FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA
ENSINANDO E APRENDENDO

T566 –SISTEMAS DIGITAIS AVANÇADOS

Aula 14- Verilog Avançado

Prof. Danilo Reis

Simulação vs. Síntese

Simulação
- Código é executado do jeito exato que foi escrito
- Usuário tem a flexibilidade de escrever no modelo comportamental.

Síntese
- Código é interpretado& hardware é criado
  - Conhecimentos da arquitetura do PLD é importante
- Ferramenta de síntese requer correta codificação para gerar a lógica correta
  - Subconjunto da linguagem Verilog é suportado
  - Estilo de codificação é importante para criar lógicas eficientes e rápidas
- Inicialização controlada pelo dispositivo
  - Implementação Lógica pode ser ajustada para suportar inicialização

Pre- & post-synthesis logic devem operar do mesmo modo

Escrevendo Sintetizável Verilog

- Construções Verilog Sintetizável
- Sensitivity lists
- Blocking vs. non-blocking
- Latches vs. registers
- Estruturas **if-else**
- Instruções **case**
- Usando **functions** e **tasks**
- loops combinacionais
- Gated clocks

Some Synthesizable Verilog Constructs

- Module
- Ports
- Parameter
- Net data types
  - wire, tri
  - tri0, tri1
  - wor, wand, trior, triand
  - supply0, supply1
- Variable (register) data types
  - reg, integer
- Assign (continuous)
- Always & initial blocks
- Gates
  - and, nand, nor, or xor, xnor
  - buf, not, bufif0, bufif1, notif0, notif1
- UDPs
- Blocking (=) & non-blocking (<=)
- Sequential blocks (begin/end)
- Tasks & functions
- Bit & part select

- Behavioral constructs
  - If-else statements
  - Case statements
  - Loop statements (with restrictions)
- Disable
- Module instantiation
- Readmemb & readmemh system tasks
- Compiler directives
  - `define, `undef, `ifdef, `else, `endif
  - `include
  - `unconnected_drive, `nounconnected_drive
  - `resetall

– *Synthesis tools may place certain restrictions on supported constructs*
– *See the online help in Quartus II (or your target synthesis tool) for a complete list*
  – *The Quartus II software supports many constructs not supported by other synthesis tools*

Algumas contruções Verilog Não Sintetizáveis

- Net data type
  - trireg
- Variable data types
  - Real, time
- Gates
  - rtran,tran,tranif0, tranif1 rtranif0,rtranif1
- Switches
  - cmos, nmos, rcmos, rnmos, pmos, rpmos
- Pullup & pulldown
- Strengths*
- Assign (procedural continuous) & deassign
- Force & release
- Delay controls*
- Event (intra-procedural)
- Named event
- Fork & join
- Specify blocks*
- `timescale*
- Most system tasks
  - Display systems tasks*

- *These are some of the constructs not supported by Quartus II synthesis*
  - ** = Ignored for synthesis*
- *See the online help in Quartus II (or your target synthesis tool) for a complete list*

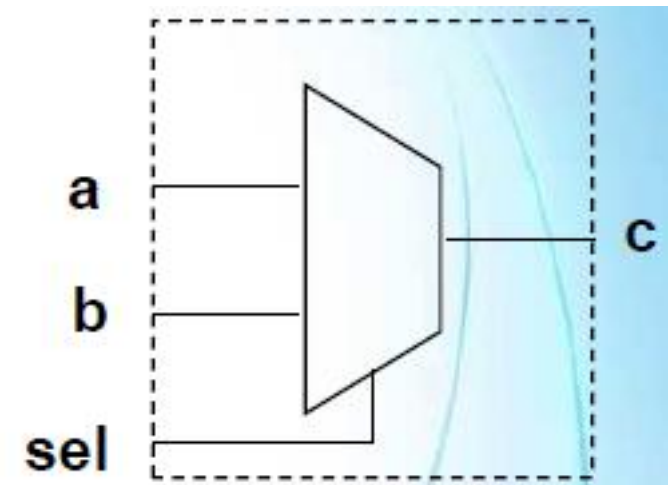
See:

- http://eesun.free.fr/DOC/VERILOG/verilog_manual1.html#14.1
- http://www.asic-world.com/verilog/synthesis2.html
- Quartus II HELP

Dois Tipos de processos RTL

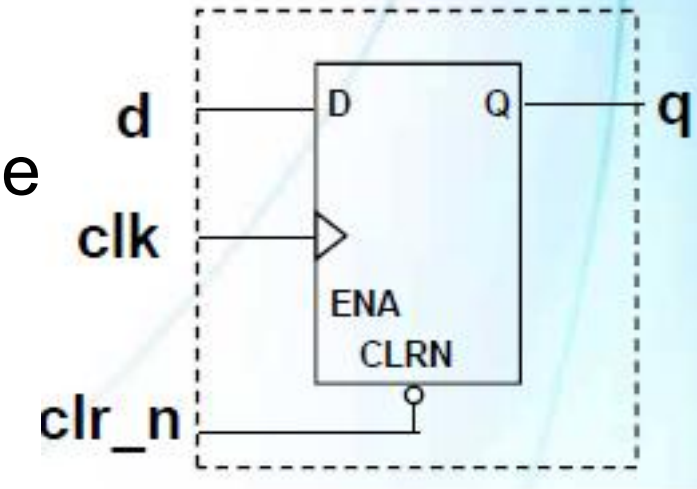
Processo Combinacional
– Sensível a todas entradas da lógica combinacional

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



Processo gatilhado a Clock
– Sensível ao clock e/ou sinais de controle

always @(posedge clk, negedge clr_n)



Sensitivity Lists

Sensitivity list incompleta no bloco **always** pode resultar em diferentes simulações RTL e gate-level

- Síntese assume sensitivity list completa
- Deve-se incluir todas entradas ao bloco(ou use @* shortcut)

```
always @ (a, b)
    y = a & b & c;
```

Incorrect Way – the simulated behavior is not that of the synthesized 3-input AND gate

```
always @ *
    y = a & b & c;
```

Correct way for the intended AND logic !

Blocking vs. Non-Blocking Recommendations

Blocking

- Use atribuições blocking em procedimentos combinacionais
- Atribuições Blocking são fáceis de ler
- Atribuições Blocking utilizam menos memórias & simulam mais rápido
  - Não agendam atualização de sinais

Non-blocking

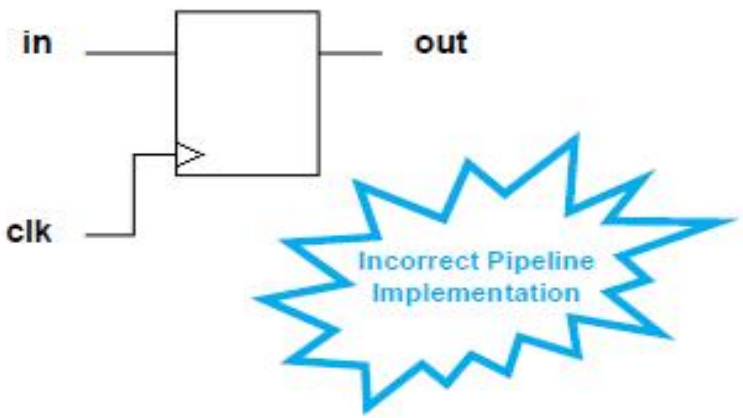
- Use atribuições non-blocking para procedimentos gatilhados ao clocked

Exemplo de processo gatilhado ao clock

Blocking (=)

```
always @(posedge clk)
  begin
        a = in;
        b = a;
        out = b;
  end
```

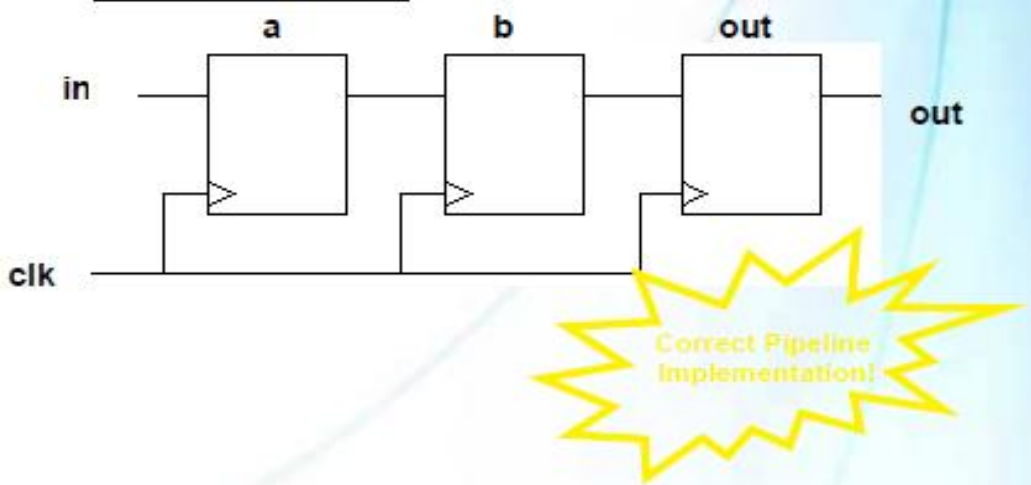
Synthesized Result:



Nonblocking (<=)

```
always @ (posedge clk)
  begin
        a <= in;
        b <= a;
        out <= b;
  end
```

Synthesized Result:



Latches vs. Registers

Dispositivos Altera tem registradores nos elementos lógicos;
Latches implementados com lógica combinacional & pode tornar a análise de tempo mais complicada
- Look-up table (LUT) devices usam LUTs com loop combinacional
- Product-term devices usam mais product-terms

Recomendações
- Projete com registradores (RTL)
- Cuidado com latches inferidos

Latches são inferidos na lógica combinacional de saída quando existem saídas não especificadas para o conjunto das condições de entrada
Leva a simulation/synthesis não serem as mesmas

if-else Structure

Estruturas **if-else** implica em priorização e dependência
- Nth clausula implica que todas N-1 prévias clausulas são falsas
  - Cuidado com lógica aninhada ou “ballooning” logic

Logical Equation

$$(<cond1> \cdot A) + (\overline{<cond1>} \cdot <cond2> \cdot B) + (\overline{<cond1>} \cdot \overline{<cond2>} \cdot cond3 \cdot C) + \ldots$$

Considere restruturar as instruções **if** de forma plana como um multiplexador e reduz a lógica

I
r

```
IF <cond1> THEN
    IF <cond2> THEN
```

```
IF <cond1> AND <cond2> THEN
```

Quando escrever estruturas **if-else**

Cobrir todos os casos
- Casos não tratados resultam em latches

Atribui valores antes de ler objetos data type
- Ler objetos data type sem atribui-los resulta em latches

Por eficiência, considere
- Use don’t cares (X) para a clausula else final (evitando desnecessária condição default)

Ferramenta de Síntese tem liberdade de códificar don't care com a máxima otimização.
- Atribuindo valores e explicitando as cobertutas diferentes do valor inicial.

Priorização & Dependencia (if-else)

else aninhados implica priorização e dependência na lógica

```
reg [4:0] state;
parameter  s0 = 5'h0, s1 = 5'h11, s2 = 5'h12, s3 = 5'h14, s4 = 5'h18;
always @ (posedge clk, negedge reset_n) begin
      if (reset_n == 0) state <= s0;
      else begin
            if (state == s0) begin
                  if (in == 1) state <= s1;  else state <= s0;
            end
            else if (state == s1) state <= s2;
            else if (state == s2) state <= s3;
            else if (state == s3) state <= s4;
            else if (state == s4) begin
                  if (in == 1) state <= s1;  else state <= s0;
            end
      end
end
```

Priorização & Removendo Dependencia (if-else)

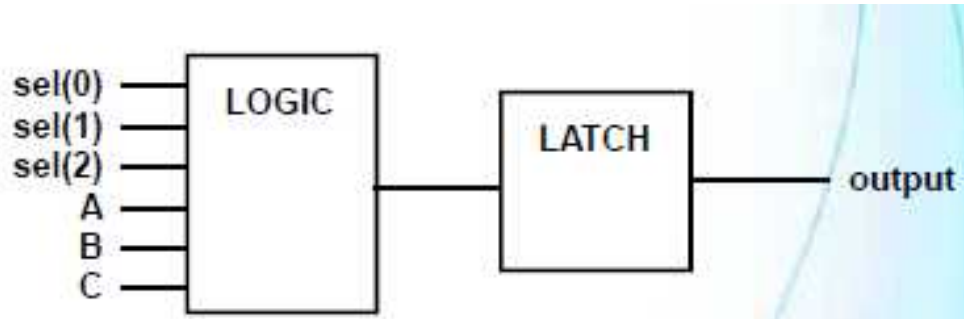
Instruções **if** Individuais quando a lógica é exclusiva pode ser mais eficiente.

```
reg [4:0] state;
parameter s0 = 5'h0, s1 = 5'h11, s2 = 5'h12, s3 = 5'h14, s4 = 5'h18;
always @ (posedge clk, negedge reset_n) begin
      if (reset_n == 0) state <= s0;
      else begin
            if (state == s0) begin
                  if (in == 1) state <= s1;  else state <= s0;
            end
            if (state == s1) state <= s2;
            if (state == s2) state <= s3;
            if (state == s3) state <= s4;
            if (state == s4) begin
                  if (in == 1) state <= s1;  else state <= s0;
            end
      end
end
```

Latches indesejados

Processos combinacionais que não cobrem todas as possibilidades nas condições de entrada geram latches

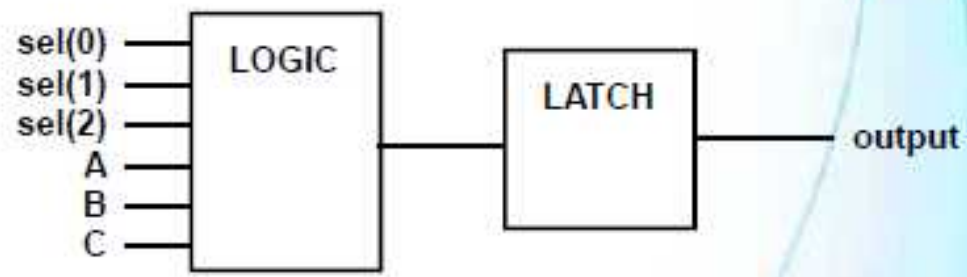
```
always @ *
begin
        if (sel == 3'b001)
                out = a;
        else if (sel == 3'b010)
                out = b;
        else if (sel == 3'b100)
                out = c;
end
```



Removendo Latches indesejados

Feche todas estruturas **if-else**
- Se possível atribua “don’t care’s” para clausula **else** para melhorar a otimização da lógica

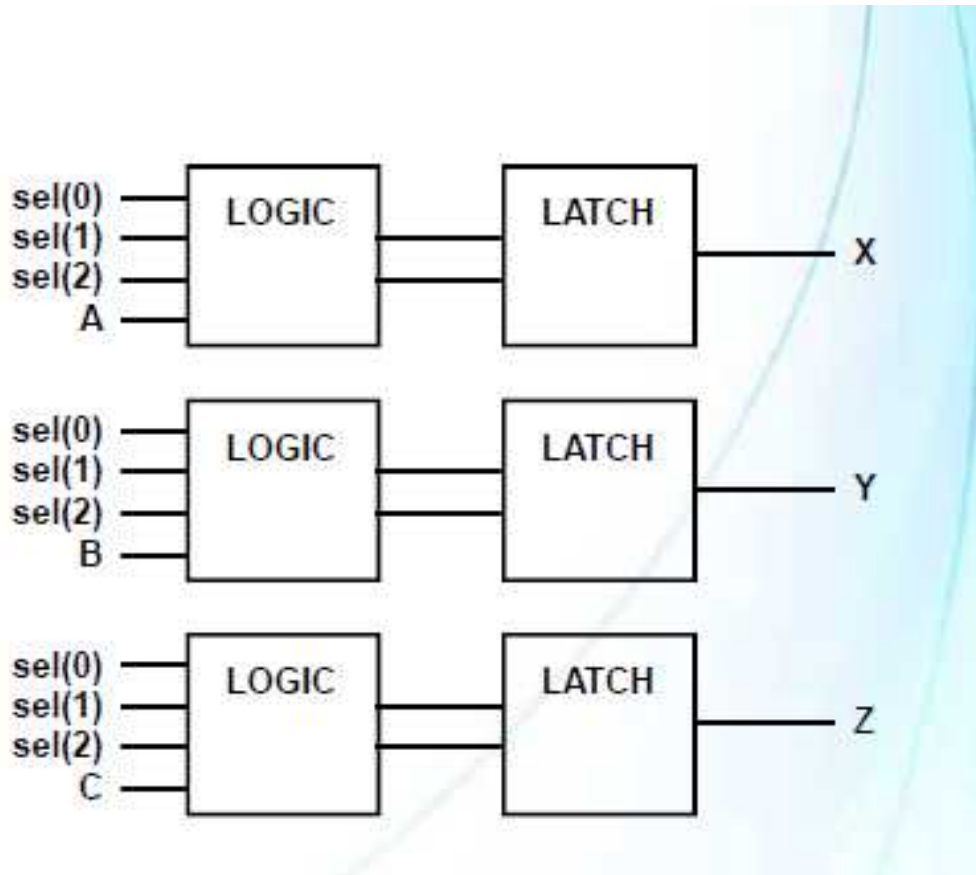
```
always @ *
begin
        if (sel == 3'b001)
                out = a;
        else if (sel == 3'b010)
                out = b;
        else if (sel == 3'b100)
                out = c;
end
```



Latches indesejados com if=else mutuamente exclusivos

Cuidado com a construção de dependências desnecessárias
- Por exemplo as saídas x, y, z são mutuamente exclusivas,clausulas **if-else** serem dependentes de todos os teste criam latches

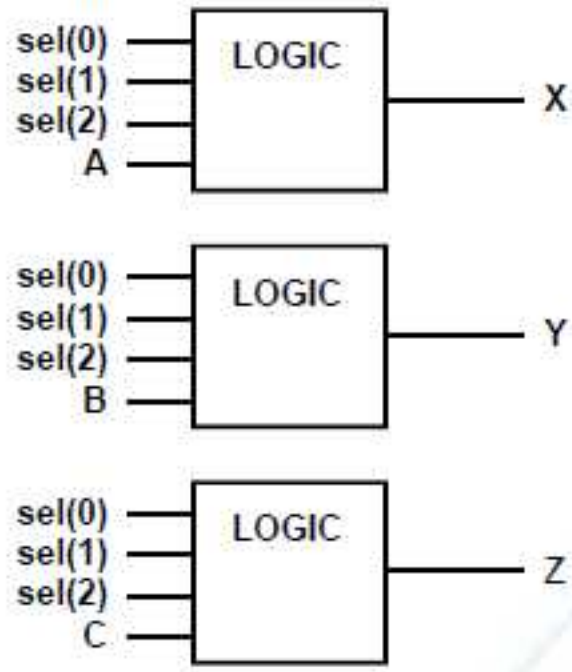
```
always @ (sel,a,b,c)
begin
        if (sel == 3'b010)
                x = a;
        else if (sel == 3'b100)
                y = b;
        else if (sel == 3'b001)
                z = c;
        else
                x = 0;
                y = 0;
                z = 0;
end
```



Removendo Latches indesejados mutuamente exclusivos

Use instruções **if** separadas e feche cada uma com uma clausula de inicialização ou clausula **else.**

```
always @ (sel,a,b,c)
begin
        x = 0;
        y = 0;
        z = 0;
        if (sel == 3'b010)
                x = a;
        if (sel == 3'b100)
                y = b;
        if (sel == 3'b001)
                z = c;
end
```

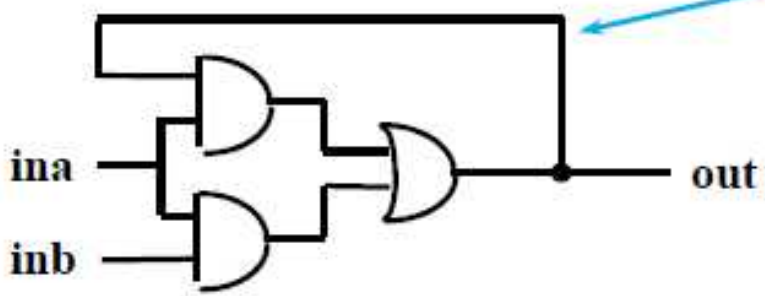


Latches indesejados if=else aninhados

Use instruções **if** aninhadas com cuidado
- Por exemplo Estes if aninhados não cobrem todas as possibilidades.(instrução if aberta) & latch é criado

```
always @ (ina, inb)
begin
    if (ina == 1) begin
        if (inb == 1) out = 1;
        end
    else out = 0;
end
```

ina	inb	out
1	1	1
0	0	0
0	1	0
1	0	?

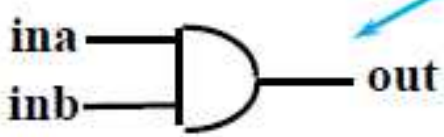


- Uncovered cases infer latches
  - No default value for data type objects
- Equation
  - A1L3 = LCELL(ina & (A1L3 # inb));

Removendo Latches indesejados if=else aninhados

```
always @ (ina, inb)
begin
    out = 0;
    if (ina == 1)
        if (inb == 1)
            out = 1;
end
```

ina	inb	out
1	1	1
0	0	0
0	1	0
1	0	0

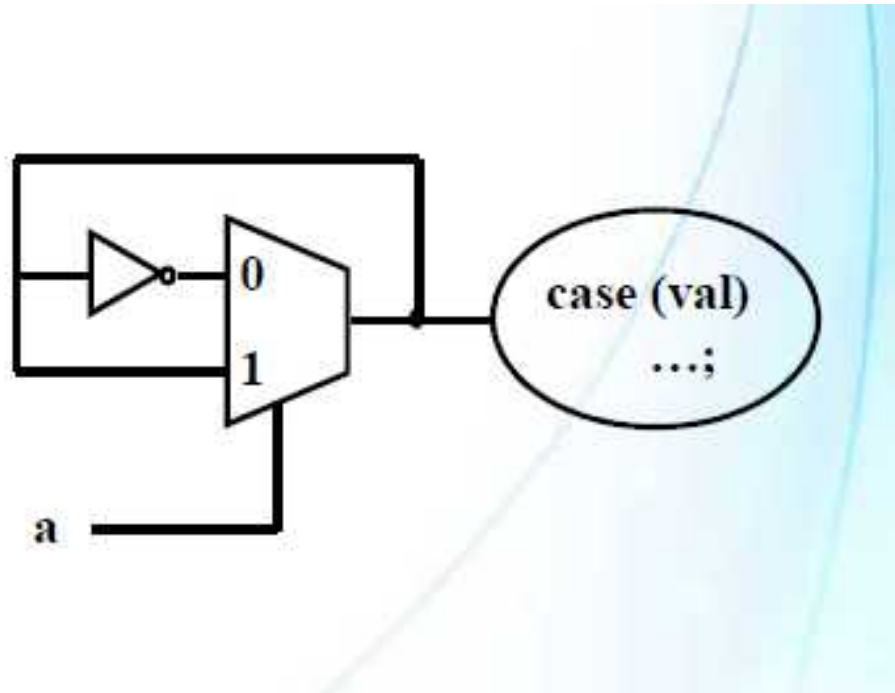


- Using initialization to cover all cases; no latch inferred
- Equation
  - A1L3 = inb & ina;

Variaveis não inicializadas

A leitura de um objeto data type em um processo combinacional antes deste objeto ter sido atribuido um valor infere um latch.

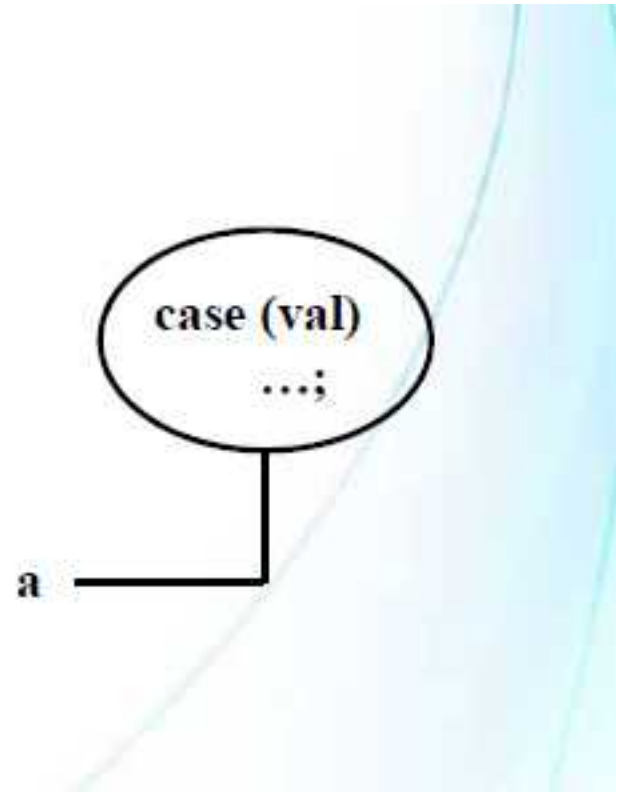
```
always @ *
    begin
        if (a == 1) val = val;
        else val = val + 1'b1;
        case (val)
                1'b0 : q = i[0];
                1'b1 : q = i[1];
        endcase
    end
```



Inicializando Variáveis

Atribuir um valor ao objeto data type antes da leitura previmi a inferência do latch

```
always @ *
    begin
        val = 1'b0;
        if (a == 1) val = val;
        else val = val + 1'b1;
        case (val)
                1'b0 : q = i[0];
                1'b1 : q = i[1];
        endcase
    end
```



Instrução **Case**

- Instruções **Case** usualmente sintezam mais eficientemente quando existe condições mutualmente exclusivas.
- Escreve-se instruções **case** quando:
  - Tem-se a certeza que todos os itens são únicos/paralelos
    - A linguagem Verilog não exige isto.
    - Cases não paralelos inferem codificação priorizada (menos eficiente)
  - Cobrir todos os casos
    - Como **if-else**, casos não cobertos inferem latches
    - Causados por:
      - Instrução **case** Incompleta
      - Saídas não definidas para um dos casos

Recomendações Instrução **Case**

- Inicialize todos casos de saídas ou assegure que as saídas foram atribuídas em todos os casos
- Use a clausula default para fechar os casos indefinidos(se sobrou algum)
- Atribua valores iniciais ou valores default para "don’t cares" (X) para otimização, se a lógica permitir.

Exemplo de instrução **Case**

```
always @ *
begin
//   in1 = 32'bx;
     case (state)
          4'b0000:   in1 = data_a;
          4'b1001:   in1 = data_b;
          4'b1010:   in1 = data_c;
          4'b1100:   in1 = data_d;
     endcase
end
```

Incomplete case

```
always @ *
begin
     case (state)
          4'b0000:   in1 = data_a;
          4'b1001:   in1 = data_b;
          4'b1010:   in1 = data_c;
          4'b1100:   in1 = data_d;
          default:   in1 = 32'bx;
     endcase
end
```

Completed case

Full Case

- Definição: Todos possíveis padrões case-expression binary pode ser tratados por um item do case ou caso default
- Instruções Non-full case inferem latches
- Adicionar "full_case synthesis directive" para instruções non-full case;
  - Usados para reduzir contagem de lógica na síntese contando apenas para as possibilidades definidas
  - Pode causar diferencas entre a pre-synthesis e post-synthesis simulation;

Mismatched Code (BAD)

```
always @ (ena or a) begin
      case ({ena,a}) //synthesis full_case
            2'b1_0:  out[a] = 1'b1;
            2'b1_1:  out[a] = 1'b1;
      endcase
      end
```

Full Statement (GOOD)

```
always @ (ena or a) begin
      case ({ena,a})
            2'b1_0:  out[a] = 1'b1;
            2'b1_1:  out[a] = 1'b1;
            default:  out = 2'bxx;
      endcase
      end
```

Case Paralelo

- Definição: Única possibilidade de atender uma expressão case com um e apenas um case item
- Instruções Non-parallel case inferem codificação com prioridade
  - Codifique todas intensionais codificações prioritárias com instruções if-else.
- Adicionar “parallel_case synthesis directive" para instruções non-parallel case
  - Used to reduce the logic size as synthesis tool removes priority

Mismatched Code (BAD)

```
always @ (a) begin
      casez (a) //synthesis parallel_case
            3'b1??: out = 2'b01;
            3'b?1?: out = 2'b10;
            3'b??1: out = 2'b11;
      endcase
end
```

Non-Prioritized Code (GOOD)

```
always @ (a) begin
      case (a)
            3'b100: out = 2'b01;
            3'b010: out = 2'b10;
            3'b001: out = 2'b11;
            default: out = 2'bxx;
      endcase
end
```

Full Case & Parallel case Directives

- Não use!
  - Code synthesizable case statements to be “full” and “parallel” when designing
- Excelente leitura para discutir a síntese da instrução case
  - http://www.sunburst-design.com/papers/CummingsSNUG1999Boston_FullParallelCase

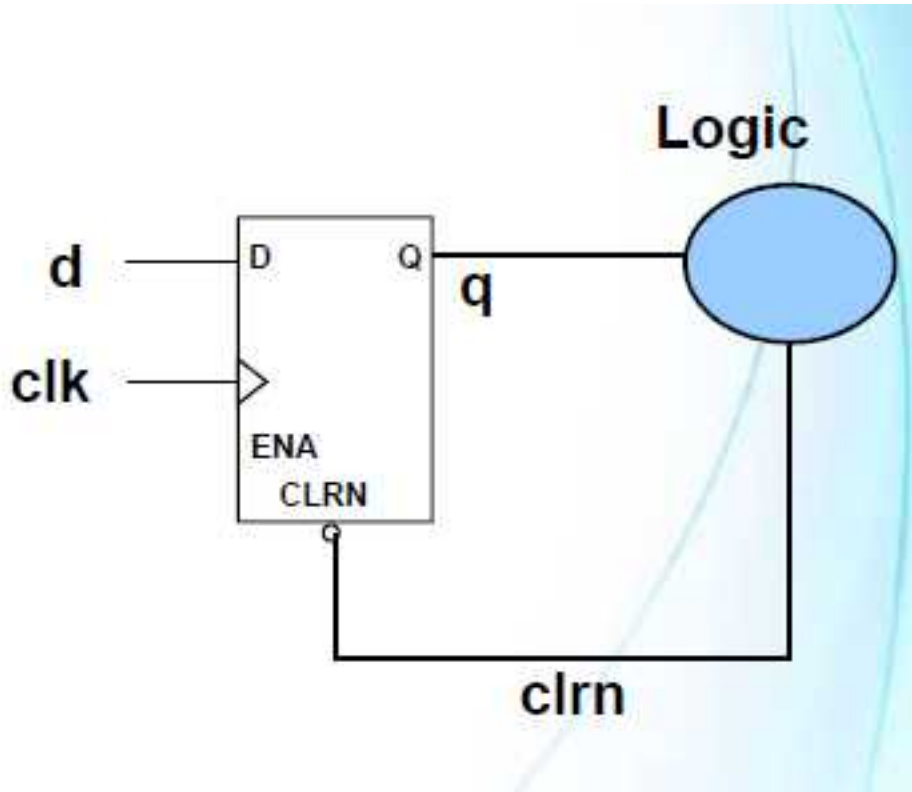
Usando Functions & Tasks

- Functions e tasks podem serem utilizadas para sintetizar módulos tornando o código mais legível e reusável;
- Function
  - Sintetiza lógica combinacional;
  - Variáveis nas functions podem ser locais
- Tasks
  - Sintetiza lógica combinacional ou sequencial;
  - Não podem conter time delay
- Cada chamada gera um bloco de lógica separada
  - Não tem compartilhamento de lógica;
  - Implementa compartilhamento de recursos, se possível (discutido depois)

Loops Combinationais

- Causa comum de instabilidades
- Comportamento do loop depende do delay através da lógica;
  - Delay de propagação pode mudar
- Comportamento da simulação e hardware podem ser diferentes

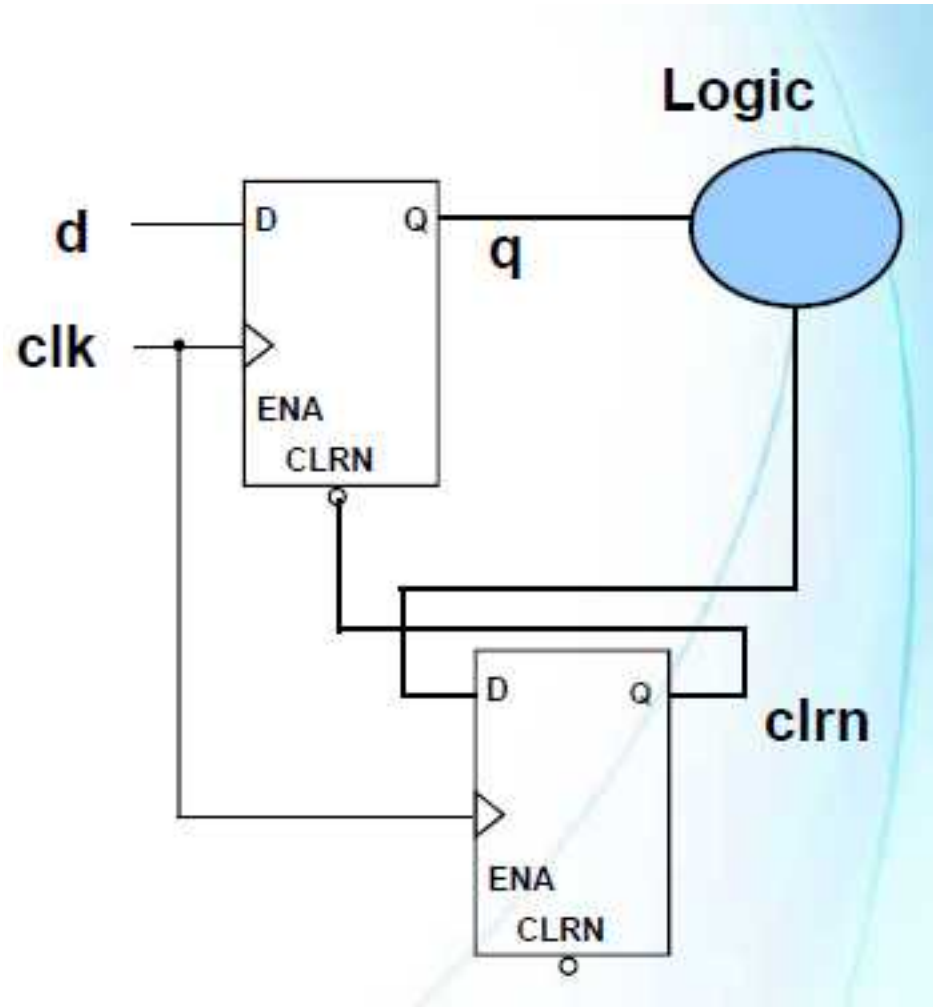
```
always @ (posedge clk, negedge clrn)
 begin
      if (!clrn)
            q <= 0;
      else
            q <= d;
end

assign clrn = (ctrl1 ^ ctrl2) & q;
```



Loops Combinationais

- Todos loops de realimentação devem ter um registrador

```
always @ (posedge clk, negedge clrn)
 begin
      if (!clrn)
            q <= 0;
      else
            q <= d;
end

always @ (posedge clk)
      clrn <= (ctrl1 ^ ctrl2) & q;
```



Gated Clocks

- Podem liderar problemas funcionais e de timing;
  - Comportamento do Clock sujeito a síntese e placement & routing
  - Pode ser a fonte de um clock skew (interferência cruzada)
  - Possíveis glitches na path do clock
- Recomendações:
  - Use habilitação de clock para funcionalidade de clock gating;
  - Use recursos dedicados do dispositivo (clock control blocks) para sincronização e redução de consumo de potência com gate clocks
  - Se você tiver que construir sua lógica de gating clock
    - Use estrutura de gating sincrona;
    - Assegure que o clock global é utilizado como sinal de clock
    - Gate o clock na fonte

Exemplos de Gated Clocks

```
assign g_clk = gate & clk;

always @ (posedge g_clk, negedge clrn)
 begin
      if (!clrn)
            q <= 0;
      else
            q <= d;
end
```

Poor clock gating – Active clock edges occurring near gate signal changes may result in glitches

```
always @ (negedge clk)
      sgate = gate

assign g_clk = sgate & clk;

always @ (posedge g_clk, negedge clrn)
 begin
      if (!clrn)
            q <= 0;
      else
            q <= d;
end
```

Better clock gating – Gate signal clocked by falling edge clk, so gate may only change on inactive clock edge (Use OR gate when falling edge is the active clock edge)

Inferindo Funções Lógicas

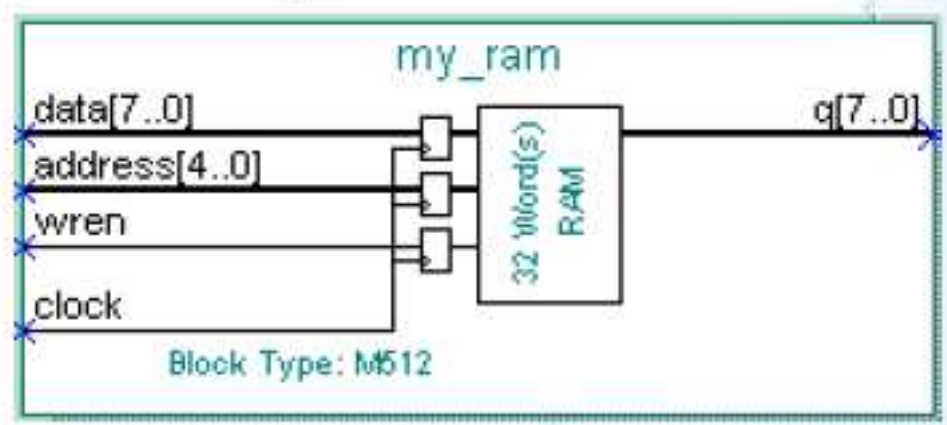
Usando modelo comportamental para descrever a lógica
Ferramenta de síntese reconhece a descrição e inserindo funções lógicas equivalentes(megafunctions).
- Funções tipicamente pre-optimizadas para utilização ou performance de uso geral;
Use os templates da ferramenta síntese como ponto de início.
- Use mode gráfico da ferramenta de síntese para verificar a reconhecimento da lógica.
Faça códigos independente do fabricante da FPGA

Inferindo Funções Lógicas

```
always @ (posedge clock)
    begin
        if (wren)
            mem[address] <= data;
        q <= mem[address_out];
    end
```

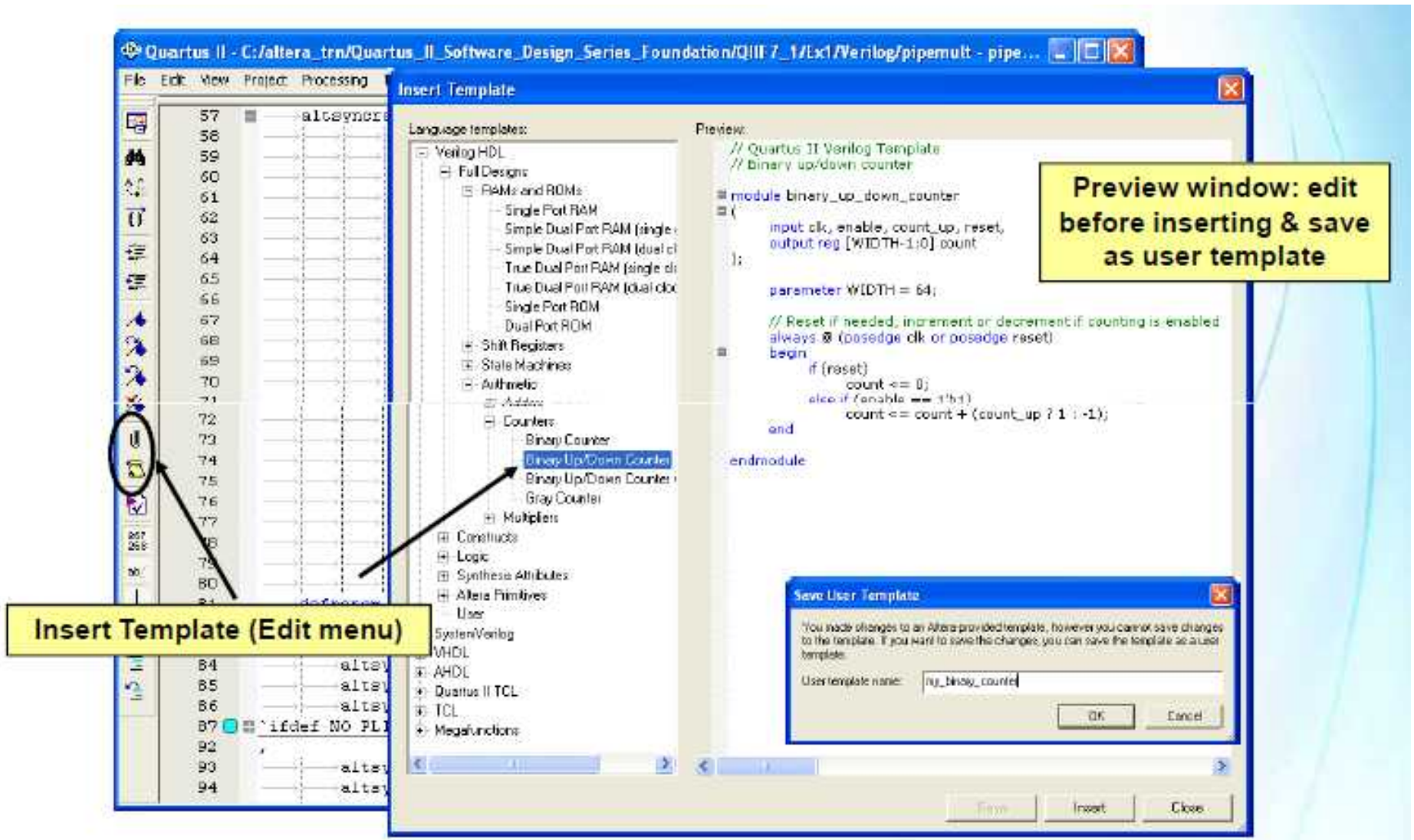
Synthesis tool sees

Replaces with



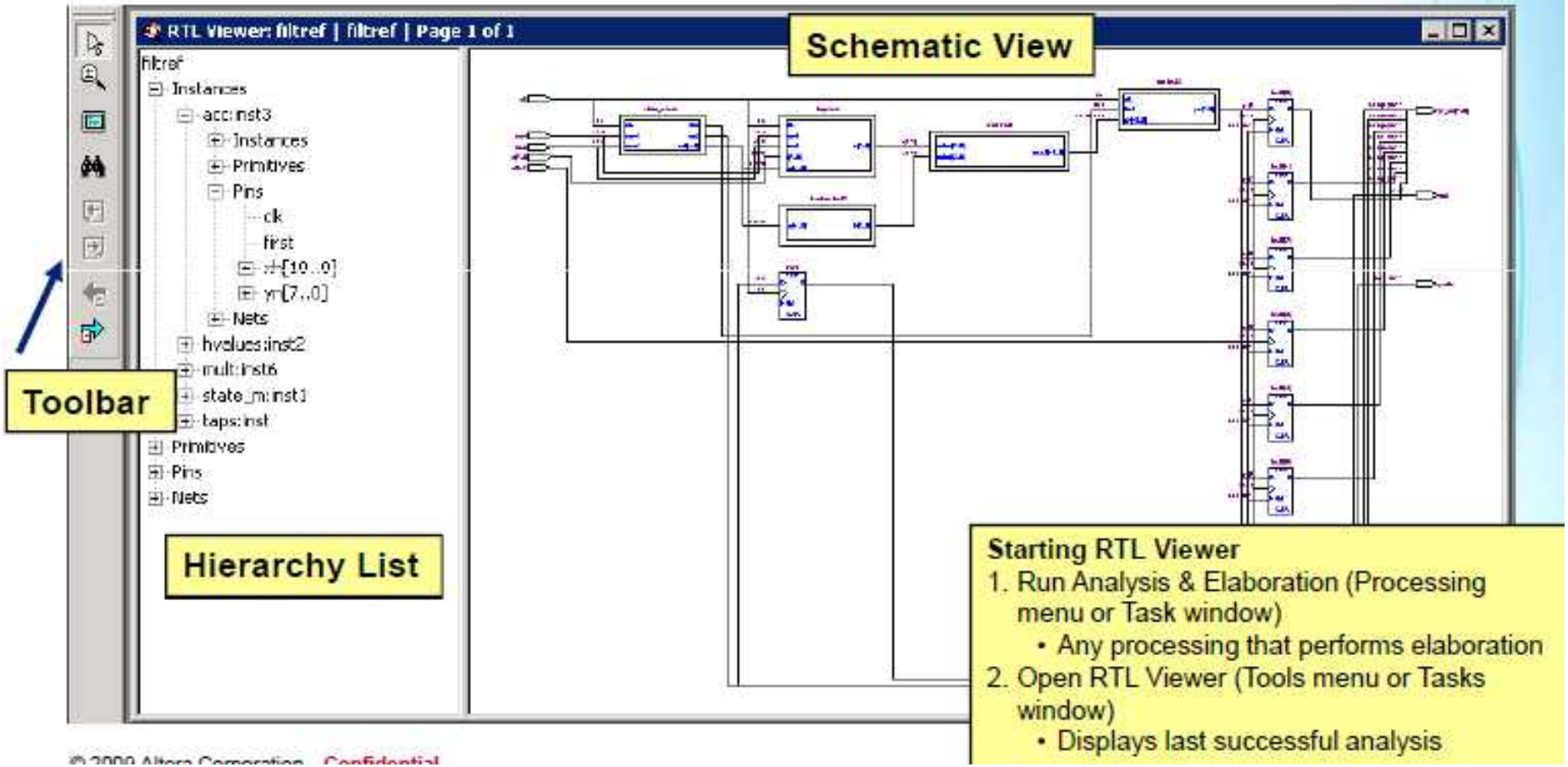
Altera megafunction and/or library cells

Quartus Verilog Templates



Quartus RTL Viewer

- Graphically represents results of synthesis

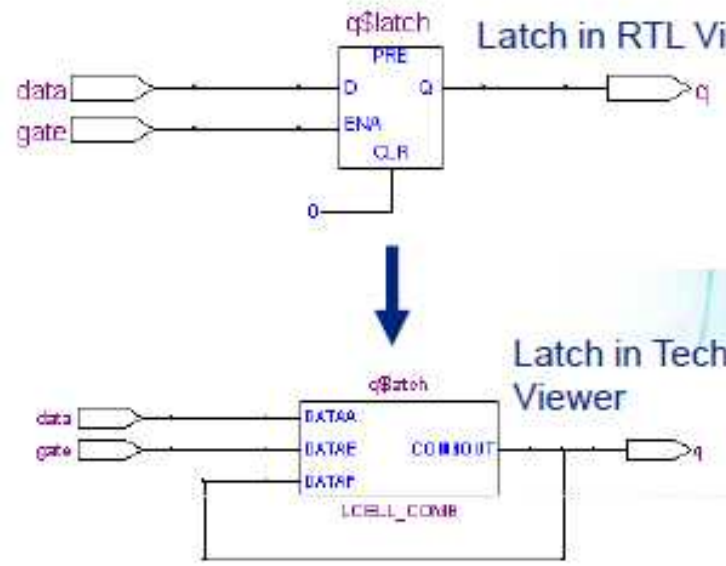


Inferindo Funções comuns

- Latches
- Registers
- Counters
- Tri-states
- Memory

Latches Desejádos

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

wire d, gate;

always @(d, gate)

    if (gate)
        q = d;
endmodule
```



sensitivity list includes both inputs

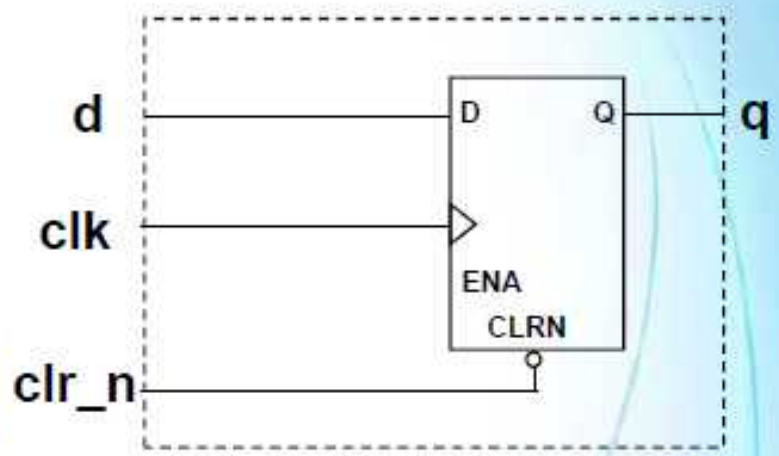
level sensitive…not edge

What happens if gate = '0'?

⇨ ***Implicit Memory & Feedback***

Inferindo Flip-flops

```
module basic_dff (
    input clk, d, clr_n;
    output reg q;
);
always @(posedge clk, negedge clr_n)
    if (!clr_n)
        q <= 0;
    else
        q <= d;
endmodule
```



- *Simple register logic with reset*
- *Recommendation: Always use reset (asynchronous or synchronous) to get system into known or initial state*

Controle secundário de sinais

- Sinais de controle de registradores varia entre as familias de FPGA &CPLD
  - Clear, preset, load, clock enable, etc.
- Evite usar sinais não disponíveis na arquitetura
  - Funcionalidades de projeto suportadas criando lógica extra na células
  - Menos eficiente, possíveis resultados mais lentos

DFF com controle secundário de sinais

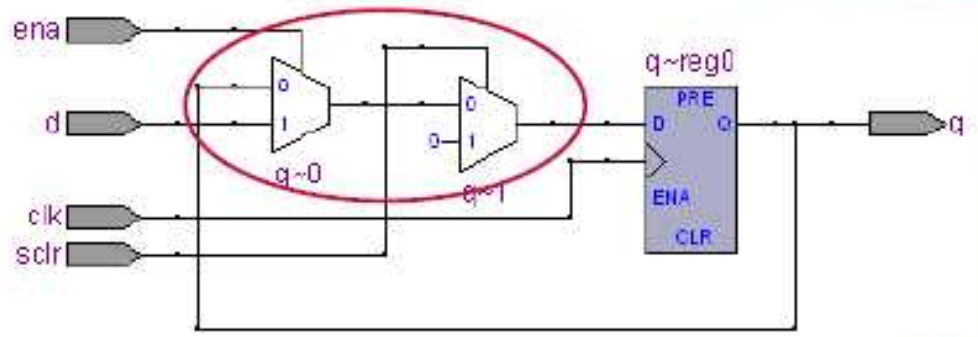
```
module dff_full (
      input clk, ena, d,
      input clr_n, sclr, pre_n,
      input aload, sload, adata, sdata,
      output reg q
);

always @(posedge clk, negedge clr_n,
                  negedge pre_n, posedge aload)
      if (!clr_n)
            q <= 1'b0;
      else if (!pre_n)
            q <= 1'b1;
      else if (aload)
            q <= adata;
      else if (ena)
            if (sclr)
                  q <= 1'b0;
            else if (sload)
                  q <= sdata;
            else
                  q <= d;
endmodule
```

- *This is how to implement all asynchronous and synchronous control signals for the Altera PLD registers*
  - *–Conditions in the sensitivity list are asynchronous*
  - *–Conditions not in the sensitivity list are synchronous*
- *Remove signals not required by your logic*
  - *–Most PLD architectures would require additional logic to support all at once*

- *Why is this asynchronous signal **adata** left out of the sensitivity list?*
  - *–Verilog limitation*
  - *–Inferred correctly (simulation mismatch)*

Prioridade incorreta controle secundário de sinais

```
module dff_sclr_ena (
      input clk, d, ena, sclr,
      output reg q
);

always @ (posedge clk)
      if (sclr)
            q <= 1'b0;
      else if (ena)
            q <= d;
endmodule
```



- *2 control signals*
- *Considerations*
  - *–Do the registers in the hardware have both ports available?*
  - *–How does hardware behave? Does clear or enable have priority?*
- *Sync clear has priority enable over in code*
- *Enable has priority over sync clear in silicon*
- *Additional logic needed to force code priority*

Prioridade no Controle secundário de sinais

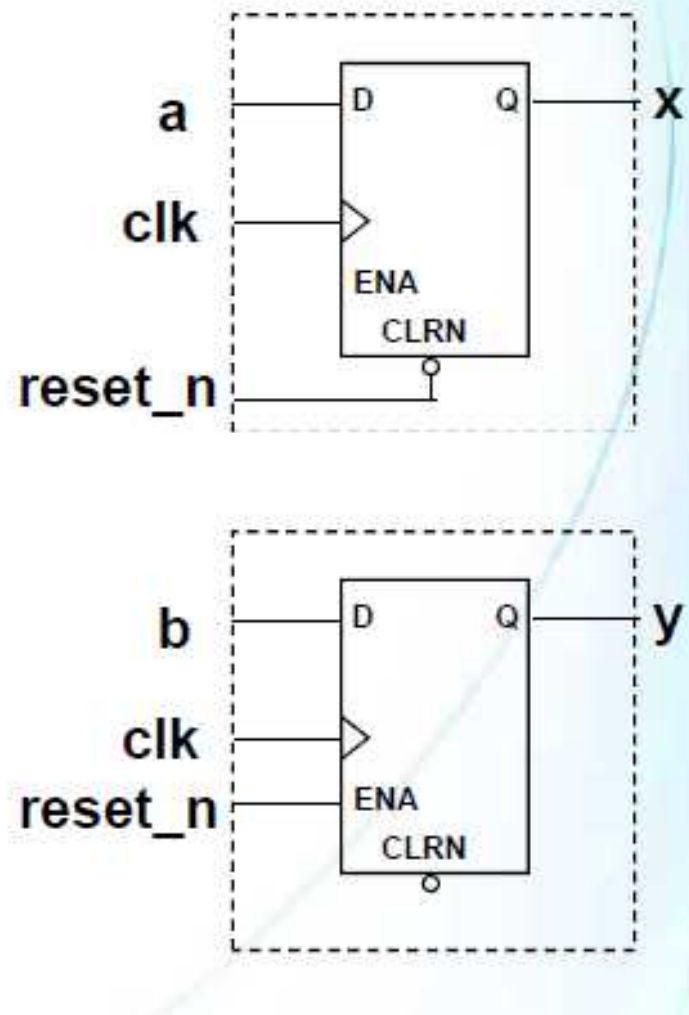
1. Asynchronous Clear (aclr)
2. Asynchronous preset (pre)
3. Asynchronous load (aload)
4. Enable (ena)
5. Synchronous clear (sclr)
6. Synchronous load (sload)

- Mesmo para todas as famílias FPGA da Altera
  - Todos os sinais não suportados para todas as famílias
- Re-ordenamento pode gerar lógica extra

Lógica incorreta de controle

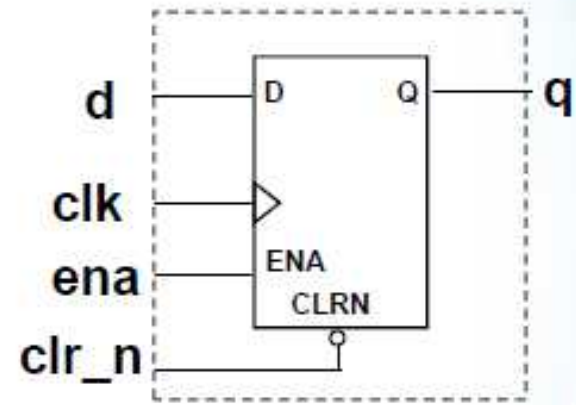
```
module dff_inc (
        input clk, reset_n, a, b,
        output reg x, y
);
always @(posedge clk, negedge reset_n)
        if (reset_n == 0)
                x <= 0;
        else begin
                x <= a;
                y <= b;
        end
endmodule
```

- *y Is Not Included in Reset Condition*
- ***reset** Clears **x** but Acts More Like an Enable for **y***



DFF com Clock Enable

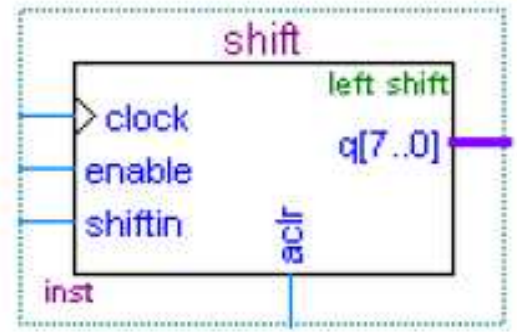
```
module dff_ena (
      input clk, clr_n, d,
      input ena_a, ena_b, ena_c,
      output reg q
);

      always @ (posedge clk, negedge clr_n)
          if (clr_n == 1'b0)
                q <= 1'b0;
          else if (ena == 1'b1)
                q <= d;

      assign ena <=
                (ena_a | ena_b) ^ ena_c;

endmodule
```



- *To ensure that this is synthesised using DFFE primitives (DFF with enable)*
  - *–Place the enable statement directly after the rising edge statement*
  - *–Place enable expressions in separate process or assignment*
- *If the synthesis tool does not recognize this as an enable it will be implemented using extra LUTs*

Shift Registers

```
module shift (
    input aclr_n, enable, shiftin, clock,
    output reg [7:0] q
);
always @(posedge clock, negedge aclr_n)
    if (!aclr_n)
        q[7:0] <= 0;
    else if (enable)
        q[7:0] <= {q[6:0], shiftin};
endmodule
```



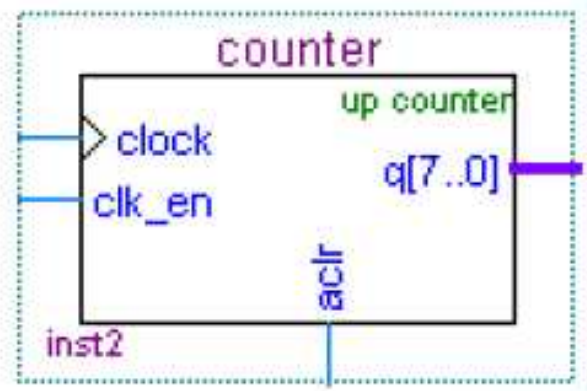
Shift function
•Use { , } for concatenation

- *Shift register with parallel output, serial input, asynchronous clear and enable which shifts left*
- *Add or remove synchronous controls in a manner similar to DFF*

Contador básico c/Async Clear & Clock Enable

```
module count (
      input clk, aclr_n, ena,
      output reg [7:0] q = 0
);

always @(posedge clk, negedge aclr_n)
      if (!aclr_n)
            q[7:0] <= 0;
      else if (ena)
            q <= q + 1;

endmodule
```

Count function



- *Binary up counter with asynchronous clear and clock enable*
- *Add or remove secondary controls similar to DFF*

Referências

- Curso oficial da Altera "Advanced Verilog Design Techniques"