CURSO BÁSICO DE VERILOG

Seção 1: Visão geral do Verilog

Verilog não é uma linguagem de programação de *software*. Uma linguagem de programação de *software* é utilizada para executar funções em um processador baseadas em um conjunto de instruções. Por outro lado, Verilog é uma linguagem de descrição de *hardware*, cujo objetivo é especificar a descrição do comportamento de um circuito que eventualmente será implementado em hardware.

A linguagem Verilog é um padrão IEEE, é uma linguagem de descrição de alto nível utilizada tanto para simulação quanto para síntese. Um subgrupo da linguagem é tipicamente utilizado para propósitos de síntese e a linguagem inteira pode ser utilizada para modelamento e simulação. Os padrões são de 1995 e de 2001. System Verilog é outro padrão que acessa uma extensão do Verilog original e é tipicamente utilizado para modelamento a nível de sistema e verificação.

Na literatura comum sobre Verilog ou outra linguagem de alto nível, observa-se alguns termos comuns que serão definidos posteriormente.

- *HDL* significa *Hardware Description Language* - Linguagem de Descrição de Hardware, e consiste em uma linguagem de programação de software utilizada para modelar um hardware ou parte dele;
- *RTL* - *Register Transfer Level* - Nível de Transferência de Registradores, define basicamente relações entre entradas e saídas em termos de fluxo de operações dentro do modelo de hardware;
- *Behavior Modelling* - Modelamento de comportamento é outro termo comum e basicamente é um modelo que define as relações entre entrada e saída;
- *Structural Modelling* - Modelamento de Estrutura é algo que é utilizado para instanciar componentes e a descrição do circuito. Pode ser a nível de porta ou a macro-nível, instanciando outros módulos no seu *design*. Em um *design* normalmente se encontra uma combinação de ambos, modelamento de estrutura e de comportamento;

A noção de síntese é de traduzir o código *HDL* para um circuito de tecnologia específica no qual se está trabalhando: um *FPGA*, uma tecnologia *ASIC*,... Seja qual for o dispositivo de destino, as bibliotecas que pertencem a este dispositivo normalmente são utilizadas no processo de tradução durante a operação de síntese.

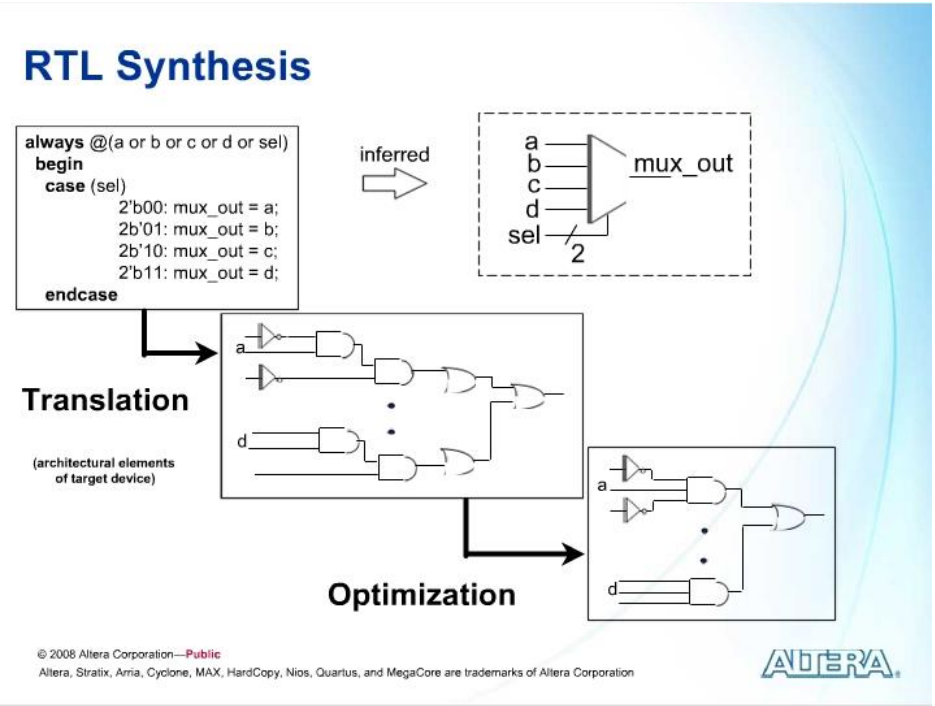
Módulo é a unidade fundamental de design no Verilog, e todo modelo em Verilog deve ser implementado dentro de um módulo.

Em um modelo comportamental, a funcionalidade é normalmente descrita, mas não é necessária uma idéia da estrutura atual do circuito, que é tipicamente a saída do sintetizador. Durante a execução do processo de síntese normal, o usuário irá especificar certos parâmetros de otimização para o sintetizador, que resultarão em uma estrutura particular do circuito; e em outra síntese, se os parâmetros de otimização forem diferentes, a saída do sintetizador pode ser diferente. Modelos comportamentais geralmente são utilizados tanto para síntese quanto para simulação. O código é feito de uma maneira comportamental porque não há nenhuma instanciação de elemento de arquitetura específico no dispositivo no qual o código será implementado, o código é tipicamente genérico. Isto possibilita que uma operação de compilação destine-se a um dispositivo, e outra compilação diferente pode ser destinada a outro dispositivo, sem modificar nenhuma parte do código.

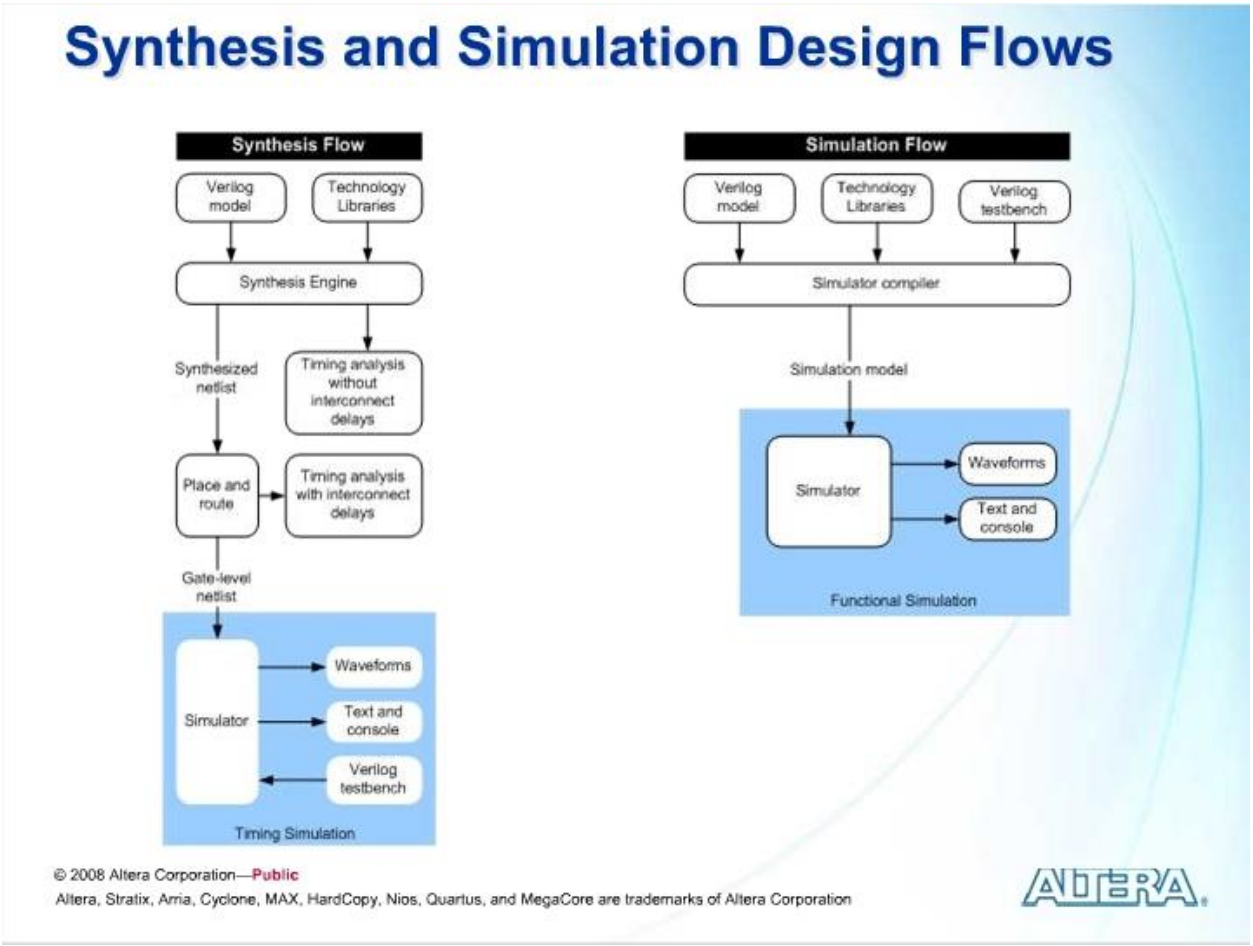
No modelo estrutural, tanto a funcionalidade como a estrutura do circuito são especificados, resultando nos elementos específicos de hardware que serão utilizados na implementação. O elementos de hardware podem ser genéricos como portas lógicas primitivas

(como uma porta E ou uma porta OU), ou uma instanciação de outro módulo que representa outra camada de abstração. É normalmente utilizado para síntese e/ou simulação, pode ser genérico, específico de um dispositivo ou ambos, o que significa que posso a descrição da estrutura inteira pode ser direcionada utilizando elementos primitivos da biblioteca que não são específicos de um dispositivo em particular, e ao mesmo tempo também é possível instanciar alguns elementos particulares da arquitetura no código, controlando a implementação do circuito. Em uma designação inicial em um design típico, encontra-se uma combinação dos modelos estruturais e comportamentais, em um modelo Verilog simples.

A síntese *RTL* é o primeiro dos processos de tradução do código original, utilizando elementos de arquitetura de um dispositivo particular que será utilizado. O sintetizador entra em um processo de otimização para ter a certeza de uma implementação ótima da descrição do circuito. No exemplo da figura abaixo há uma declaração *CASE* sendo implementada dentro de um bloco *always*, que tipicamente infere a implementação de um multiplexador.



O diagrama na figura a seguir demonstra um fluxo normal de design de síntese e simulação. No fluxo de síntese, observa-se um modelo Verilog como entrada, e suas bibliotecas de tecnologia também como entradas do engine de síntese. O sintetizador sintetiza seu *design* com o *netlist* sintetizado que pode de fato colocar um valor no dispositivo alvo da síntese. Nota-se neste fluxo de síntese que a saída do sintetizador pode também ser utilizada para gerar uma estimativa anterior de temporização, antes que fase de *place and route* seja executada no design.



O benefício disto pode ser uma idéia da performance, antes da entrada no processo de *place and route*. A saída da engine do *place and route* é tipicamente alimentada em um analisador de tempo estático, onde a análise detalhada de temporização é executada no circuito. A saída da engine de *place and route*, ou depois deste estágio, também pode produzir uma *netlist* em nível de portas que pode se introduzido em um simulador para executar uma simulação em nível de portas no design. Em um fluxo puro de simulação, o modelo Verilog, as bibliotecas tecnológicas e o Verilog *testbench* se encontram alimentando um simulador, como *ModelSim* ou VCS. O simulador compila o design com toda esta informação para gerar um modelo de simulação ou um *netlist* de simulação, que alimenta a engine de simulação. A *engine* de simulação pode então ser utilizada para executar a simulação de funções em seu código para o propósito de verificação.

Seção 2: Estrutura de um Módulo

Um módulo típico em Verilog está encapsulado entre duas palavras chave: "module" e "endmodule". Dentro de um módulo existem quatro declarações: declarações dos *ports*, declarações de tipo de dados, descrição da funcionalidade do circuito, i.e., comportamental ou estrutural, e finalmente a opção de incluir especificações de temporização.

- Verilog é uma linguagem case sensitive, significando que o mesmo nome de uma variável declarada em letras maiúsculas ou minúsculas, da perspectiva Verilog, consiste em duas variáveis diferentes;

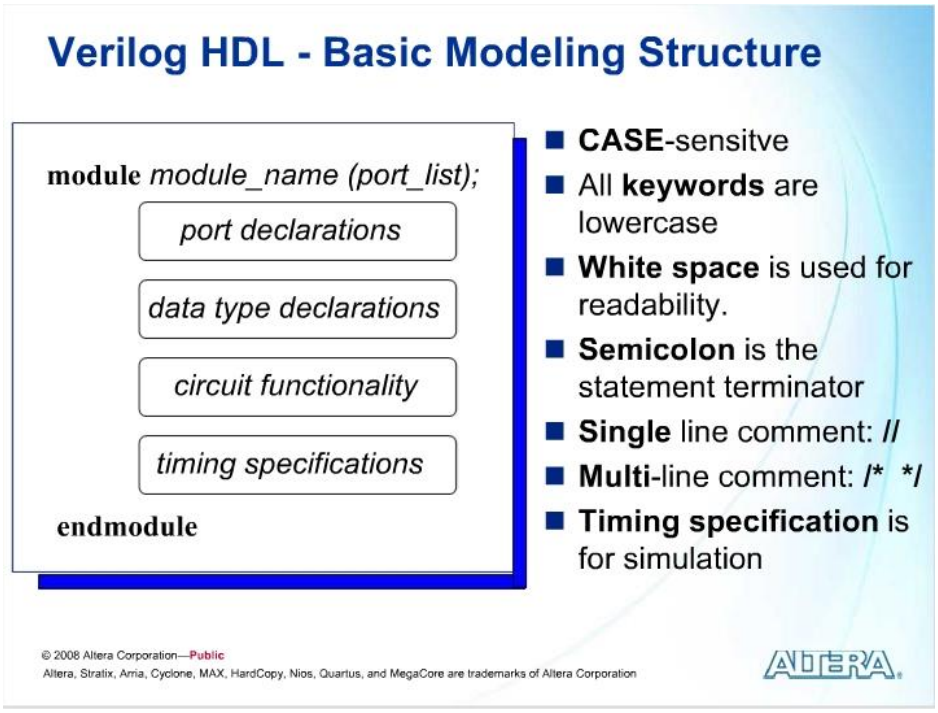
- Todas as palavras chaves devem ser em letras minúsculas no *design*;

- Espaços em branco são utilizados para melhor legibilidade do código, sendo recomendado o uso do traço baixo ("_");

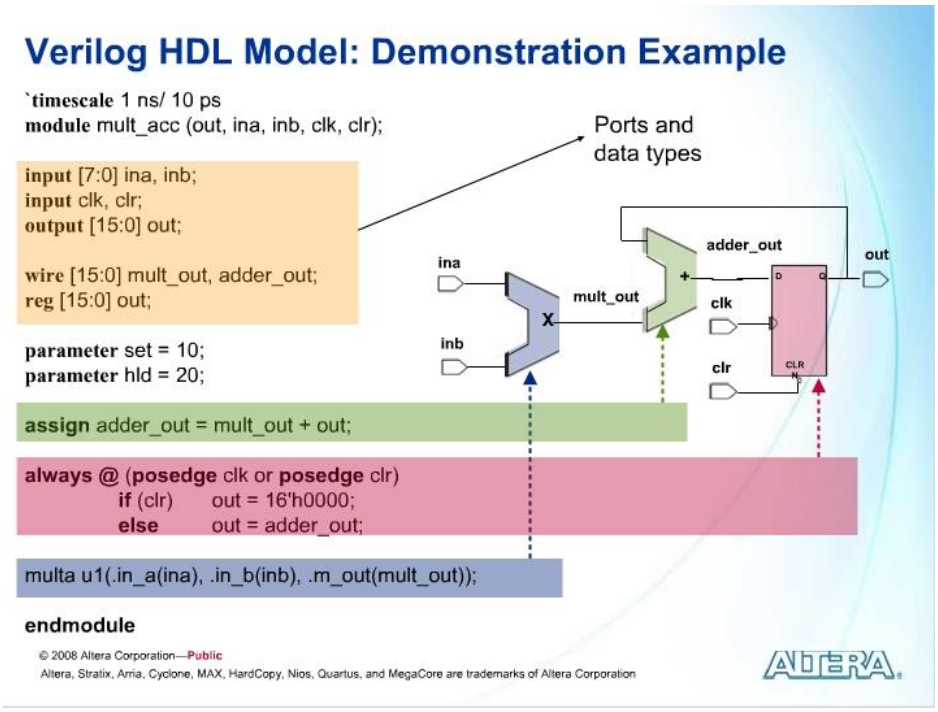
- O ponto e vírgula é a declaração de terminação, significando que se uma linha não for finalizada com o ponto e vírgula, o Verilog entende todos os caracteres como uma linha até encontrar o próximo ponto e vírgula;

- Comentários de uma linha são colocados após barras duplas "//" e comentários de mais de uma linha são iniciados por uma barra e um asterisco ("/*") e finalizados com um asterisco e uma barra ("*/");

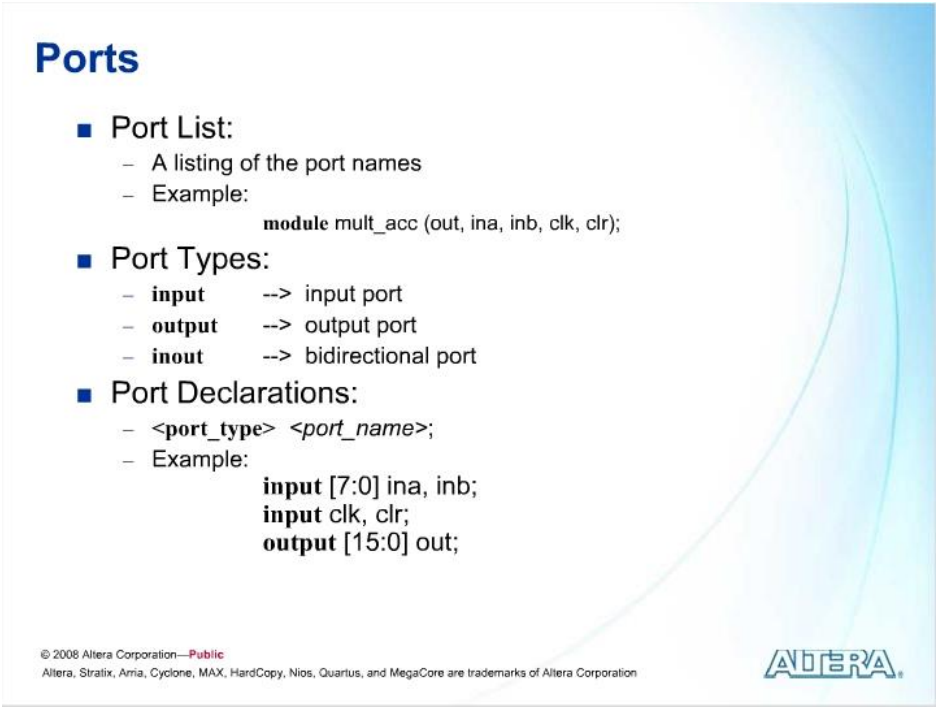
- A especificação de temporização é utilizada para propósitos de simulação somente. Quando há alguma especificação de temporização em um módulo, o sintetizador ignora completamente.



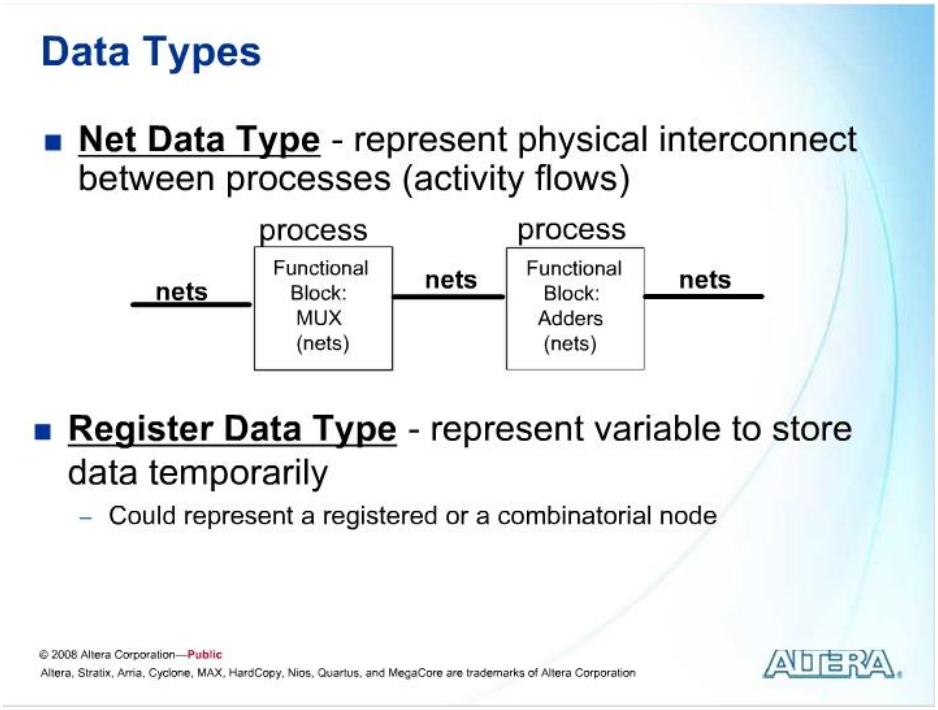
Abaixo se encontra um exemplo de um típico módulo Verilog. Neste módulo observa-se o início com um atributo que neste caso é uma escala de tempo que especifica a resolução para o simulador, utilizado somente para fins de simulação. Após encontra-se a palavra chave "module", especificando o nome do módulo, e entre parênteses os nomes dos ports de entrada e saída do módulo. Na primeira seção, os tipos de dados dos ports são definidos como conexões externas do módulo como entradas e saídas, definindo também os tipos de dados para estas entradas e saídas. A declaração de designação "assign" na caixa em verde é denominada de declaração de designação contínua, que resulta em um circuito combinacional (neste caso é sintetizada em um somador). No bloco rosa há o que denomina-se de bloco sequencial, que neste caso gera um flip-flop acionado pela borda positiva, e finalmente na seção azul há uma instanciação de um componente que foi declarado em algum outro módulo, componente este denominado "multa", com um nome da instanciação de "u1", juntamente com os nomes dos sinais que alimentam os ports de entrada e saída deste componente. A última declaração no módulo é a palavra chave "endmodule".



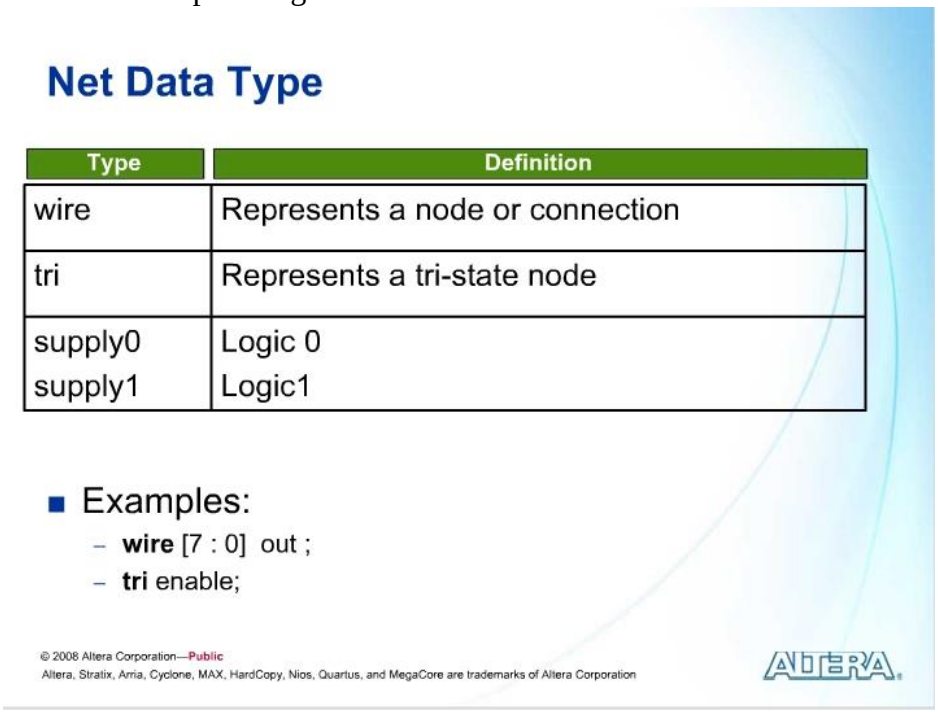
Ports são os meios primários de comunicação com um módulo. Existem três tipos fundamentais de *ports* em Verilog, denominados de *ports* de entrada (*input*), de saída (*output*) e de entrada-saída (*inout*), que possibilitam conexões bidirecionais para fora ou para dentro de um módulo. No exemplo exibido abaixo, ports "ina" e "inb" são ports de oito bits do tipo entrada, ports clock - "clk" e clear - "clr" são ports de entrada e o port "out" é um port de saída de dezesseis bits.



Existem dois tipos de dados fundamentais em Verilog denominados *net* e *register*. Tipos de dados *net* geralmente representam informação de interconexão e o tipo de dado *register* geralmente representa variáveis de armazenamento temporário, que pode ser sintetizado em um flip-flop no hardware (ou registrador de hardware), ou em um nó combinatório.

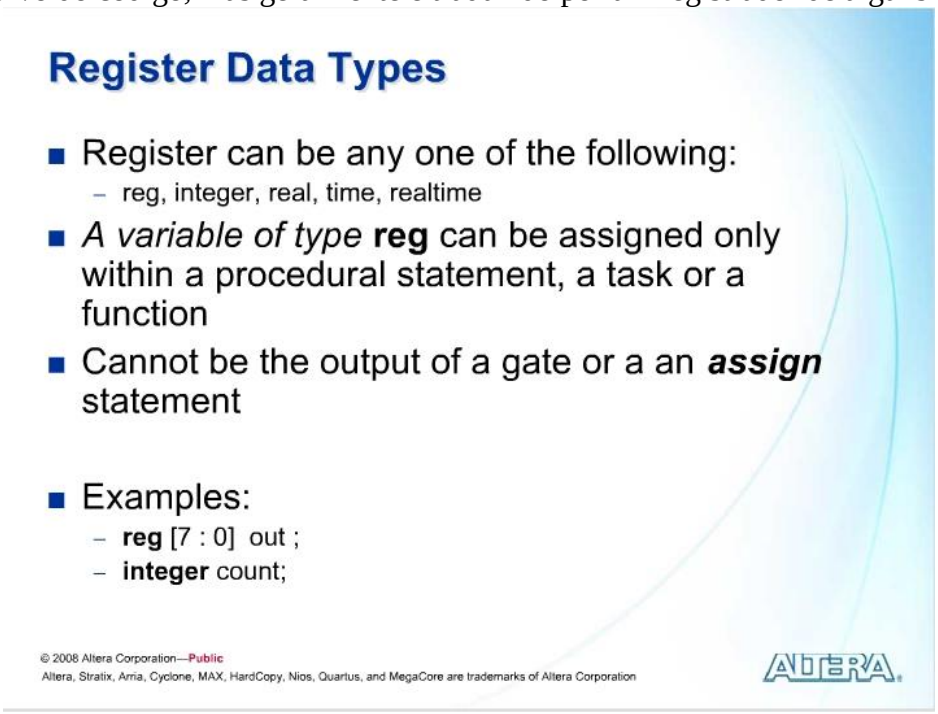


Como mencionado anteriormente, tipos de dados *net* representam informação de interconexão. O slide abaixo exibe os tipos mais comuns de dados *net* que são utilizados em módulos Verilog, denominados tipos de dados "*wire*", que representam um nó ou uma conexão. O tipo de dado "*tri*" representa um nó "*tristate*" e fornece uma fonte para *hardcode* de nível lógico zero ou um. Por exemplo, o *bus* "*out*" é um *bus* de oito bits do tipo "*wire*" e o sinal enable é um sinal *tristate* no exemplo a seguir.

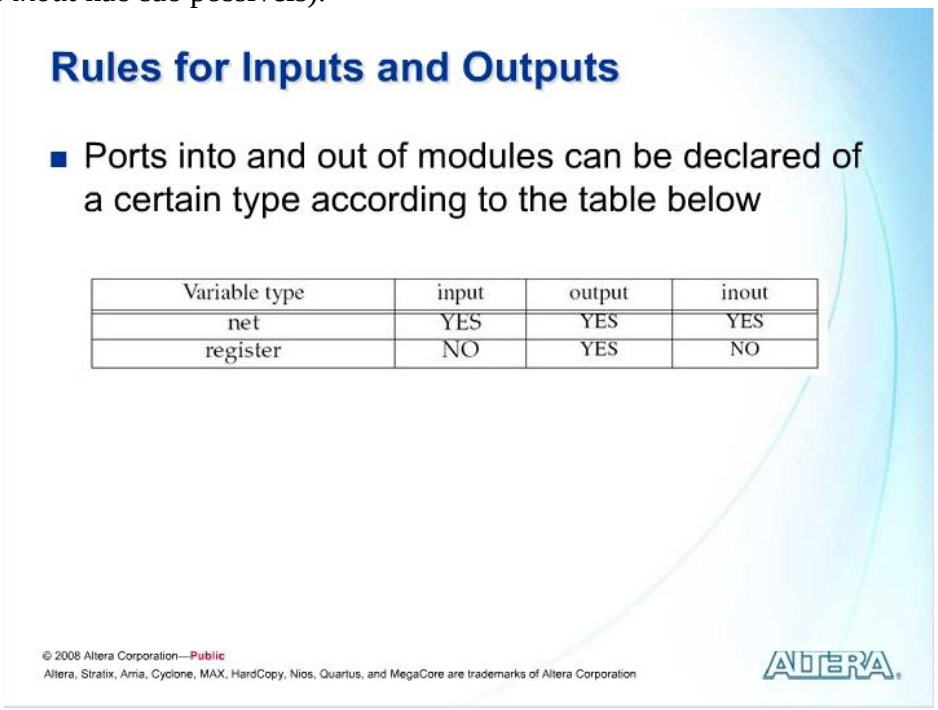


Variáveis do tipo de dados *register* podem ser sintetizadas tanto em nós combinacionais ou sequenciais, ou são sintetizadadas como uma função dos contatos e tipos de declarações nas quais são utilizadas. Um tipo de dado *register* pode ser um dos seguintes tipos: *register*, *integer*, real, time e realtime. Uma variável do tipo *reg* pode ser designada somente por um procedimento de declaração, uma tarefa ou uma função. Uma variável do tipo *reg* também não pode ser a saída de uma porta ou de uma declaração de designação (são regras do Verilog). O exemplo abaixo

exibe uma variável de oito bits denominada "out" do tipo *reg*, e também uma variável denominada "count "do tipo *integer*. O significado de um inteiro (*integer*) é dependente do dispositivo alvo do código, mas geralmente é traduzido por um registrador de alguns bits.

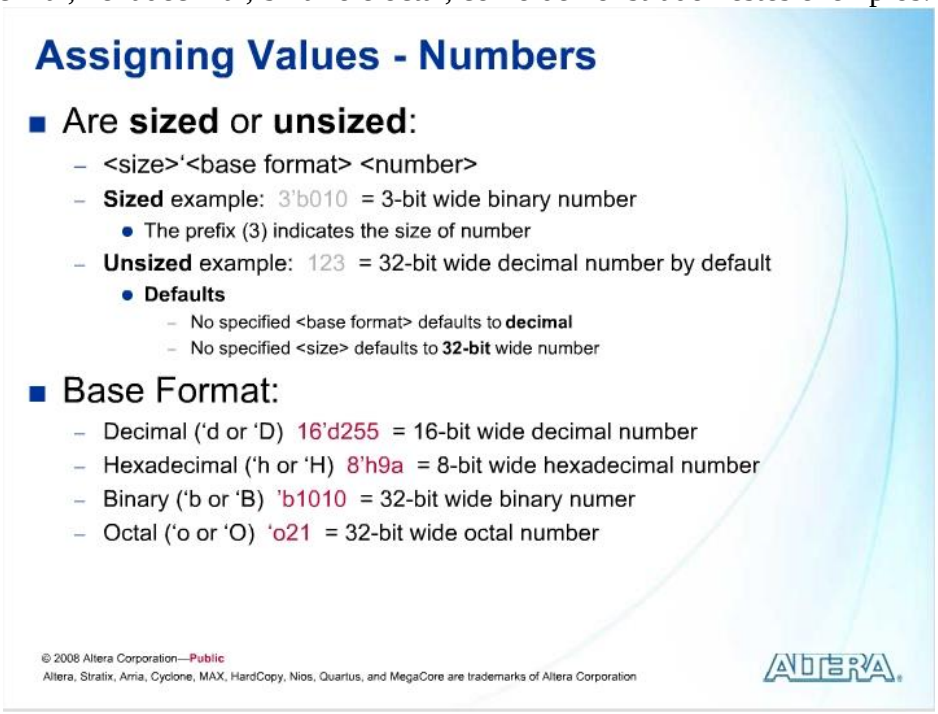


Conexões para dentro ou para fora de um módulo possuem regras básicas, de acordo com a tabela a seguir. Se você possui uma variável do tipo *net*, ela pode ser declarada do tipo *input*, *output*, ou *inout*. Uma variável do tipo *register* somente pode ser declarada do tipo *output* (os tipos *input* e *inout* não são possíveis).

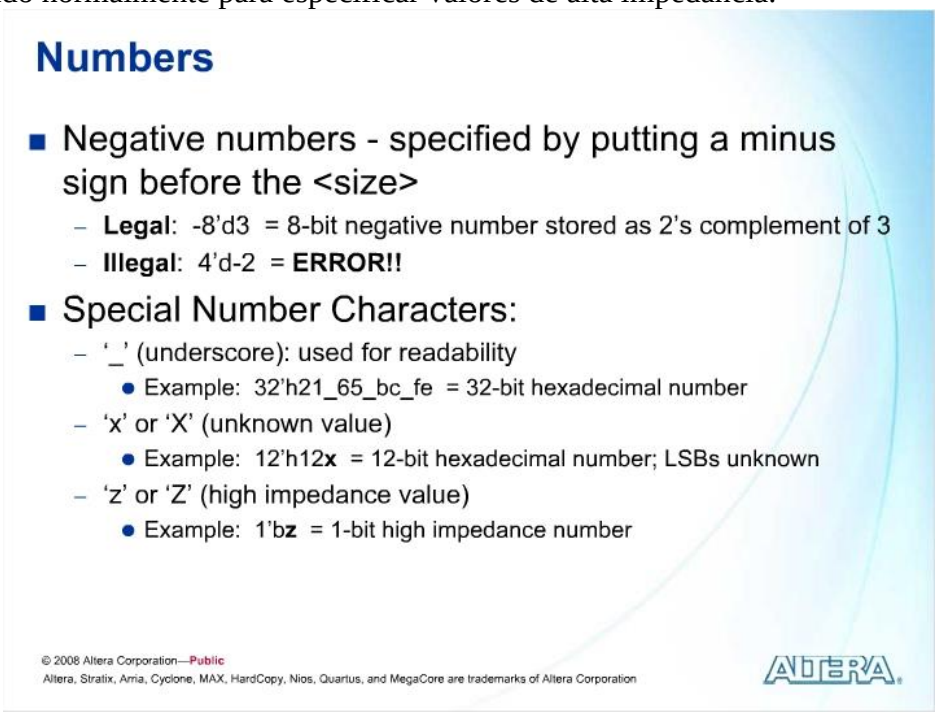


Valores designados para as variáveis podem ser dimensionados ou não dimensionados. Se o tamanho de uma variável não for especificada, ela será traduzida em um tamanho de 32 bits decimais por padrão. Para especificar o tamanho de uma variável ou o tamanho de um valor que é atribuído a uma variável, o formato é exibido aqui. O primeiro exemplo demonstra 3'b010, que diz ao compilador que esta entidade é uma entidade de três bits, o formato da informação está em

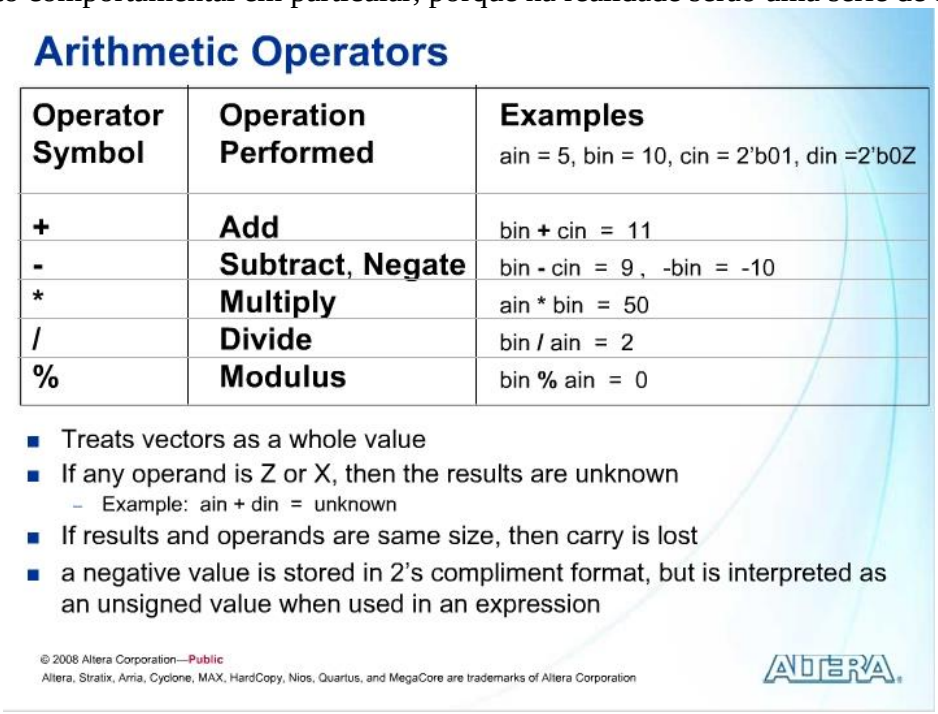
binário e o valor atribuído à variável é "010". É possível especificar o radix da informação que é atribuída a uma variável, para melhorar a legibilidade do código. Podem ser especificados os formatos decimal, hexadecimal, binário e octal, como demonstrado nestes exemplos:



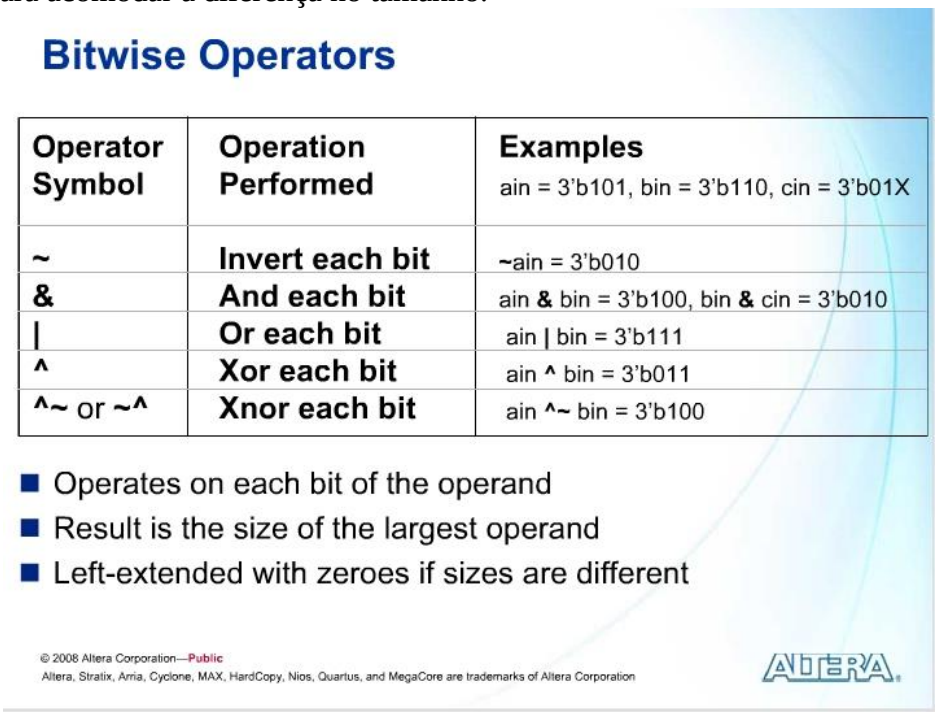
Números negativos em Verilog são representados precedendo o tamanho do valor com um sinal negativo. Por exemplo, -8'd3 representa um número negativo armazenado em 8 bits como o complemento de 2 do número 3. O segundo exemplo 4'd-2 resulta em um erro de sintaxe porque o sinal negativo é colocado depois da definição do tamanho. Existem alguns caracteres especiais que são utilizados no Verilog, incluindo o underscore ("_"), que é utilizado normalmente para finalidade de legibilidade do código, "x" minúsculo ou "X" maiúsculo, utilizado normalmente para especificar valor desconhecido, e "z" minúsculo ou "Z" maiúsculo, que é utilizado normalmente para especificar valores de alta impedância.



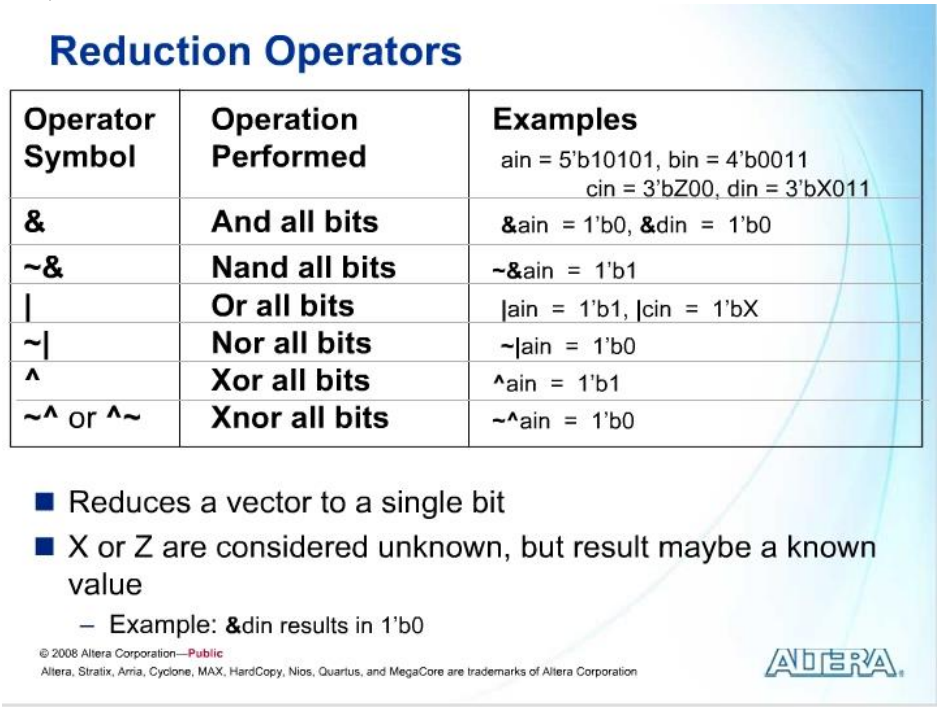
Em termos de operadores aritméticos, o Verilog define soma, subtração, multiplicação e divisão. A linguagem Verilog trata o operando nestas operações como um valor total. Se qualquer um destes operandos é "z" ou "x" o resultado será desconhecido, se o resultado e os operandos são do mesmo tamanho, o sinal de *carry* (vai um) será perdido (deve-se ter certeza de que a saída deve ter um tamanho maior que os operandos). Todos os valores negativos são armazenados no formato de complemento de 2, e deve-se ter esta informação em mente em um modelamento comportamental em particular, porque na realidade serão uma série de bits.



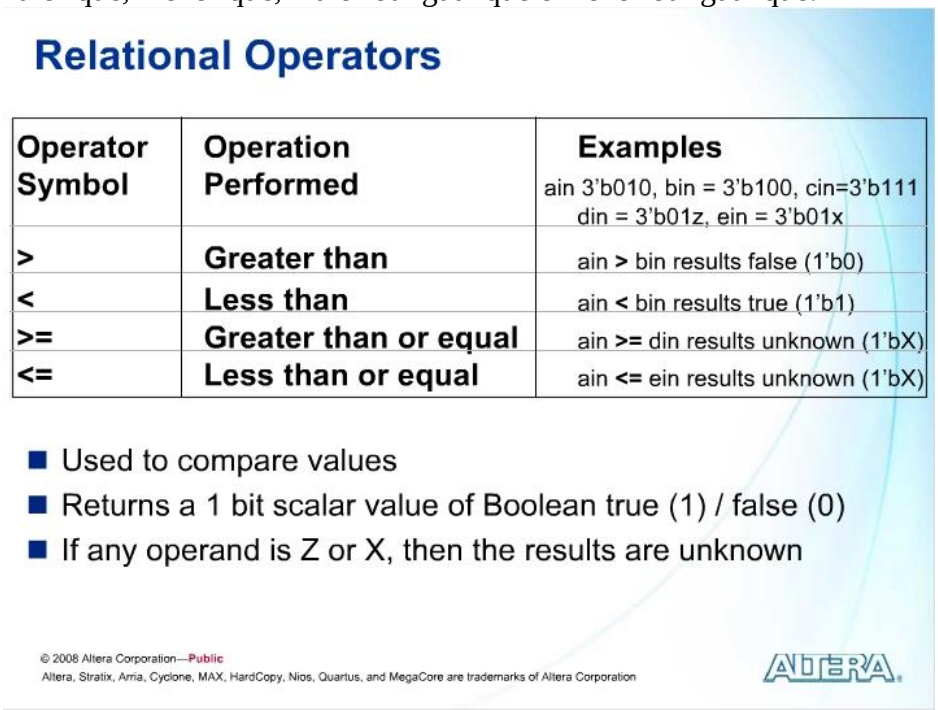
Operadores bit a bit podem ser operadores unários ou binários. Verilog define os operadores *Invert*, *And*, *Or*, *Xor* e *Xnor*. Estes operadores atuam em cada bit do operando, e normalmente o resultado é do mesmo tamanho que o maior operando, e na ocorrência de uma má combinação no tamanho, usualmente o operando de menor tamanho é preenchido à esquerda com zeros para acomodar a diferença no tamanho.



Operadores de redução são operadores unários que reduzem um vetor a um único bit. Verilog define operadores de redução *And*, *Nand*, *Or*, *Nor*, *Xor* e *Xnor*. Valores "x" ou "z" são considerados desconhecidos, mas o resultado pode ser um valor conhecido. No exemplo abaixo, se din=3'bX011, &din resulta em 1'b0.

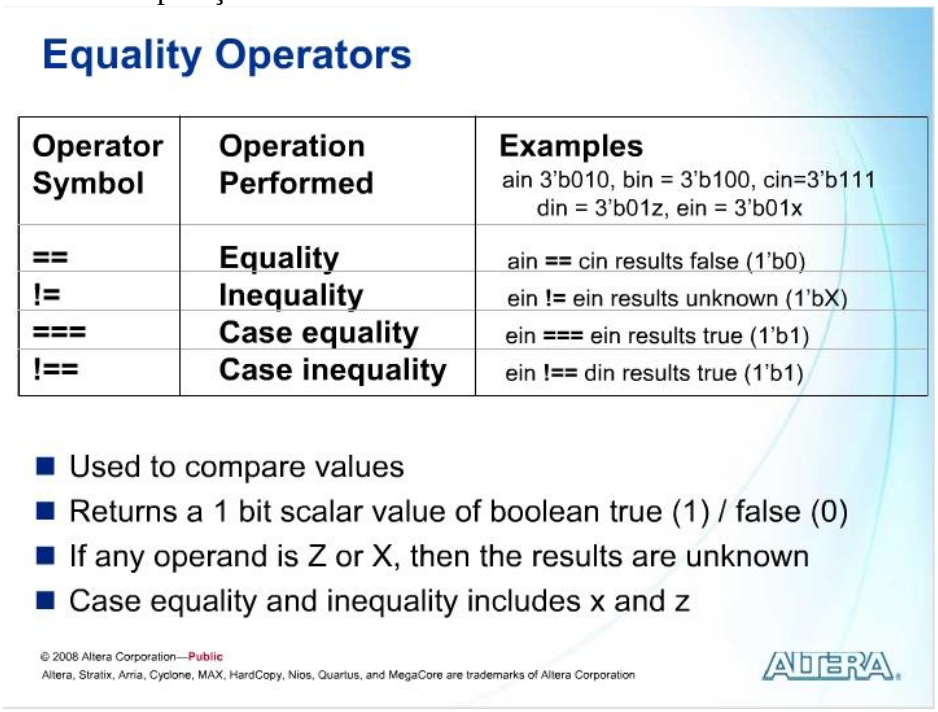


Operadores relacionais são utilizados para comparar valores. Estes operadores fornecem um valor de um bit, que representa usualmente um valor falso ou verdadeiro. Se qualquer um dos operadores for "z" ou "x", o resultado é desconhecido. A linguagem Verilog define os quatro operadores maior que, menor que, maior ou igual que e menor ou igual que.

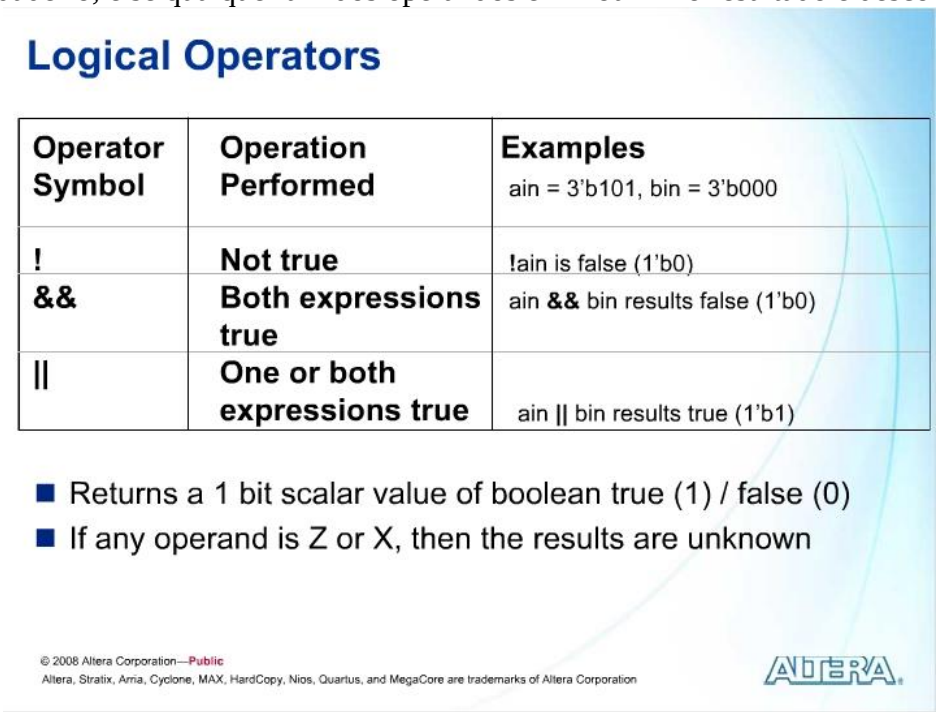


Operadores de igualdade são utilizados para comparar valores, e normalmente retornam um valor escalar de um bit que representa falso ou verdadeiro. Em Verilog existem quatro tipos diferentes de operadores de igualdade que podem ser utilizados: igualdade, desigualdade, caso igualdade e caso desigualdade; que literalmente comparam bit a bit os operandos para decidir se

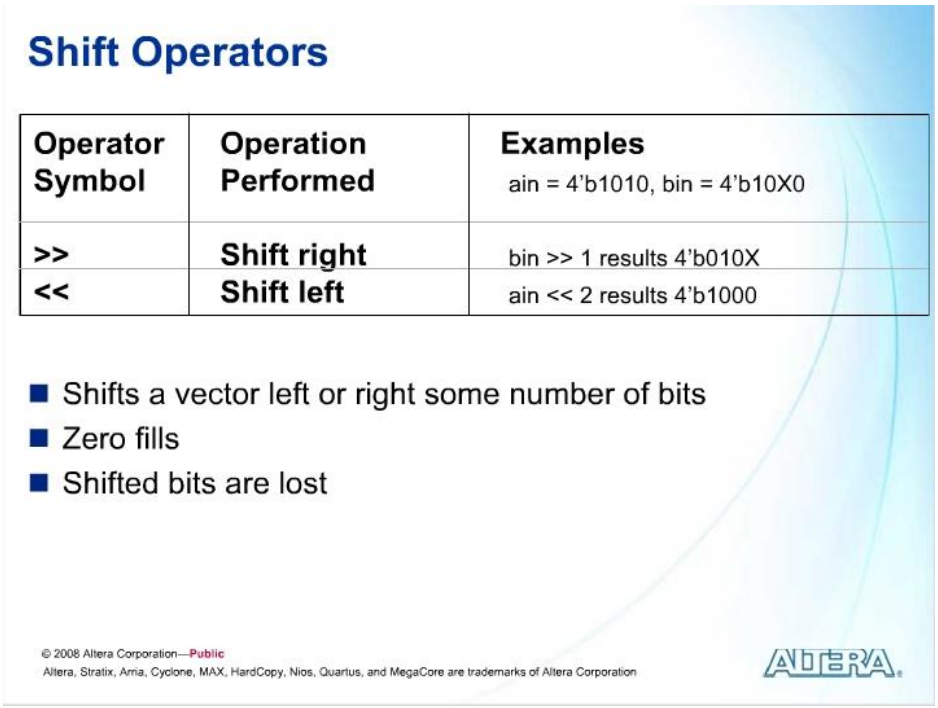
a saída será 1 ou 0. No caso de se utilizar os operadores de igualdade e desigualdade, se qualquer um dos operandos é "z" ou "x", o resultado será desconhecido. No caso dos operadores de caso igualdade e caso desigualdade, os valores "z" e "x" são incluídos como parte de definição de tipos, sendo aceitos na operação.



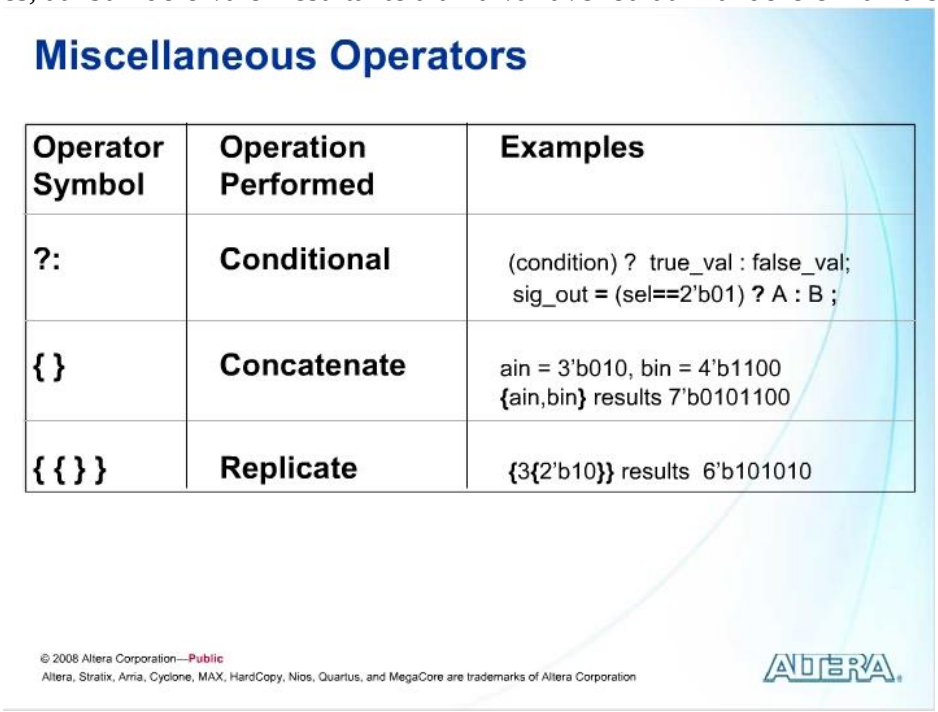
Os operadores lógicos em Verilog podem ser unários ou binários. A linguagem Verilog define o operador "NÃO", um operador unário, um operador "E" binário e um operador "OU" binário. Quando estes operadores são utilizados, retornam um valor de um bit que representa falso ou verdadeiro, e se qualquer um dos operandos é "z" ou "x" o resultado é desconhecido.



Operadores de deslocamento são utilizados para deslocar um vetor para a esquerda ou para a direita. Os bits deslocados são normalmente perdidos, e as posições anteriores destes bits são preenchidas com zeros.

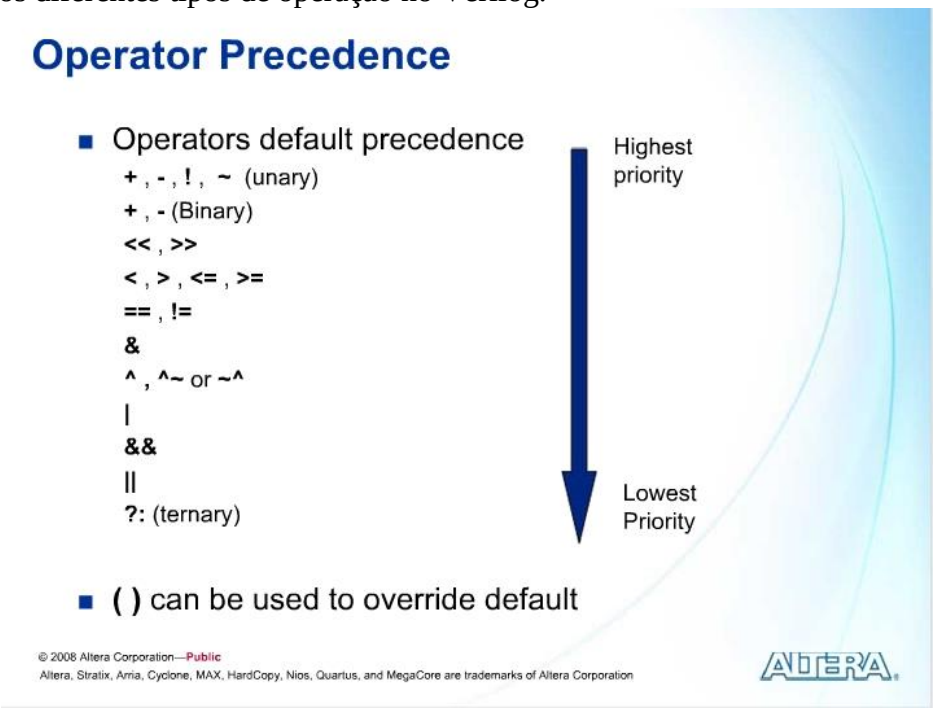


Um dos operadores mistos mais utilizados em Verilog são o operador condicional, que consiste em uma maneira conveniente de representar uma versão compacta da declaração if...then...else. No exemplo sig_out = (sel = = 2'b01)?A:B, o sinal sel é comparado com o valor "01", e se for verdadeira a igualdade o valor "A" é atribuído à variável "sig_out", caso contrário o valor "B" é atribuído. A operação de concatenação é outra operação muito utilizada em Verilog, para basicamente combinar dois vetores para produzir um vetor maior, resultante da concatenação dos vetores operandos. A operação de replicação pode replicar um mesmo vetor diversas vezes, atribuindo o valor resultante a uma variável ou utilizando-o em uma expressão.



É muito recomendável, ao utilizar diversos operadores em uma mesma linha de código, utilizar parênteses para a precedência nas operações com os operandos, obtendo-se assim mais clareza no código, exibindo-se exatamente as operações na ordem em que serão executadas. Em

casos em que isto não é possível, a lista abaixo demonstra como o compilador ordena a prioridade dos diferentes tipos de operação no Verilog.

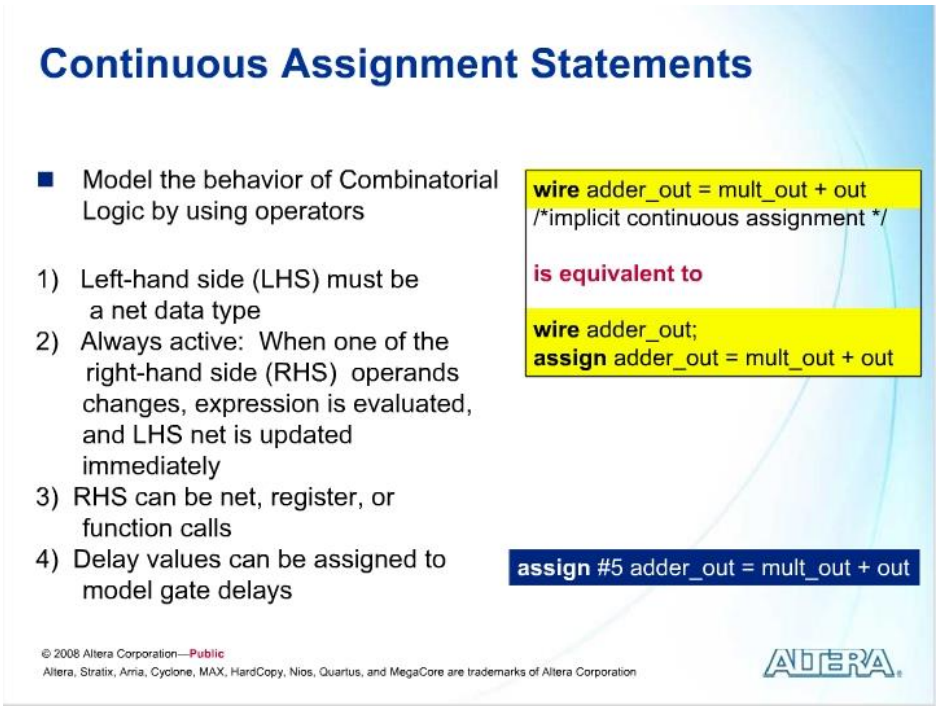


Executando Designações:

Em Verilog existem duas maneiras fundamentais de se efetuar as designações: declarações de designação contínuas e declarações de designação de procedimento. Declarações de designação contínuas são utilizadas normalmente para gerar circuitos combinacionais e pode ser efetuada durante o estágio de declaração de variáveis, como no exemplo a seguir em que a variável "adder_out", do tipo *wire*, é igual à "mult_out + out". O exemplo abaixo produz uma designação de um circuito combinacional. No corpo do módulo pode ser utilizada a palavra chave "*assign*" para criar uma declaração de designação continua. Isto significa que a qualquer momento em que qualquer uma das variáveis "mult_out" ou "out" modificar seu valor, a variável "adder_out" será atualizada para seu novo valor. Verilog possibilita também definir valores de atraso quando a designação é executada. No exemplo:

assign #5 adder_out = mult_out + out

significa que a variável "adder_out" modificará seu valor 5 unidades de tempo depois que "mult_out" ou "out" modificarem seu valor. Se o simulador estiver sendo executado e a variável "out", por exemplo, se modifica em t=1, a variável "adder_out" se modificará em t=6. As declarações de designação continua seguem as seguintes regras:



Declaração de designação de procedimento são designações efetuadas dentro de um bloco de designação de procedimento. Em Verilog, existem dois tipos de blocos de designação de procedimento, denominados bloco "*initial*" e bloco "*always*". Um bloco *initial* é utilizado normalmente para inicializar declarações de comportamento somente para fins de simulação. Então, qualquer que seja o código incluso dentro de um bloco *initial*, é ignorado pelo sintetizador. Um bloco *always* é utilizado normalmente para descrever a funcionalidade de um circuito utilizando declarações de comportamento. Cada bloco *always* e *initial* representa um processo separado, e em um circuito normalmente se encontram muitos blocos *always*. Na simulação, todos estes blocos *always* estarão sendo executados concorrentemente ao mesmo tempo. Estes processos são executados em paralelo e iniciam no tempo de simulação zero. Depois do início da sessão de simulação, estes processos estão imediatamente ativos e a qualquer tempo qualquer uma das entradas destes blocos executarão o código dentro do bloco.

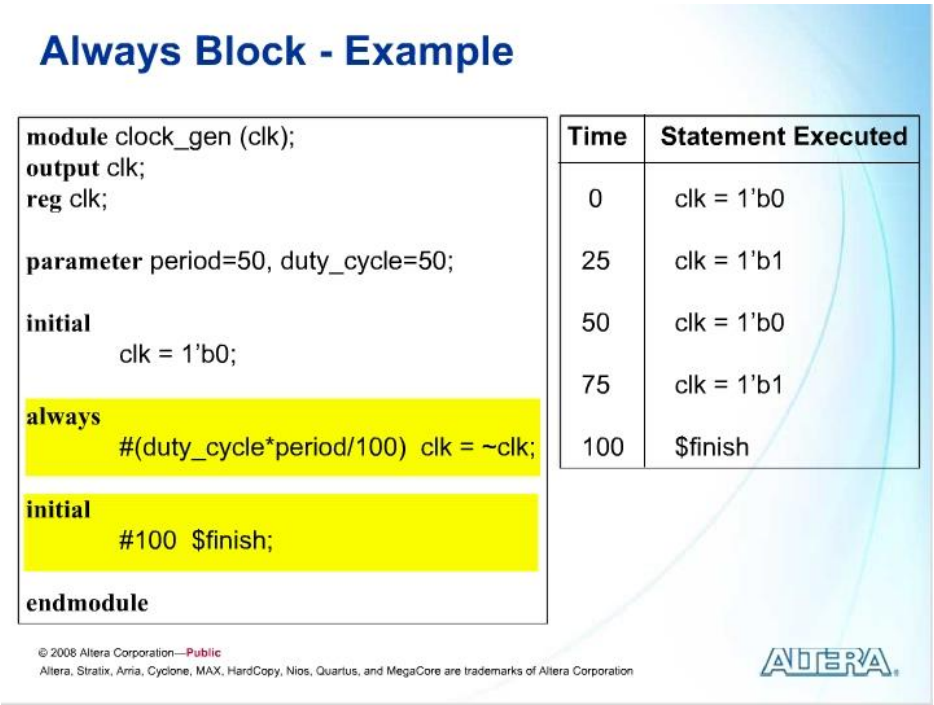
As declarações dentro de um processo são executadas seqüencialmente. Então as declarações dentro de um bloco *always*, seja qual forem, serão executadas na ordem em que estão. Blocos *always* e *initial* não podem ser aninhados, não sendo possível um bloco *always* dentro de outro bloco *always* ou de um bloco *initial*.

Um bloco *initial* consiste de declarações de comportamento que tipicamente incluem declarações de monitoramento, geração de formas de onda, e qualquer processo ou seqüência de inicialização que deve ser executado pelo menos uma vez durante a sessão de simulação. Um bloco *initial* inicia sua execução em t=0, é executado do início ao fim e então cessa sua execução, ou seja, é executado somente uma vez. Todas as declarações de comportamento dentro de um bloco *initial* são executadas seqüencialmente, sendo que a ordem de posicionamento destas declarações dentro do bloco deve ser levada em consideração. Observa-se que as declarações dentro de um bloco *initial* são completamente ignoradas pelo sintetizador, pois o bloco *initial* é definido na linguagem para propósitos de simulação e nunca é utilizado para síntese.

Como nos blocos *initial*, o bloco *always* consiste de declarações de comportamento sendo sempre executados, i.e., não são executados somente uma vez ou param sua execução durante a sessão de simulação. Se existem múltiplos blocos *always* no módulo, o que é comum, todos estes blocos serão executados concorrentemente ao mesmo tempo, porém, declarações dentro destes blocos serão executadas seqüencialmente. Blocos *always* são utilizados para modelar um processo que é continuamente repetido em um circuito digital, um exemplo poderia ser um flip-

flop ou registrador que é continuamente acionado por um sinal de clock. Um bloco *always* se inicia no tempo igual a zero, e executa as declarações comportamentais continuamente na forma de um loop. É possível obter uma lista de sensibilidade em um bloco *always*, o que significa que a execução em um bloco *always* não inicia a menos que uma ou mais das variáveis na lista de sensibilidade modifique seu valor. E como no bloco *initial*, as declarações comportamentais dentro de um bloco *always* executam-se sequencialmente, então a ordem das declarações dentro do bloco é levada em conta.

Abaixo se encontra um exemplo da implementação de um bloco *always*, com um módulo Verilog denominado "clock_gen" com uma saída simples denominada "clk" do tipo *reg*. O primeiro bloco *initial* inicializa a variável "clk" com o valor zero, e então o bloco *always* executa a declaração de que cada ciclo de trabalho é multiplicado pelo período e dividido por 100. Sempre que o simulador ultrapassar este tempo dado por este código, o sinal de clock será alternado porque este valor é igual ao clock negado. Há outro bloco *initial*, que diz que após 100 unidades de tempo é executada uma chamada ao sistema dada por "*finish*" para finalizar a sessão de simulação. Neste exemplo o período é de 50 e o ciclo de trabalho de 50, o que se traduz em um período de clock efetivo de 25 unidades de tempo. Se esta simulação for executada, o sinal de clock alternará seus valores a cada 25 unidades de tempo e em t=100 a simulação será finalizada.



Dentro de um bloco de designação de procedimento existem dois tipos de declarações de designação que podem ser utilizados para propósitos de modelamento: designação de bloqueamento e designação de não-bloqueamento. As designações de bloqueamento são executadas na ordem em que são especificadas em um bloco seqüencial, o que significa que se existem duas designações de bloqueamento consecutivas, a segunda designação não pode ser executada até que a primeira tenha finalizado sua execução. Designações de não-bloqueamento possibilitam o agendamento de designações sem bloquear a execução de declarações que se seguem em um bloco seqüencial. Se existem duas designações de não-bloqueamento consecutivas, a segunda designação pode ser executada ao mesmo tempo em que a primeira é executada. E novamente estes tipos de designações de bloqueamento e não-bloqueamento podem estar dentro de um mesmo bloco de procedimento. Os valores ou os tipos de valores que são utilizados dentro de um bloco seqüencial devem ser do tipo *reg*, que pode incluir variáveis do tipo definido como *reg*, *integer*, real, time ou realtime.

Two types of Procedural Assignments

- Blocking Assignment (=) : executed in the order they are specified in a sequential block
- Nonblocking Assignment (<=) : allow scheduling of assignments without blocking execution of the statements that follow in a sequential block

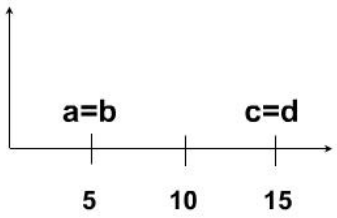
- Reside inside of procedural blocks
- Update values of **reg**, **integer**, **real**, **time, or realtime** variables **(i.e. Left Hand side type)**

O slide da figura abaixo demonstra alguns exemplos de uso das designações de bloqueamento e não-bloqueamento. O exemplo à esquerda demonstra um bloco *initial* com duas designações, "a=b" e "c=d". Em t=5, o simulador traduz que "b" é atribuída a "a", e em t=10, "d" será atribuída a "c". Como está sendo utilizada uma designação de bloqueamento, "d" atribuída a "c" não pode ser executada até que "b" seja atribuída a "a", que somente ocorre quando t=5. Então em t=5 a designação de "a=b" é executada, e 10 unidades de tempo depois, a designação "c=d" é executada. No exemplo de não-bloqueamento, temos um bloco *initial* que executa as mesmas designações no exemplo do lado esquerdo, exceto pela utilização de designações de não-bloqueamento. Como podemos observar claramente deste exemplo, caso seja utilizada designação de não-bloqueamento, ambas declarações podem ser executadas ao mesmo tempo, e então a designação "c <= d" não necessita esperar até que a designação "a <= b" seja executada. Então a designação "c <= d" é executada em t=10, e não em t=15.

Blocking vs. Nonblocking Assignments

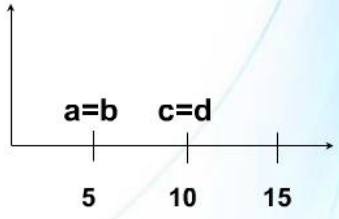
Blocking (=)

```
initial
  begin
      #5     a = b;
      #10    c = d;
  end
```

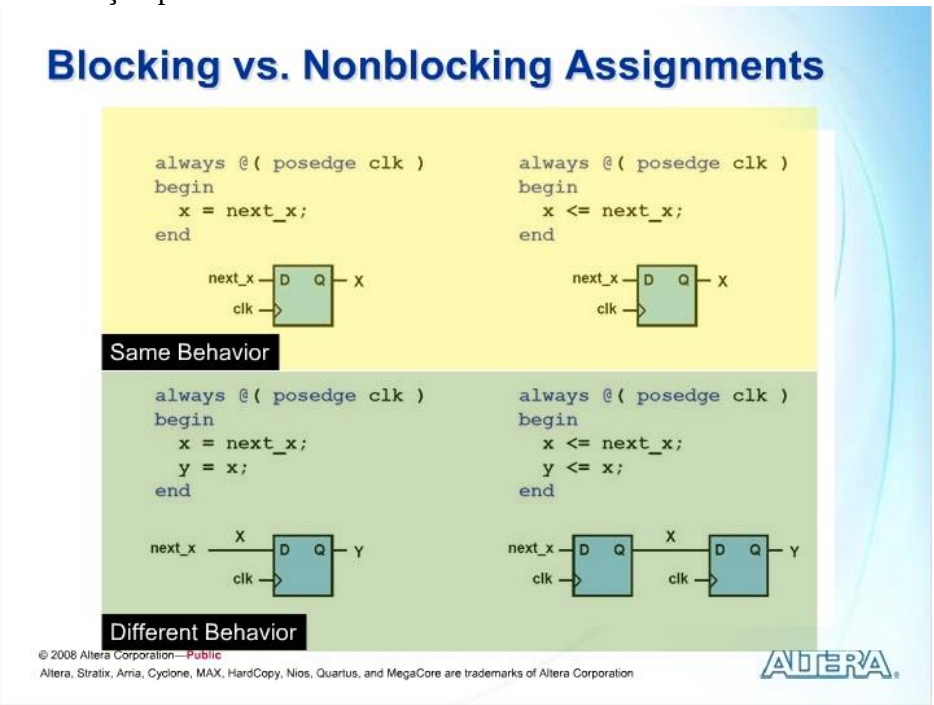


Nonblocking (<=)

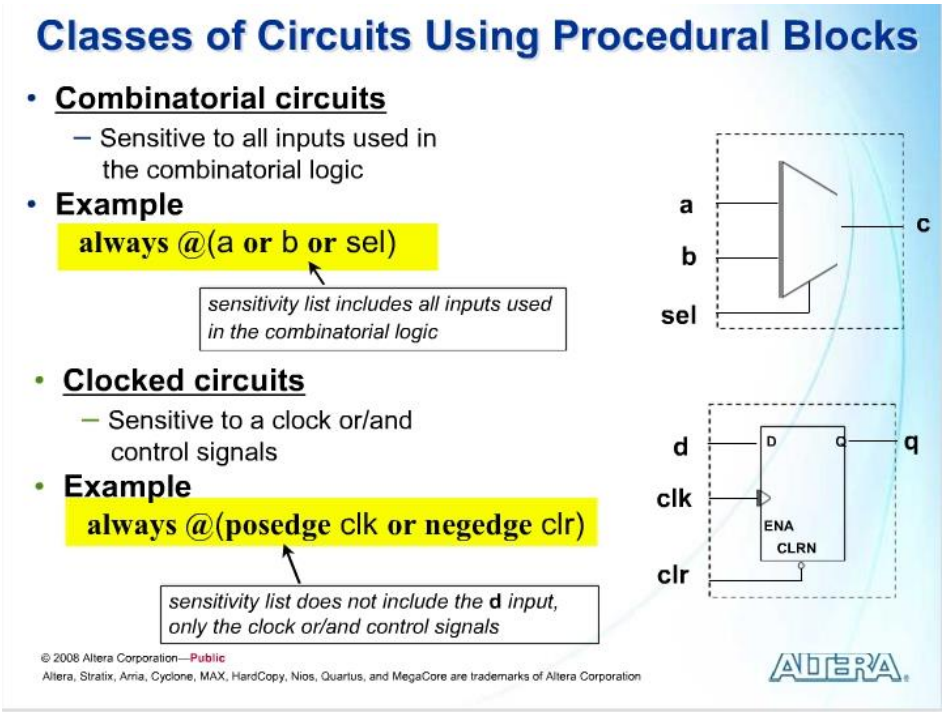
```
initial
  begin
      #5     a <= b;
      #10    c <= d;
  end
```



Abaixo encontra-se outro exemplo de comparação entre as designações de bloqueamento e de não-bloqueamento, o que pode resultar em uma diferença significativa no comportamento do circuito. O uso cuidadoso de designações de bloqueamento e de não bloqueamento dever ser levado em consideração para sintetizar o circuito corretamente.

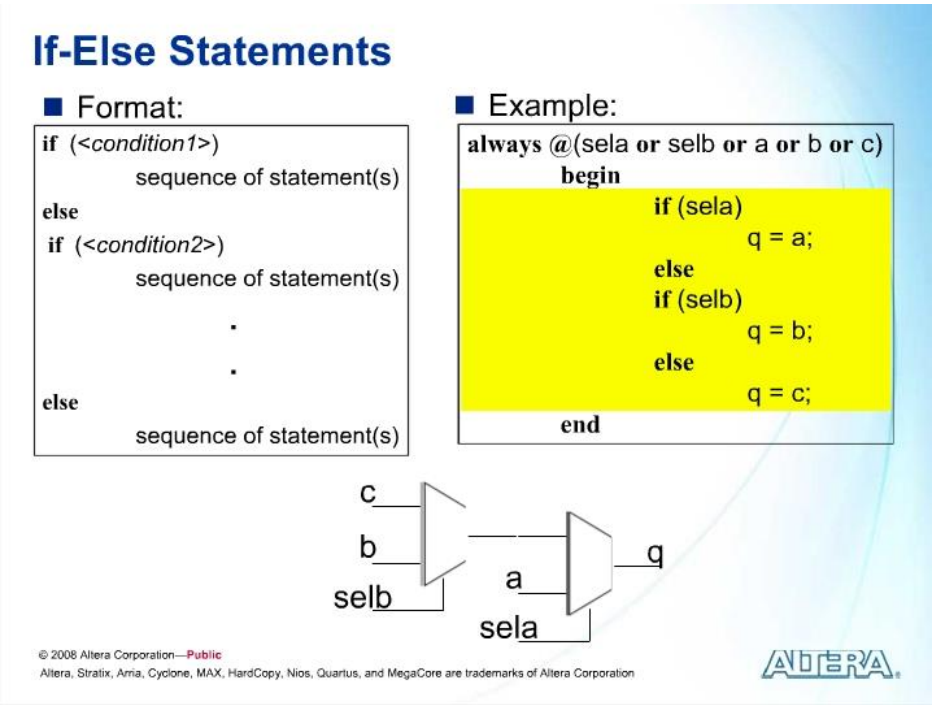


O bloco de procedimento *always* pode ser utlizado por circuitos combinacionais e circuitos com clock. O tipo de circuito sintetizado é tipicamente uma função do tipo de uma função primitiva que é utilizada dentro de um bloco *always*. Por exemplo, se na lista de sensibilidade há somente variáveis, isto pode resultar em um circuito combinacional pelo sintetizador. Se no bloco *always* existem as funções pré-construídas "posedge" ou "negdge" clock, isto pode resultar na síntese de um circuito com clock ou em um circuito sequencial. Nestes tipos de circuitos não é necessário especificar a entrada do elemento de registro na lista de sensibilidade, porque assume-se que quando o clock muda o dado já está estável na entrada no flip-flop.

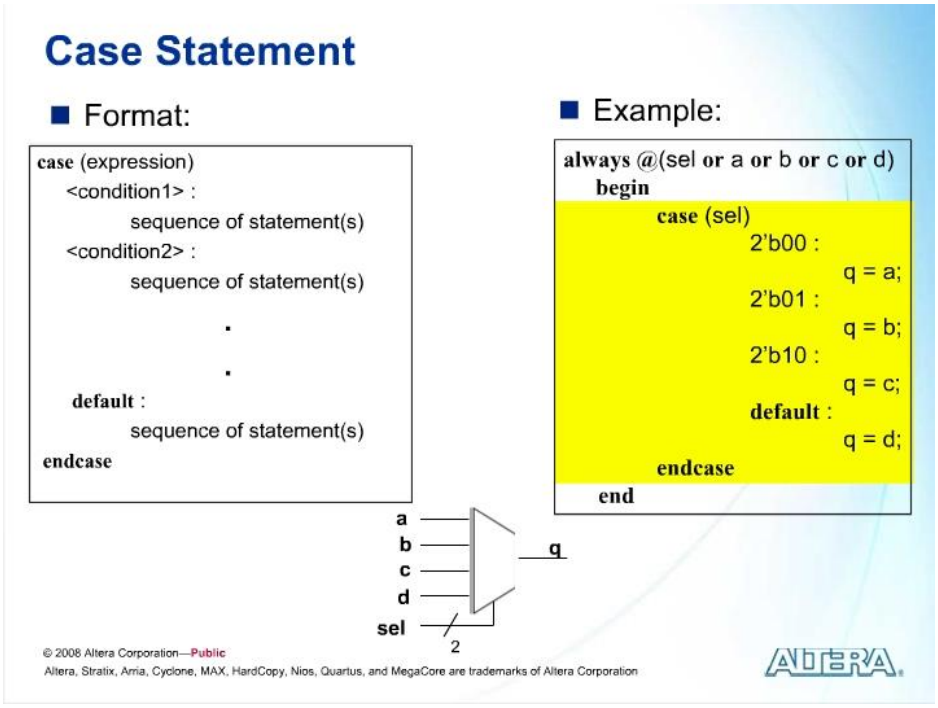


Declarações de comportamento são declarações que podem ser utilizadas dentro de um bloco de designação de procedimento, i.e., um bloco *always* ou *initial*. A linguagem Verilog define três tipos diferentes de delarações de comportamento que podem ser utilizadas, denominadas declarações "if-else", declarações "case" e declarações "loop".

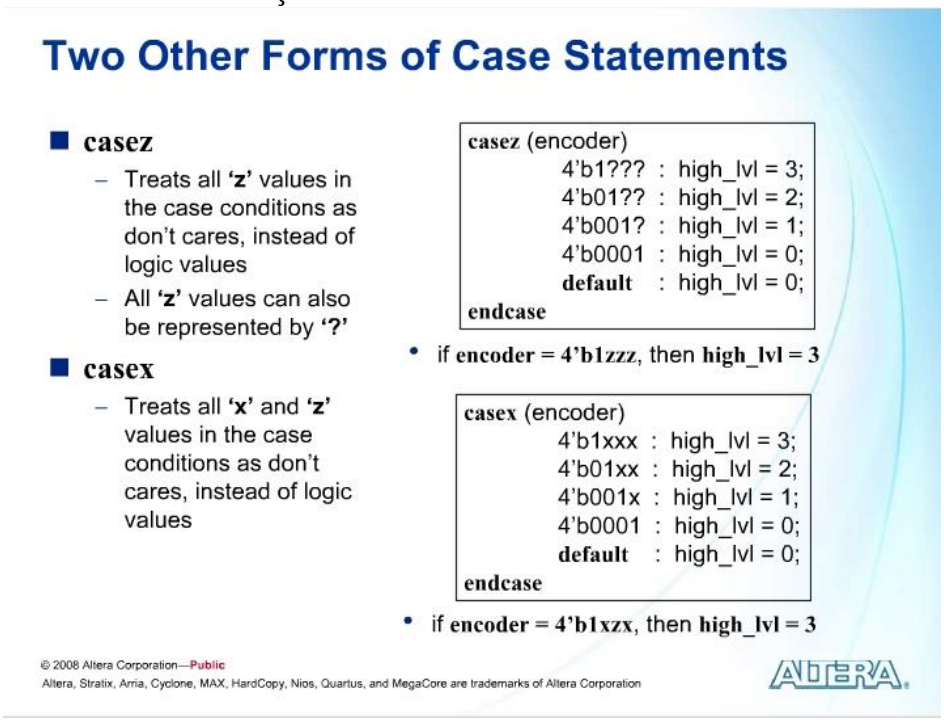
O formato das declarações if-else é demonstrado aqui. É possível obter declarações if-else aninhadas quantas vezes for necessário. O único fato a se prestar atenção é que no caso do uso de declarações if-else aninhadas teremos um circuito que é sintetizado utilizando uma implementação de prioridade de três tipos, que resulta em muitas camada de lógica. No exemplo à direita, a lista de sensibilidade inclui todas as entradas no circuito combinacional, assim como todas as variáveis "select", neste caso "a", "b" e "c", "sela" ou "selb". Isto significa que este bloco *always* será chamado ou executado a qualquer tempo em que qualquer uma destas variáveis modificar seu valor.



A declaração case é uma maneira conveniente de implementar declarações if-then-else ou comportamento if-then-else complexos. A diferença entre uma declaração case e uma if-else é o fato de que todas as declarações no case são executadas ao mesmo tempo. A saída típica do sintetizador da declaração case é um multiplexador. No exemplo abaixo e à direita, observa-se que a lista de sensibilidade especifica novamente todas as entradas, o que inclui as variáveis "a", "b", "c" e "d", e as variáveis de seleção do multiplexador, e então a declaração case é executada. Neste exemplo, se a variável de seleção é igual a "00", "a" é atribuído para "q", senão observa-se a próxima declaração, se a variável de seleção é igual a "01", "b" é designado a "q", se é igual a "10", "c" é designado a "q", e há uma declaração padrão para o caso das anteriores falharem, no caso designando "d" para "q". Assim, a declaração case é finalizada.

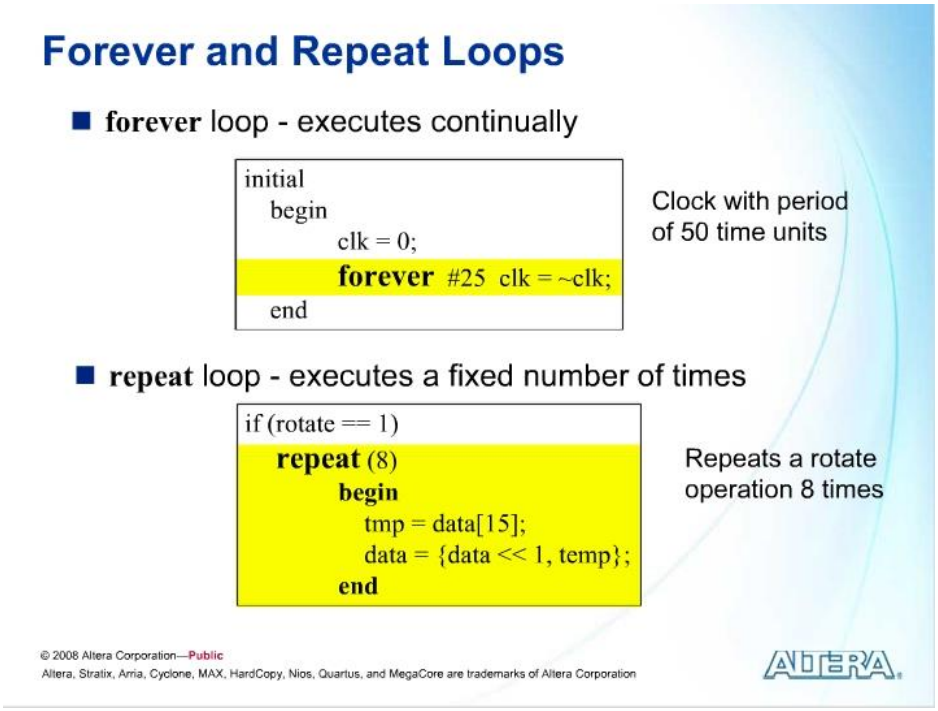


O Verilog também define dois outros casos da declaração case, denominados declaração casez e declaração casex. A declaração casez trata todos os valores "z" na condição case como não importa, ao invés de valores lógicos. Todos os valores "z" podem ser representados por um ponto de interrogação. No exemplo demonstrado aqui, se o valor do encoder igual a "B1zzz", na declaração case é representado por "B1???" e a declaração é executada como uma condição verdadeira, e a declaração "high_lvl=3" é executada. A declaração casex trata todos os valores "x" e "z" nas condições como não importa ao invés de valores lógicos. Então, sempre que houver um "x" na declaração casex, ele representa uma condição não importa, e a declaração é executada de acordo com a condição.

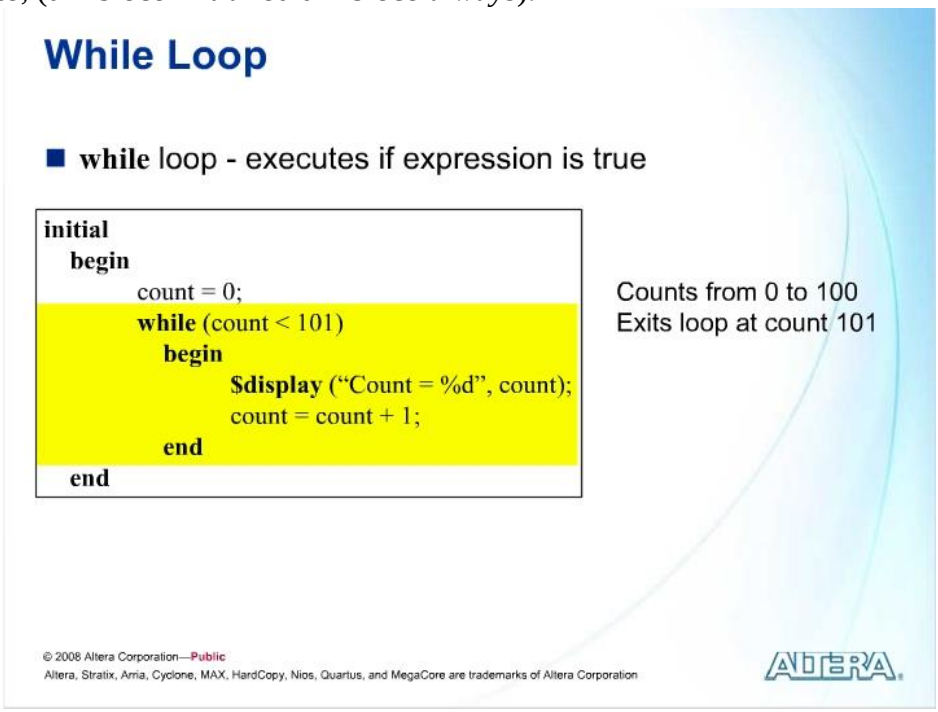


No Verilog estão definidas uma variedade de declarações loop. O loop forever é uma maneira comum de especificar sinais que irão se repetir continuamente, os quais são utilizados para geração de sinais de clock, por exemplo. Neste exemplo há um bloco *initial*, e ao adentrar o

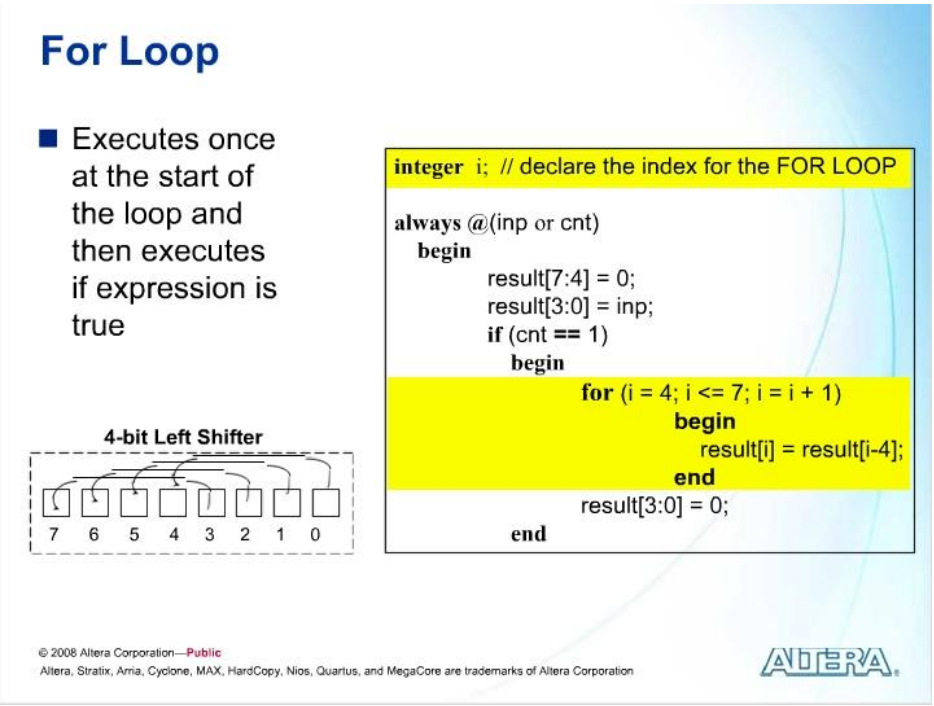
bloco o clock é inicializado com zero, e a cada 25 unidades de tempo o clock é alternado pela negação do sinal de clock, e na execução da simulação será observado este comportamento do clock. Outra forma de looping é a utilização da palavra chave repeat, e neste exemplo temos repeat(8), o que significa que todo o código que está encapsulado entre as declarações begin e end será repetido 8 vezes. Neste exemplo, é a implementação de uma operação de deslocamento de um registrador de oito bits.



Outra forma de loop que é definido no Verilog é o loop while. Neste exemplo, enquanto a contagem é menor que "101", as declarações entre o begin e end são executadas, e não importa se todas estas declarações estão encapsuladas dentro de um bloco *initial*, novamente estas são declarações de comportamento que necessitam estar dentro de um bloco de designação de procedimento, (um bloco *initial* ou um bloco *always*).



Outra forma comum de loop utilizada no Verilog é o loof for, que basicamente executa o loop de um início, e continua a execução enquanto uma condição do loop for verdadeira. No exemplo, o loop for vai ser executado e se iniciará designando um valor 4 à variável "i", e então esta variável é incrementada por 1 à cada iteração, que é executada continuamente até "i" ser menor ou igual a 7. Neste exemplo as declarações dentro do loop serão executadas para i = 4, 5, 6 ou 7.



A maneira em que o circuito é codificado terá um impacto direto na existência de sinais de controle síncronos ou assíncronos implementados nos flip-flops. O código no lado esquerdo abaixo demonstra como presets e clears síncronos necessitam ser codificados em Verilog. Neste código em particular, há um bloco *always* acionado pela borda positiva do sinal de clock. Neste bloco *always*, os valores ou os estados dos sinais clear e preset são verificados, e baseado nisto, o clear ou set do flip-flop é acionado, e se ambos não estão ativos, i.e., se ambos, neste caso, são iguais a zero, na borda de subida do clock a entrada "d" aparecerá, em sincronia com o sinal de clock, na saída "q" do flip-flop. O código à esquerda demonstra que na lista de sensibilidade, o código está sensível à borda de subida do clock ou do clear. Então se o sinal clear muda de zero para um, desconsiderando o valor do sinal do clock ou a borda do sinal do clock, a execução passa para o flip-flop sendo efetuado um clear, e então, esta implementação, em particular, é uma implementação assíncrona de um clear que é completamente independente do clock.

Synchronous vs. Asynchronous

Synchronous Preset & Clear

```
module sync (d,clk, clr, pre, q);

input d, clk, clr, pre ;
output q ;

reg q ;

always @(posedge clk)
        begin
            if (clr)
              q <= 1'b0 ;
            else if (pre)
              q <= 1'b1 ;
             else
               q <= d ;
         end
endmodule
```

Asynchronous Clear

```
module async (d,clk, clr, q);

input d, clk, clr  ;
output q ;

reg q ;

always @(posedge clk or posedge clr)
        begin
            if (clr)
              q <= 1'b0 ;
            else
               q <= d ;
         end
endmodule
```

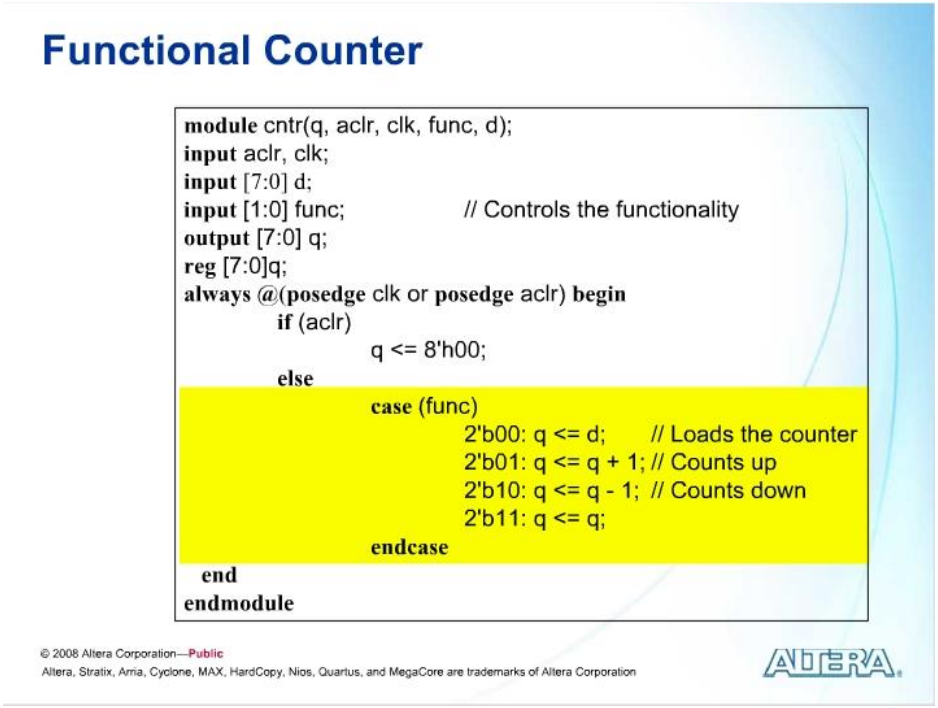

Para implementar a habilitação de um clock, a maneira mais fácil é demonstrada no exemplo abaixo, com um bloco *always* que é sensível à borda de subida do clock, e dentro deste bloco observa-se que o "if" é ativo se "ena" é igual a um, e então "d" é atribuído a "q". Mesmo que o clock ocorra, se "ena" é igual a zero no bloco *always*, "q" mantém seu estado prévio e o valor "d" não é atribuído no flip-flop.

Clock Enable

```
module clk_enb (d, ena, clk, q) ;

input d, ena, clk ;
output q ;

reg q ;

/* If clock enable port does not exist in target technology,
 a mux is generated */

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


No exemplo a seguir, de um contador que possui uma operação clear assíncrona, na lista de sensibilidade do bloco *always* há uma borda de subida do clock ou borda de subida de "aclr". No bloco *always*, se o clear assíncrono for ativado, a saída será limpa sendo atribuído o valor "00", caso contrário executa-se a seção else desta declaração if-else, dentro da qual há uma declaração case, que observa basicamente o valor dos sinais de controle de "func". Se o valor de "func" for igual a "00", a entrada no contador é carregada para inicializá-lo, se "func" é igual a

"1" é incrementado, se "func" é igual a "2" é decrementado e se "func" é igual a "3" o estado atual do contador se mantém.



A linguagem Verilog também define funções e tarefas, que são subprogramas, similares aos subprogramas vistos em linguagens de alto nível como C, C++, Pascal, ou qualquer outra linguagem de alto nível usual. São úteis para código repetitivo com similaridades e diferenças entre estes dois tipos. Adiciona legibilidade ao módulo, possibilitando a compressão do código. Uma função geralmente retorna um único valor, baseado nas suas entradas, e geralmente produz lógica combinacional. Por exemplo, se utilizarmos uma declaração continua de designação assign como:

assign mult_out=mult(ina,inb);

mul neste caso é uma função que foi definida anteriormente. Uma tarefa no contexto do Verilog é como um procedimento em outras linguagens de alto nível. Pode representar lógica combinacional ou de registradores. Tarefas são invocadas como declarações no código Verilog, e a linha de código instanciado com uma tarefa será substituída pelo código escrito dentro da tarefa. No exemplo stm_out(nxt, first, sel, filter) a tarefa é denominada stm_out com variáveis nxt, first, sel e filter.

O exemplo a seguir demonstra uma definição de função, que se inicia com a palavra chave "function", a especificação do nome da função, e se é uma função de um bit ou multibit, i.e., se vai retornar um valor de um bit ou multibit. É especificada a entrada para a função, neste exemplo duas entradas "a" e "b", e duas variáveis locais "r" e "i" a serem utilizadas dentro da função. No exemplo, a função implementa uma operação de multiplicação utilizando o algoritmo combinatório. Nota-se que a última declaração na função atribuí um valor ao nome da função que será retornado após a chamada a esta função.

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
```

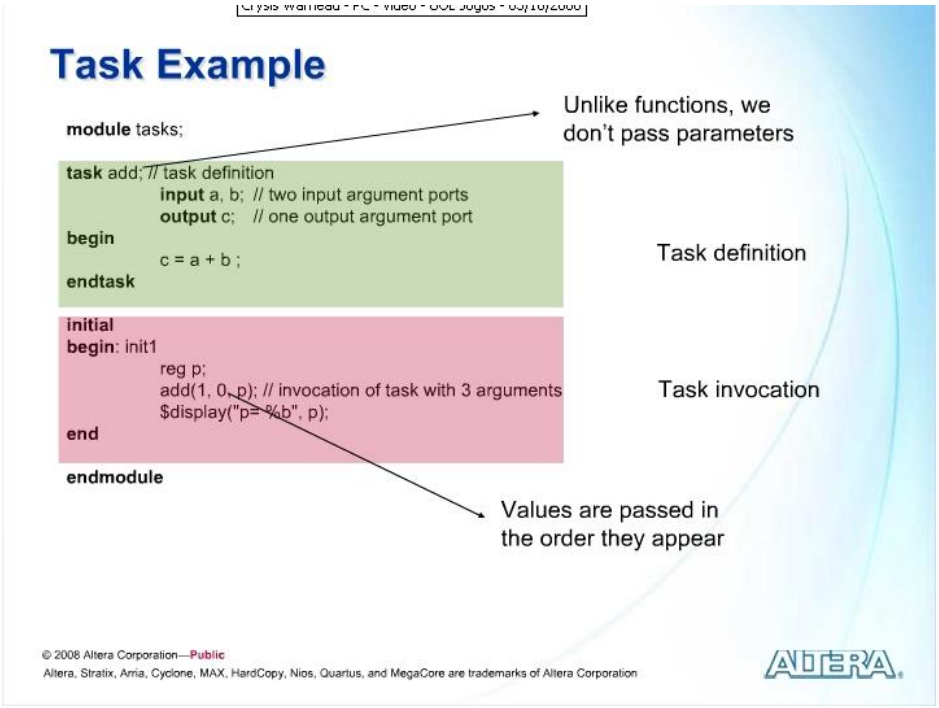

No exemplo abaixo é demonstrada uma instanciação da função que foi declarada, com uma declaração de designação continua ou combinacional que designa o valor que é retornado pela função "mult", com entradas "a" e "b", sendo atribuído à variável "mult_out".

Function Invocation - MAC

```
`timescale 1 ns/ 10 ps
module mult_acc (out, ina, inb, clk, clr);

input [7:0] ina, inb;
input clk, clr;
output [15:0] out;

wire [15:0] mult_out, adder_out;
reg [15:0] out;

parameter set = 10;
parameter hld = 20;

assign adder_out = mult_out + out;
always @ (posedge clk or posedge clr)
          if (clr)      out = 16'h0000;
          else          out = adder_out;

// Function Invocation
assign   mult_out = mult(ina, inb);

endmodule
```


A seguir encontra-se um exemplo completo de uma tarefa, com um módulo denominado "tasks" que define basicamente uma tarefa simples denominada "add". Serão transmitidos dois parâmetros para esta tarefa, parâmetros "a" e "b", sendo produzido um parâmetro denominado "c". Dentro do corpo da tarefa é definida uma operação, neste exemplo "c=a+b". Dentro do módulo há um bloco de procedimento *initial* que é utilizado para executar algumas designações de procedimentos, encapsuladas entre um begin e um end porque existem múltiplas declarações dentro deste bloco *initial*. Uma variável local é definida e denominada "p", e então é efetuada uma chamada à tarefa "add" com três parâmetros de valores 1, 0 e "p" (1 é atribuído à variável "a", 0 à "b" e "p" à "c"). Após a chamada desta função, será atribuído à "p" o valor 1 (resultado

da operação 1+0). O "$display" é uma chamada do sistema que irá exibir o valor de "p". Este é um exemplo simples de tarefa, mas engloba os elementos diferentes de uma tarefa que podem ser utilizados dentro de um módulo.



Na tabela abaixo são apresentadas algumas diferenças entre tarefas e funções, e uma das principais é que fundamentalmente uma função é executada no tempo 0, enquanto uma tarefa é basicamente uma substituição de uma porção de código dentro de um módulo Verilog no ponto em que é instanciada.