executado continuamente** à partir do seu “lançamento”.
      - A **condição** para que o bloco seja lançado é normalmente **descrita após o sinal de @**.
      - **Restrição:** Não lida com o tipo de dados *wire*, mas pode tratar *reg* e inteiros convencionais.

```
always @ (a or b or sel)
begin
    y = 0;
    if (sel == 0) begin
        y = a;
    end else begin
        y = b;
    end
end
```

Mux 2:I

# Verilog

- Linguagem
  - Atribuição de variáveis
    - Bloco **always**
      - Duas possíveis maneiras de se listar as condições:
        - Sensível a nível (circuitos combinacional): Exemplo anterior.
        - Sensível a borda (flip-flops):
        - Operador “`=`” é **atribuição bloqueante** para blocos combinacionais.
        - Operador “`<=`” é **atribuição não-bloqueante** para blocos sequenciais.

```
always @ (posedge clk)
if (reset == 0) begin
    y <= 0;
end else if (sel == 0) begin
    y <= a;
end else begin
    y <= b;
end
```

Mux 2:I

# Verilog

- Linguagem
  - Atribuição de variáveis
    - Task e function
      - Como forma de **reuso do código**, Verilog provê suporte a duas estruturas conhecidas como *task* e *function*.
      - Ex.: Código utilizado para cálculo de paridade par:
      - task pode conter **delays** enquanto function não.
      - function retorna um único valor, e task não.

```
function parity;
    input [31:0] data;
    integer i;
begin
    parity = 0;
    for (i=0; i<32; i=i+1) begin
        parity = parity ^ data[i];
    end
end
endfunction
```

# Verilog

- Linguagem
  - *Test benches*
    - Código criado - de acordo com as regras da linguagem -precisa ser **testado** e **validado**.
    - **Testes e validações** precisam ser feitos para assegurar que realmente estão **de acordo** com a **especificação**.
    - É uma **maneira** de se **testar** códigos criados **sem** que necessariamente tenha-se que **considerar** os **passos seguintes**.
    - Provê **flexibilidade** e **economia de tempo**.

# Verilog

- Linguagem
  - *Test benches* - Exemplo arbitrador:
    - (Aloca acesso a recursos compartilhados)

```
//Código do módulo
module arbiter (
    clock,
    reset,
    req_0,
    req_1,
    gnt_0,
    gnt_1,
);
    input clock, reset, req_0, req_1;
    output gnt_0, gnt_1;
    reg gnt_0, gnt_1;
```

```
always @ (posedge clock or posedge reset)
    if (reset) begin
        gnt_0 <= 0;
        gnt_1 <= 0;
    end else if (req_0) begin
        gnt_0 <= 1;
        gnt_1 <= 0;
    end else if (req_1) begin
        gnt_0 <= 0;
        gnt_1 <= 1;
    end
endmodule
```

# Verilog

- Linguagem
  - *Test benches* - Exemplo arbitrador:

```
//Código do test bench
module arbiter_tb;
    reg clock, reset, req0, req1;
    wire gnt0, gnt1;
    initial begin
        $monitor("clock=%b,reset=%b,req0=%b,req1=%b,gnt0=%b,gnt1=%b",clock,reset,req0,req1,gnt0,gnt1);
        clock=0;
        reset=0;
        req0=0;
        req1=0;
        #5 reset=1;
        #15 reset=0;
        #10 req0 = 1;
        #10 req0 = 0;
        #10 req1 = 1;
        #10 req1 = 0;
        #10 {req0,req1} = 2'b11;
        #10 {req0,req1} = 2'b00;
        #10 $finish;
    end
```

```
always begin
    #5 clock = !clock;
end

arbiter U0 (
    .clock(clock),
    .reset(reset),
    .req_0(req0),
    .req_1(req1),
    .gnt_0(gnt0),
    .gnt_1(gnt1)
);
endmodule
```

# Verilog

- Linguagem
  - *Test benches – Exemplo arbitrador: (saída)*

```
clock=0,reset=0,req0=0,req1=0,gnt0=x,gnt1=x  
clock=1,reset=1,req0=0,req1=0,gnt0=0,gnt1=0  
clock=0,reset=1,req0=0,req1=0,gnt0=0,gnt1=0  
clock=1,reset=1,req0=0,req1=0,gnt0=0,gnt1=0  
clock=0,reset=0,req0=0,req1=0,gnt0=0,gnt1=0  
clock=1,reset=0,req0=0,req1=0,gnt0=0,gnt1=0  
clock=0,reset=0,req0=1,req1=0,gnt0=0,gnt1=0  
clock=1,reset=0,req0=1,req1=0,gnt0=1,gnt1=0  
clock=0,reset=0,req0=0,req1=0,gnt0=1,gnt1=0  
clock=1,reset=0,req0=0,req1=0,gnt0=1,gnt1=0  
clock=0,reset=0,req0=0,req1=1,gnt0=1,gnt1=0  
clock=1,reset=0,req0=0,req1=1,gnt0=0,gnt1=1  
clock=0,reset=0,req0=0,req1=0,gnt0=0,gnt1=1  
clock=1,reset=0,req0=0,req1=0,gnt0=0,gnt1=1  
clock=0,reset=0,req0=1,req1=1,gnt0=0,gnt1=1  
clock=1,reset=0,req0=1,req1=1,gnt0=1,gnt1=0  
clock=0,reset=0,req0=0,req1=0,gnt0=1,gnt1=0  
clock=1,reset=0,req0=0,req1=0,gnt0=1,gnt1=0  
clock=0,reset=0,req0=0,req1=0,gnt0=1,gnt1=0
```

# Verilog

- Linguagem
  - Exemplo 1: Hello World

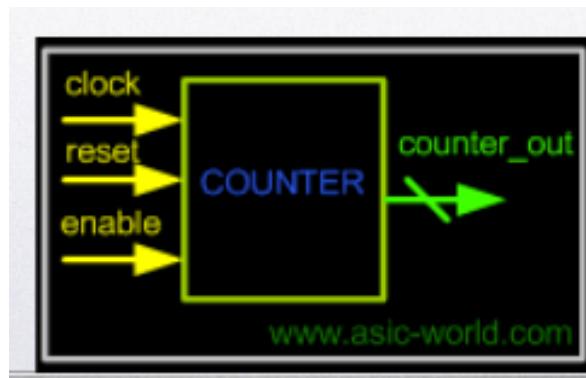
```
module helloWorld;  
    initial begin  
        $display ("Hello world!!");  
        #10 $finish;  
    end  
endmodule
```

- Saída:

```
$ Hello world!!
```

# Verilog

- Linguagem
  - Exemplo 2: Contador
    - Especificações:
      - Contador síncrono de 4bits;
      - *reset* síncrono ativo em *high*;
      - *enable* ativo em *high*;
      - A cada ciclo de *clock*, incremento de 1.



# Verilog

- Linguagem
  - Exemplo 2: Contador

```
module counter (clock, reset, enable, counter_out);
    input clock;
    input reset;
    input enable;

    wire clock;
    wire reset;
    wire enable;

    output reg[3:0] counter_out;

    always @ (posedge clock)
        begin: COUNTER
            if (reset == 1'b1) begin
                #1 counter_out <= 4'b0000;
            end else if (enable == 1'b1) begin
                #1 counter_out <= counter_out + 1;
            end
        end //do bloco COUNTER
endmodule
```

# Verilog

- Linguagem
  - Exemplo 2: *Testbed*
    - Exercício: À partir do código anterior, criar um *testbed* para verificar o correto funcionamento do hardware contador.

# Verilog

- Linguagem
  - Exemplo 2: *Testbed*
    - Execício: Possível solução

```
'include "counter.v"
module counter_tb();
    reg clock, reset, enable;
    wire [3:0] counter_out;

    initial begin
        $display("time\t clk reset enable counter");
        $monitor("%g\t %b %b %b %b", $time, clock,
            reset, enable, counter_out);
        clock = 1;
        reset = 0;
        enable = 0;
        #5 reset = 1;
        #10 reset = 0;
        #10 enable = 1;
        #100 enable = 0;
        #5 $finish;
    end

    always begin
        #5 clock = ~clock;
    end

    counter U_counter (clock, reset, enable,
        counter_out);
endmodule
```

# Verilog

- Linguagem
  - Considerações Gerais:
    - Escrita do código:

. Mal escrito:

```
module addbit(a,b,,ci,sum,co);
  input a,b,ci; output sum,co;
  wire a,b,ci,sum,co;
endmodule
```

. Bem escrito:

```
module addbit (
  a,
  b,
  ci,
  sum,
  co);
  input a;
  input b;
  input ci;
  output sum;
  output co;
  wire a;
  wire b;
  wire ci;
  wire sum;
  wire co;
endmodule
```

# Verilog

- Linguagem
    - Considerações Gerais:
      - Escrita do código:
        - Verilog é “case-sensitive”.
        - Não é possível se iniciar identificadores com valores numéricos.
        - Armazenamento de valores:

# Verilog

- Linguagem
  - Considerações Gerais:
    - Escrita do código:

- Módulos conectados por **ordem das portas** (implícito)

```
addbit u0 (
    r1[0],
    r2[0],
    ci,
    result[0],
    c1);
```

```
addbit u1 (
    r1[1],
    r2[1],
    c1,
    result[1],
    c2);
```

- Módulos conectados por **nome** (explícito)

```
addbit u0 (
    .a (r1[0]),
    .b (r2[0]),
    .ci (ci),
    .sum(result[0]),
    .co (c1)
);
```

```
addbit u1 (
    .a (r1[1]),
    .b (r2[1]),
    .ci (c1),
    .sum(result[1]),
    .co (c2)
);
```

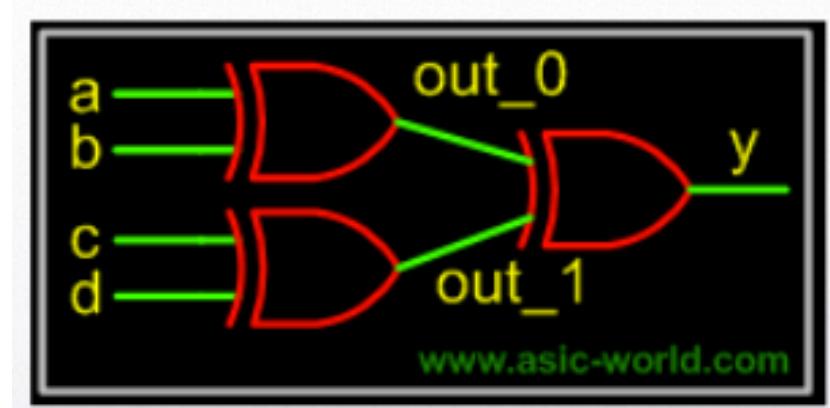
# Verilog

- Linguagem
  - Considerações Gerais:
    - Instanciamento de um módulo:

```
module parity (a, b, c, d, y);
    input a,b,c,d;
    output y;

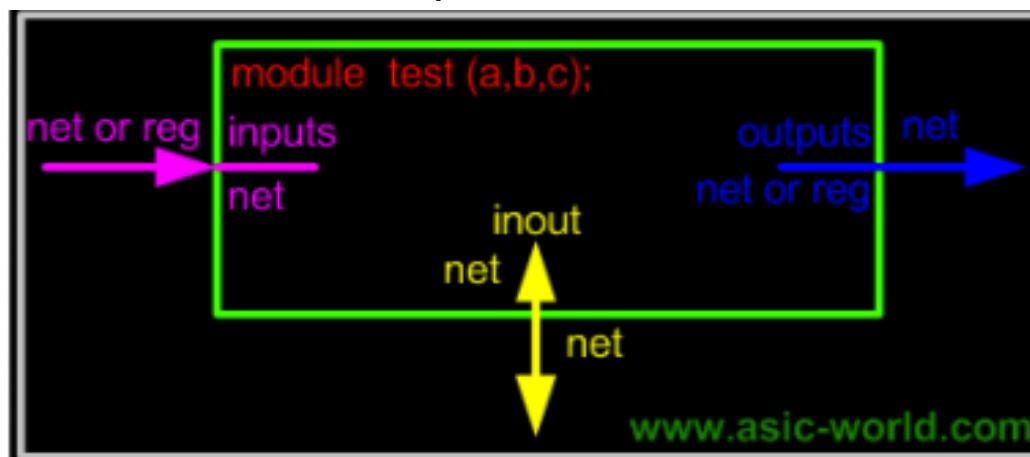
    wire a,b,c,d,y;
    wire out_0, out_1;

    xor u0 (out_0,a,b);
    xor u1 (out_1,c,d);
    xor u2 (y,out_0,out_1);
endmodule
```



# Verilog

- Linguagem
  - Considerações Gerais:
    - Regras de conexões de portas
      - Entradas: Internamente necessitam ser tratadas como tipo *net* (**wire**). Externamente as entradas podem ser conectadas a vars. do tipo **reg** ou **wire**.
      - Saídas: Internamente podem ser **reg** ou **wire**. Externamente devem ser conectadas a variáveis do tipo **wire**.
      - Inouts: Internamente ou externamente só podem ser do tipo **wire** e conectadas a este mesmo tipo.



# Verilog

- Exercício:

Implementar um circuito decodificador definido pela Tabela:

A	Decode
00	0001
01	0010
10	0100
11	1000