



FUNDAÇÃO EDSON QUEIROZ  
UNIVERSIDADE DE FORTALEZA  
ENSINANDO E APRENDENDO

## T566 –SISTEMAS DIGITAIS AVANÇADOS

---

# Aula 15- Verilog Avançado

Prof. Danilo Reis



## Tri-States

IEEE-defined 'Z' or 'z' value

- Simulation: Behaves like high-impedance state
- Synthesis: Converted to tri-state buffers

Altera devices have tri-state buffers only in I/O cells

- Benefits:
  - Eliminates possible bus contention
  - Location of internal logic is a non-issue

Cost savings

Don't pay for unused tri-state buffers

- Less testing required of devices
- Internal tri-states must be converted to combinatorial logic
- Complex output enable may cause errors or inefficient logic

The wire and tri keywords have identical syntax and function

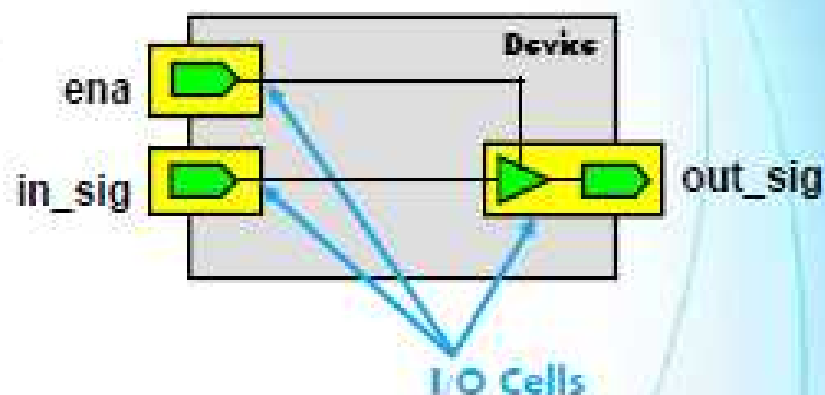
- Use wire to indicate a net with a single driver
- Use tri to indicate a net that can have multiple drivers



## Inferindo Tri-States corretamente

### Conditional Signal Assignment

```
module dff_1(in_sig, ena, out_sig);  
  
input in_sig, ena;  
output tri out_sig;  
  
assign out_sig = ena ? in_sig : 1'bZ;  
  
endmodule
```

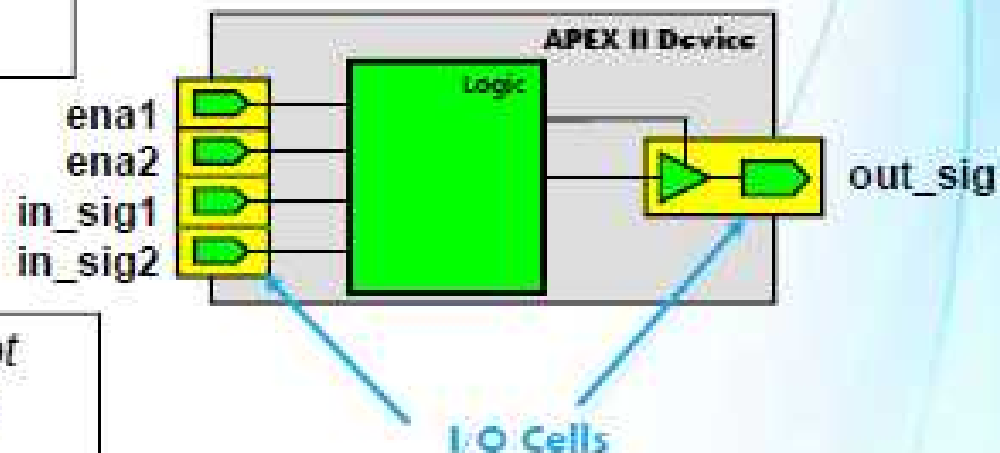


- Only 1 Assignment to Output Variable
- Uses Tri-State Buffer in I/O Cell
  - **out\_sig** should connect directly to top-level I/O pin (through hierarchy is ok)



## Inferindo Tri-States incorretamente

```
module dff_1(in_sig, ena, out_sig);  
  
input in_sig, ena;  
output tri out_sig;  
  
assign out_sig = ena1 ? in_sig1 : 1'bZ;  
assign out_sig = ena2 ? in_sig2 : 1'bZ;  
  
endmodule
```



- 2 Assignments to Same Signal Not Allowed in Synthesis Unless 'Z' Is Used
- Output Enable Logic Emulated in LEs
- Simulation & Synthesis Do Not Match



## Pinos Bidirecionais

```
module module_name (  
    ...,  
    inout bidi,  
    ...  
);  
  
wire incoming_signal;  
  
assign incoming_signal = bidi;  
  
logic operating on incoming_signal  
logic generating outgoing_signal  
  
assign bidi = (oe ? outgoing_signal : 1'bZ);
```

- Declare Pin As Direction ***inout***
- Use ***inout*** As Both Input & Tri-State Output
- For registered bidirectional I/O, use separate process to infer registers

*bidi as an input*

*bidi as an tri-stated output*



## Memory

Synthesis tools have different capabilities for recognizing memories

Synthesis tools are sensitive to certain coding styles in order to recognize memories

- Usually described in the tool documentation (e.g. Quartus II Handbook)

Tools may have limitations in architecture implementation

Newer Altera devices support synchronous inputs only

- Limitations in clocking schemes
- Memory size limitations
- Read-during-write support

Must create an array variable to hold memory values



## Inferindo Single-Port Memory

```
module sp_ram_async_read (  
    output [7:0] q,  
    input [7:0] d,  
    input [6:0] addr,  
    input we, clk  
);
```

```
reg [7:0] mem [127:0];
```

```
always @(posedge clk)  
    if (we)  
        mem[addr] <= d;
```

```
assign q = mem[addr];
```

```
endmodule
```


Memory array

- Code describes a **128 x 8 RAM** with synchronous write & asynchronous read
- **Cannot be implemented in Altera embedded RAM due to asynchronous read**
  - Uses general logic and registers



## Inferindo Single-Port Memory

```
module sp_ram_sync_rdwo (  
    output reg [7:0] q,  
    input [7:0] d,  
    input [6:0] addr,  
    input we, clk  
);  
  
reg [7:0] mem [0:127];  
  
always @(posedge clk)  
begin  
    if (we)  
        mem[addr] <= d;  
    q <= mem[addr];  
end  
  
endmodule
```



- Code describes a **128 x 8 RAM** with synchronous write & synchronous read
- Old data read-during-write behaviour
  - Memory read in same process/cycle as memory write using **non-blocking** statements
  - Check target architecture for support as unsupported features built using LUTs/registers

Recommendation: Read Quartus II Handbook, Volume 1, Chapter 6 for more information on inferring memories and read during write behavior





## Inferindo Single-Port Memory

```
module sp_ram_sync_rdwn (  
    output reg [7:0] q,  
    input [7:0] d,  
    input [6:0] addr,  
    input we, clk  
);  
  
reg [7:0] mem [0:127];  
  
always @(posedge clk)  
begin  
    if (we)  
        mem[addr] = d; // Blocking  
        q = mem[addr]; // Blocking  
end  
  
endmodule
```

- Same memory with new data read-during-write behaviour
  - Read performed in process/cycle using **blocking** statements
- Check target architecture for support
- Use `ramstyle` attribute set to `"no_rw_check"` to disable checking and prevent extra logic generation

Recommendation: Read Quartus II Handbook, Volume 1, Chapter 6 for more information on inferring memories and read during write behavior



## Inferindo Dual-Port Memory

```
module sdp_sc_ram (  
    output reg [7:0] q,  
    input [7:0] d,  
    input [6:0] wr_addr, rd_addr,  
    input we, clk  
);  
  
reg [7:0] mem [0:127];  
  
always @(posedge clk) begin  
    if (we)  
        mem[wr_addr] <= d;  
    q <= mem[rd_addr];  
end  
  
endmodule
```

- Code describes a **simple dual-port** (separate read & write addresses) 64 x 8 RAM with single clock
- Code implies old data read-during-write behaviour
  - New data support in simple dual-port requires additional RAM bypass logic



## Inferindo Dual-Port Memory

```
module dp_dc_ram (  
    output reg [7:0] q_a, q_b,  
    input [7:0] data_a, data_b,  
    input [6:0] addr_a, addr_b,  
    input clk_a, clk_b, we_a, we_b  
);
```

```
reg [7:0] mem [0:127];
```

```
always @(posedge clk_a)  
begin  
    if (we_a)  
        mem[addr_a] <= data_a;  
    q_a <= mem[addr_a];  
end
```

```
always @(posedge clk_b)  
begin  
    if (we_b)  
        mem[addr_b] <= data_b;  
    q_b <= mem[addr_b];  
end  
endmodule
```

- Code describes a true dual-port (two individual addresses) 64 x 8 RAM with dual clocks
- May not be supported in all synthesis tools
- Old data same-port read-during-write behaviour shown
  - Supported only in Stratix III, Stratix IV and Cyclone III devices
  - Other device families support new data (blocking assignments)
- Mixed port behaviour undefined with multiple clocks



## Inicializando o conteúdo da memória

```
module ram_init (  
    output reg [7:0] q,  
    input [7:0] d,  
    input [6:0] wr_addr, rd_addr,  
    input we, clk  
);  
  
reg [7:0] mem [0:127];  
  
initial  
    $readmemh("ram.dat", mem);  
  
always @(posedge clk) begin  
    if (we)  
        mem[wr_addr] <= d;  
    q <= mem[rd_addr];  
end  
  
endmodule
```

- Use *\$readmemb* or *\$readmemh* system tasks to assign initial contents to inferred memory
- Initialization data stored in *.dat* file converted to **.MIF** (Altera memory initialization file)
- Contents of *.MIF* downloaded into FPGA during configuration
- Alternate: Use an initial block and loop to assign values to array address locations



## Usando Tasks para Inicializar memória

Each number in file given separate address

- White spaces and comments separate numbers

Examples

- \$readmemb (“<file\_name>”, <memory\_name>);
  - \$readmemb (“<file\_name>”, <memory\_name>, <mem\_start\_addr>);
  - \$readmemb (“<file\_name>”, <memory\_name>, <mem\_start\_addr>, <mem\_finish\_addr>);
- <mem\_start\_addr> and <mem\_finish\_addr> are optional

Indicate beginning and ending memory addresses in which data will be loaded

Addresses can be ascending or descending

- Defaults are the start index of the memory array and the end of the data file or memory array
- Uninitialized addresses will be don't care's (X's) for synthesis and will be loaded with 0's by Quartus II software

*ram.dat*

```
0000_0000
0000_0101
0000_1010
0000_1111
0001_0100
0001_1001
0001_1110
0010_0011
0010_1000
// Repeat segment at
// address 20 hex
@20
0000_0000
0000_0101
0000_1010
0000_1111
0001_0100
0001_1001
0001_1110
0010_0011
0010_1000
```



## Sinais de controle não suportados

```
module ram_unsupported (  
    output reg [7:0] q,  
    input [7:0] d,  
    input [6:0] addr,  
    input we, clk  
);  
  
reg [7:0] mem [0:127];  
  
always @(posedge clk, negedge aclr_n)  
begin  
    if (!aclr_n) begin  
        mem[addr] <= 0;  
        q <= 0;  
    end  
    else if (we) begin  
        mem[addr] <= d;  
        q <= mem[addr];  
    end  
end  
  
endmodule
```

- *Memory content cannot be cleared with reset*
- *Synthesizes to logic*
- *Recommendations*
  1. *Avoid reset checking in RAM read or write processes*
  2. *Be wary of other control signals (i.e. clock enable) until validated with target architecture*



## Inferindo ROM (case)

```
reg [6:0] q;  
  
always @ (posedge clk)  
begin  
    case (addr)  
        6'b000000: q <= 8'b0111111;  
        6'b000001: q <= 8'b0011000;  
        6'b000010: q <= 8'b1101101;  
        6'b000011: q <= 8'b1111100;  
        6'b000100: q <= 8'b1011010;  
        6'b000101: q <= 8'b1110110;  
        ***  
        6'b111101: q <= 8'b1110111;  
        6'b111110: q <= 8'b0011100;  
        6'b111111: q <= 8'b1111111;  
    end case  
end
```

- Automatically converted to ROM
  - Tools generate ROM using embedded RAM & initialization file
- Requires constant explicitly defined for each choice in CASE statement
- May use **romstyle** synthesis attribute to control implementation
- Like RAMs, address or output must be registered to implement in Altera embedded RAM



## Inferindo ROM (Memory File)

```
module dp_rom (  
    output reg [7:0] q_a, q_b,  
    input [6:0] addr_a, addr_b,  
    input we, clk  
);  
  
reg [7:0] mem [0:127];  
  
initial  
    $readmemh("ram.dat", mem);  
  
always @ (posedge clk)  
begin  
    q_a <= mem[addr_a];  
    q_b <= mem[addr_b];  
end  
  
endmodule
```

- *Using \$readmemb or \$readmemh to initialize ram contents*
- *No write control*
- *Example shows dual-port access*
- *Automatically converted to ROM*
- *Tools generate ROM using embedded RAM & initialization file*





## Codificação de Maquinas de Estados

Parameters or local parameters\* used to define states

- Parameter “values” are replaced with state encoding values as chosen by synthesis tool

Use options/constraints in synthesis tool to control encoding style (e.g. binary, one-hot, safe, etc.)

- `define statements also supported, but not recommended

Registers are used to store state

```
reg [2:0] current_state, next_state;
```

Do not bit-slice current\_state or next\_state (e.g. state[1:0])

Separate sequential process from combinational process

Sequential process should always include synchronous or asynchronous reset

Use case statement to do the next-state logic, instead of if-else statement

- Some synthesis tools do not recognize if-else statements for implementing state machines



## Codificação de Maquinas de Estados

**Tools Menu ⇒ State Machine Viewer**

**State Flow Diagram**

**Use Drop-Down to Select State Machine**

**Highlighting State in State Transition Table Highlights Corresponding State in State Flow Diagram**

**State Transition/Encoding Table**

	Source State	Destination State	Condition
1	tap3	tap4	
2	tap2	tap3	
3	tap1	tap2	
4	tap4	tap1	(next)
5	tap4	idle	(next)
6	idle	tap1	(next)
7	idle	idle	(next)

Transitions / Encoding



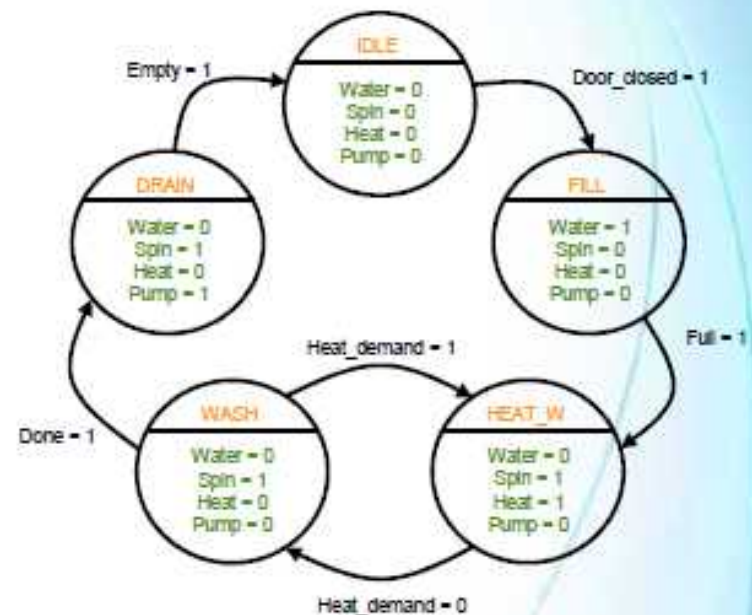
## Declaração de Estados

```
module state_machine (  
    input clk, reset, door_closed, full,  
    input heat_demand, done, empty,  
    output reg water, spin, heat, pump  
);
```

```
reg [2:0] current_state, next_state;
```

```
parameter idle=0, fill=1, heat_w=2, wash=3,  
    drain=4;
```

States



State Registers



## Próximo Estado lógico

//State transitions

```
always @(posedge clk)
```

```
if (reset)
```

```
current_state <= idle;
```

```
else
```

```
current_state <= next_state;
```

//Next State logic

```
always @ *
```

```
begin
```

```
next_state = current_state; //default condition
```

```
case (current_state)
```

```
idle: if (door_closed) next_state = fill;
```

```
fill: if (full) next_state = heat_w;
```

```
heat_w: if (heat_demand) next_state = wash;
```

```
wash: begin
```

```
if (heat_demand) next_state = heat_w;
```

```
if (done) next_state = drain;
```

```
end
```

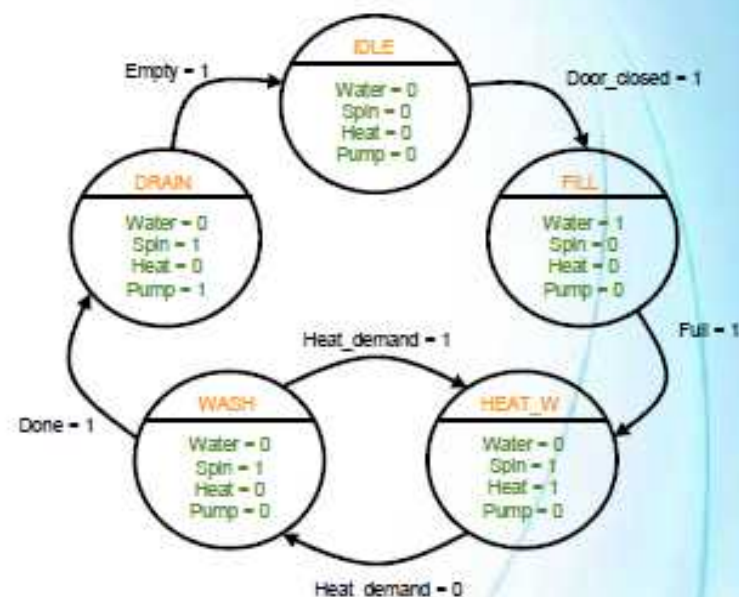
```
drain: if (empty) next_state = idle;
```

```
endcase
```

```
end
```

*State transitions*

*Next state logic*







## Saídas Combinacionais com Moore

```
//output logic
```

```
always @*
```

```
begin
```

```
  water = 0;
```

```
  spin = 0;
```

```
  heat = 0;
```

```
  pump = 0;
```

```
  case (current_state)
```

```
    idle:
```

```
      ;
```

```
    fill:
```

```
      water=1;
```

```
    heat_w:
```

```
      begin spin =1; heat = 1; end
```

```
    wash:
```

```
      spin =1;
```

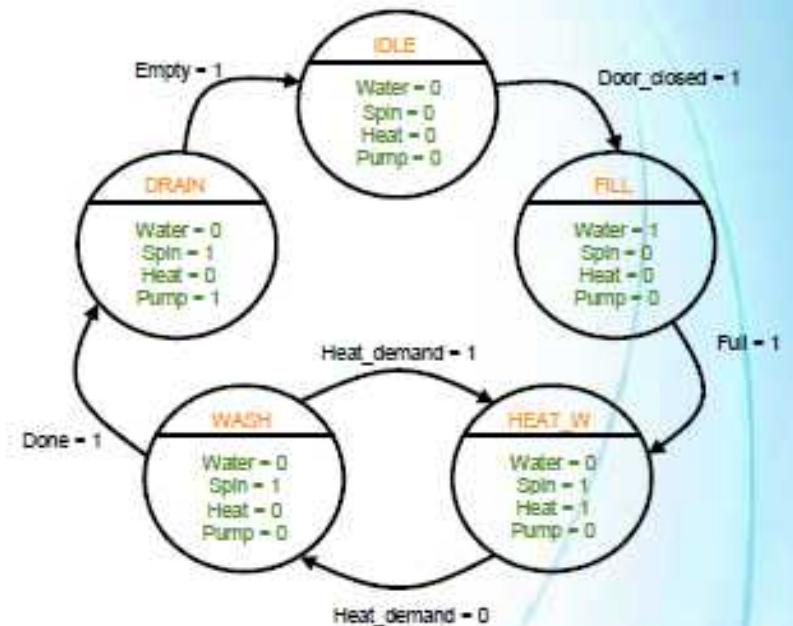
```
    drain:
```

```
      begin spin =1; pump=1; end
```

```
  endcase
```

```
end
```

*Default output conditions*

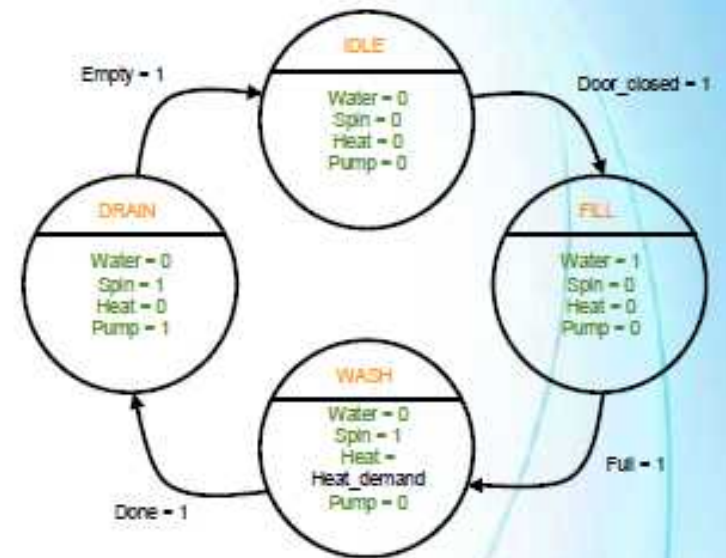


– Output logic function of state only



## Saídas Combinacionais com Mealy

```
//output logic
always @ *
begin
    water=0;
    spin=0;
    heat=0;
    pump=0;
    case (current_state)
        idle:
            ;
        fill:
            water=1;
        wash:
            begin spin =1; heat = heat_demand; end
        drain:
            begin spin =1; pump=1; end
    endcase
end
```



– Output logic function of state only and input(s)



## Estilos de codificação de máquinas de Estados

State	Binary Encoding	Grey-Code Encoding	One-Hot Encoding	Custom Encoding
Idle	000	000	00001	?
Fill	001	001	00010	?
Heat_w	010	011	00100	?
Wash	011	010	01000	?
Drain	100	110	10000	?

Quartus II default encoding styles for Altera devices

- One-hot encoding for look-up table (LUT) devices

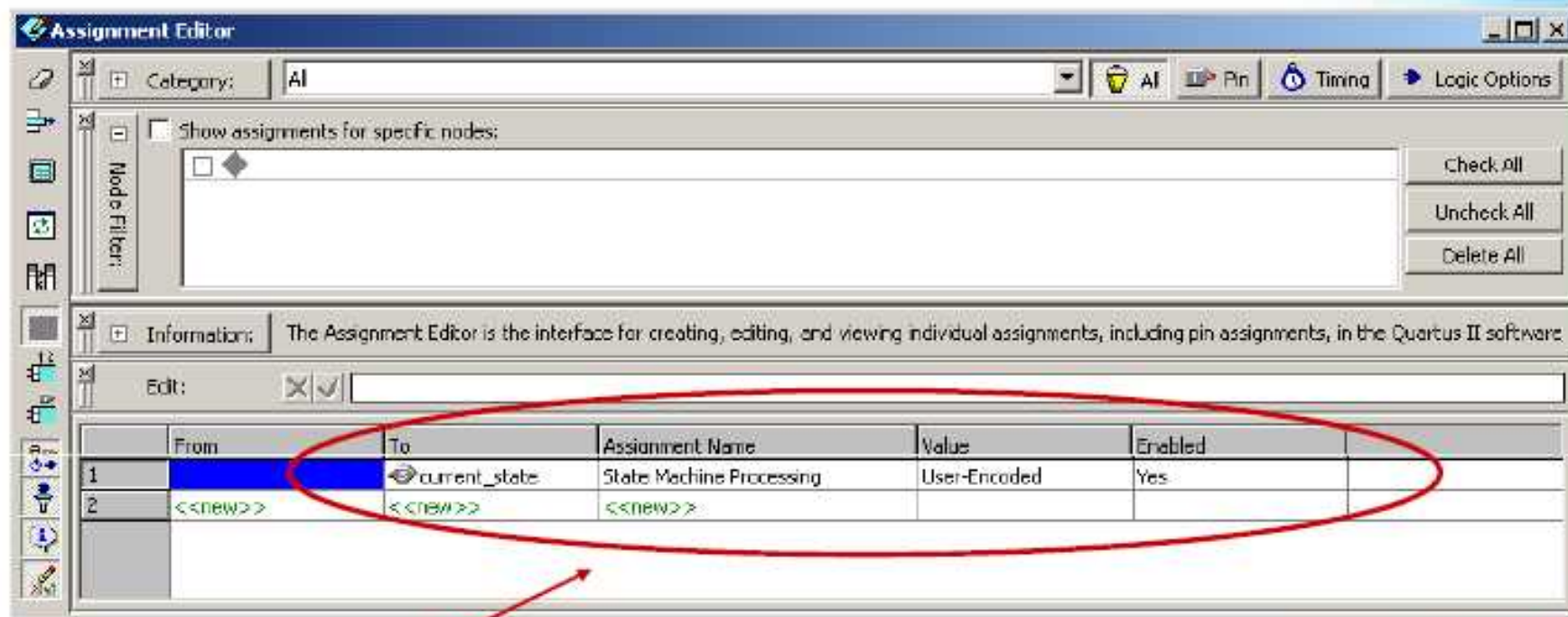
Architecture features lesser fan-in per cell and an abundance of registers

- Binary (minimal bit) or grey-code encoding for product-term devices

Architecture features fewer registers and greater fan-in



## Quartus II- Controle do estilo de codificação



**Apply Assignment to  
State Variable**

Options:

- One-Hot
- Gray
- Minimal Bits
- Sequential
- User-Encoded
- Johnson





## Estados Indefinidos

Noise and spurious events in hardware can cause state machines to enter undefined states

If state machines do not consider undefined states, it can cause mysterious “lock-ups” in hardware

Good engineering practice is to consider these states

To account for undefined states

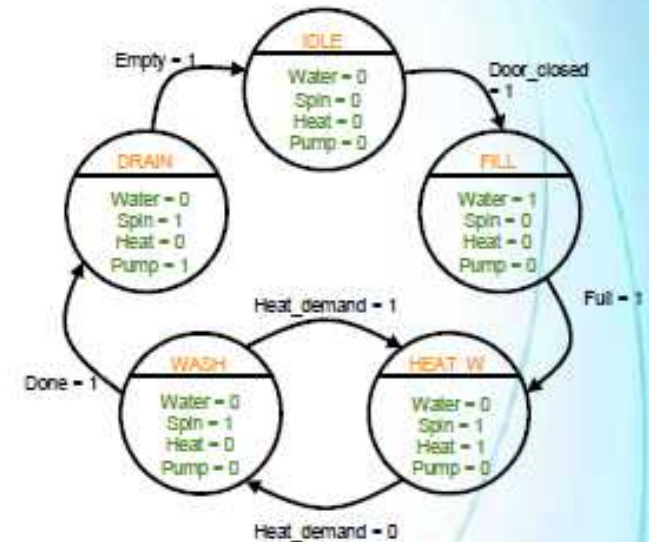
- Explicitly code for them (manual)
- Use “safe” synthesis constraint (automatic)



## Máquina de Estados segura??

```
//State transitions
always @(posedge clk)
    current_state <= next_state;

//Next State logic
always @*
begin
    next_state = current_state; //default condition
    case (current_state)
        idle:      if (door_closed) next_state = fill;
        fill:      if (full) next_state = heat_w;
        heat_w:    if (heat_demand==0) next_state= wash;
        wash:      begin
                    if (heat_demand) next_state=heat_w;
                    if (done) next_state=drain;
                end
        drain:     if (empty) next_state=idle;
        default:   next_state = idle;
    endcase
end
```

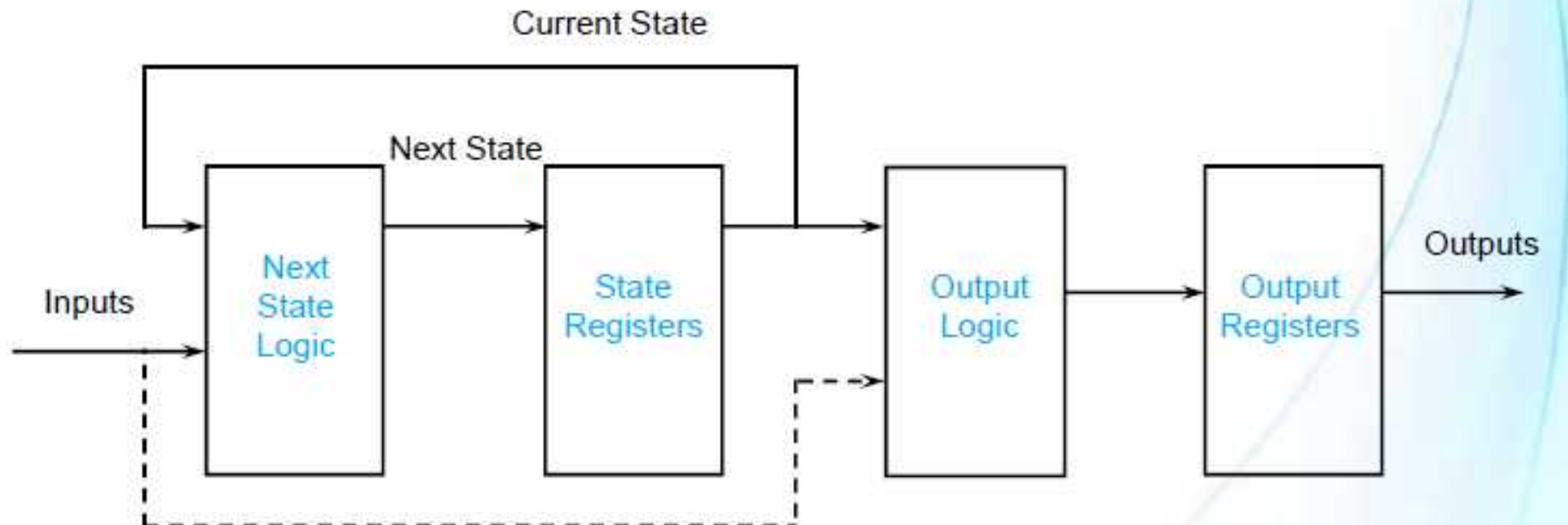


– Using default clause to cover all undeclared states



## Saídas bufferizadas

Remove glitches by adding output registers  
- Adds a stage of latency

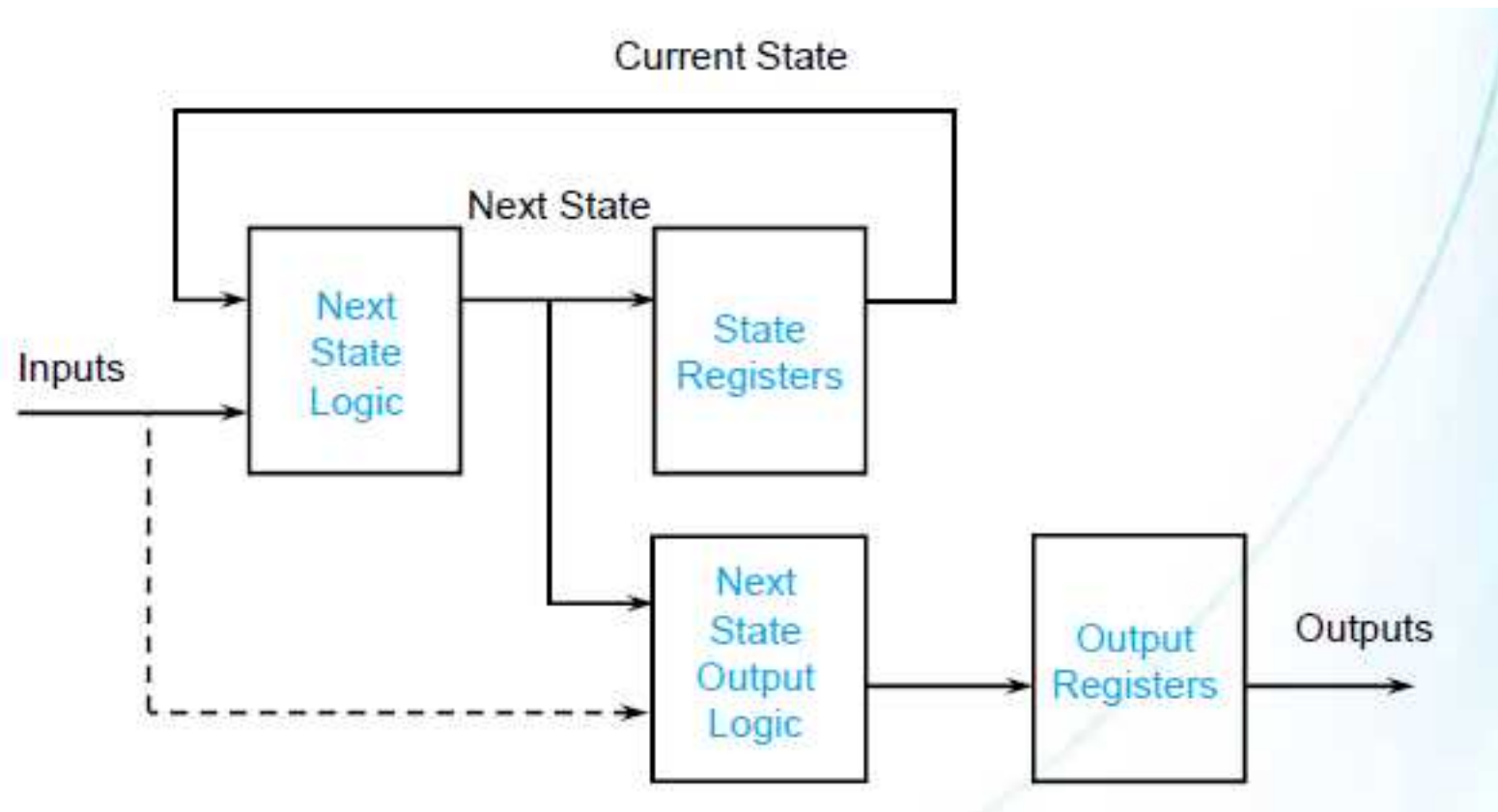




## Saídas bufferizadas com/sem Latência

Base outputs on next state vs. current state

- Output logic uses next state to determine what the next outputs will be
- On next rising edge, outputs change along with state registers





## Saídas bufferizadas com/sem Latência

```
//output logic
```

```
always @ (posedge clk)
```

```
begin
```

```
  water = 0;
```

```
  spin = 0;
```

```
  heat = 0;
```

```
  pump = 0;
```

```
  case (next_state)
```

```
    idle:
```

```
    fill:      water <= 1;
```

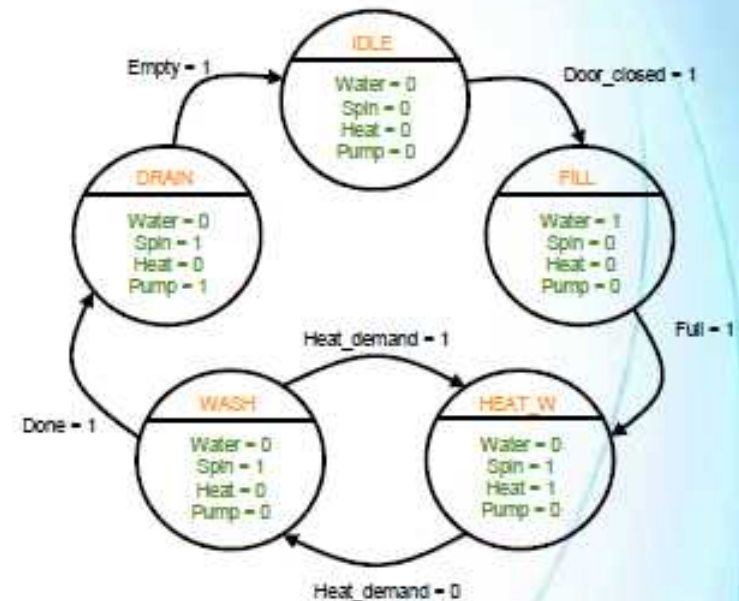
```
    heat_w:    begin spin <= 1; heat <= 1; end
```

```
    wash:      spin <= 1;
```

```
    drain:     begin spin <= 1; pump <= 1; end
```

```
  endcase
```

```
end
```



- Base output logic case statement on next state variable (instead of current state variable)
- Wrap output logic with a clocked process



## Usando Estilos de codificação customizadas

Remove glitches without output registers

Eliminate combinatorial output logic

Outputs mimic state bits

- Use additional state bits for states that do have exclusive outputs

State	Outputs	Custom Encoding
	Water Spin Heat Pump	
Idle	0 0 0 0	0000
Fill	1 0 0 0	1000
Heat_w	0 1 1 0	0110
Wash	0 1 0 0	0100
Drain	0 1 0 1	0101



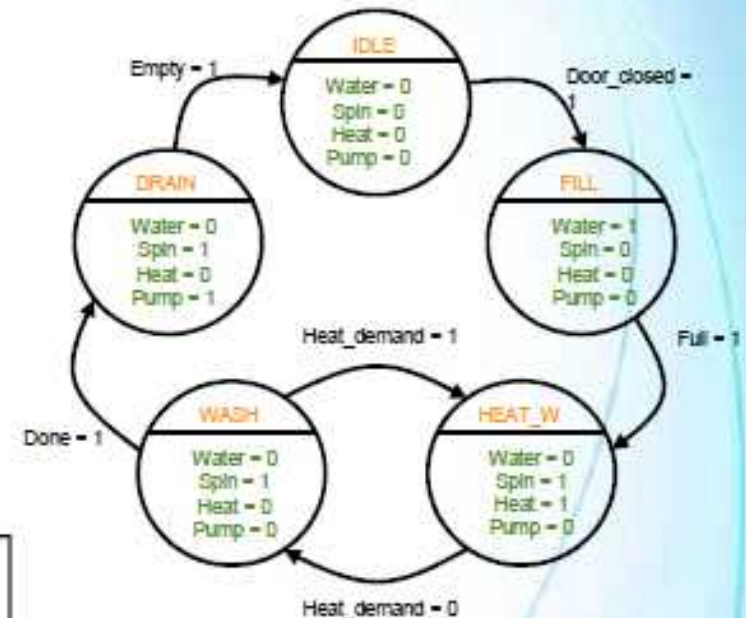


## Estados com codificação customizadas

```
reg [3:0] current_state, next_state;  
  
parameter idle = 'b0000, fill = 'b1000,  
            heat_w = 'b0110, wash = 'b0100,  
            drain = 'b0101;
```

*Custom Encoding*

- Must also set **State Machine Processing** assignment to “**User Encoded**”
- Output assignments are coded per previous examples (slides 83 or 84)
  - Synthesis automatically handles reduction of output logic
- Some tools use synthesis attributes like **enum\_encoding** OR **syn\_enum\_encoding** to perform custom state encoding





## Escrevendo Máquinas de Estados eficientes

Remove counting, timing, arithmetic functions  
from state machine & implement externally  
Reduces overall logic & improves performance





## Melhorando a utilização de Lógica & Performance

- Balancing Operators
- Resource Sharing
- Logic Duplication
- Pipelining



## Operadores

Synthesis tools replace operators with pre-defined  
(preoptimized)  
blocks of logic

Designer should control when & how many operators  
- Ex. Dividers

Dividers are large blocks of logic

Every '/' and '%' inserts a divider block and leaves it up to  
synthesis tool to  
optimize

Better resource optimization usually involves cleverly  
using multipliers or shift  
operations to do divide

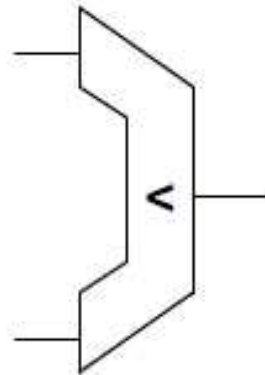
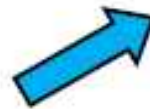


## Gerando Lógica dos Operadores

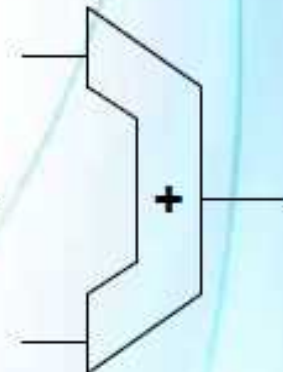
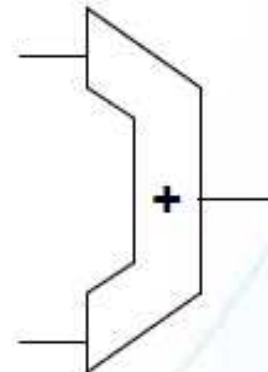
- Synthesis tools break down code into logic blocks
- They then assemble, optimize & map to hardware

```
if (sel < 10)  
    y = a + b;  
else  
    y = a + 10;
```

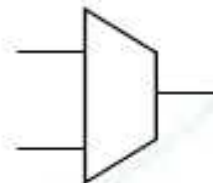
1 Comparator



2 Adders



1 Muiltplexer





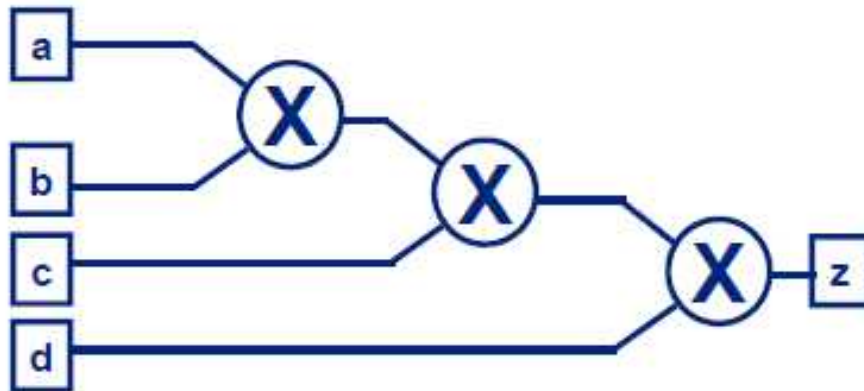
## Balanciando Operadores

Use parenthesis to define logic groupings

- Increases performance & utilization
- Balances delay from all inputs to output
- Circuit functionality unchanged

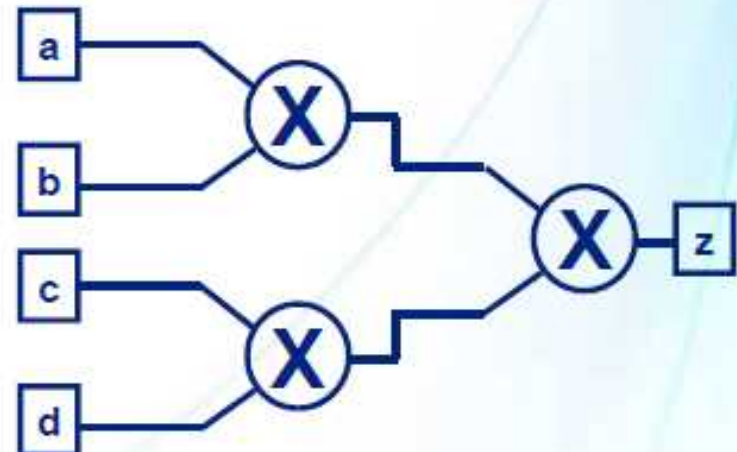
### Unbalanced

$$\text{out} = a * b * c * d;$$



### Balanced

$$\text{out} = (a * b) * (c * d);$$



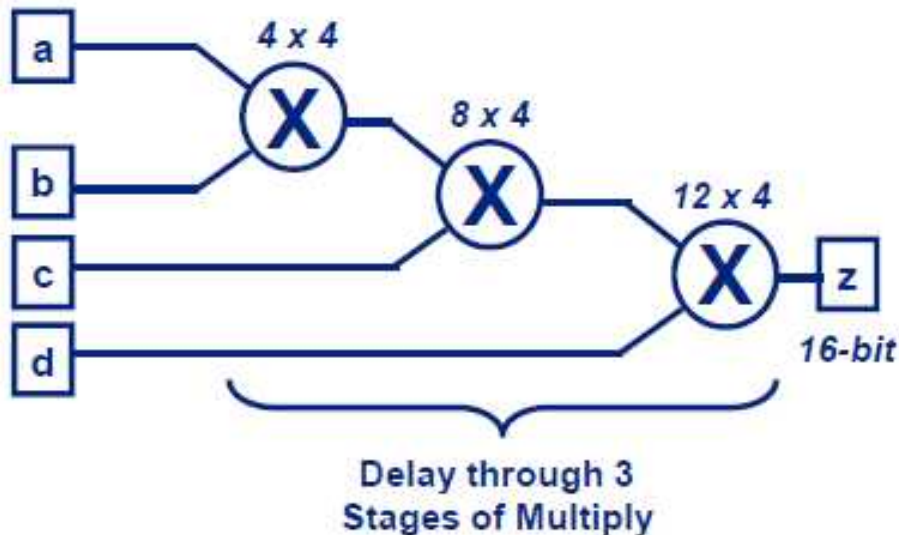


## Balanciando Operadores : Exemplo

- a, b, c, d: 4-bit vectors

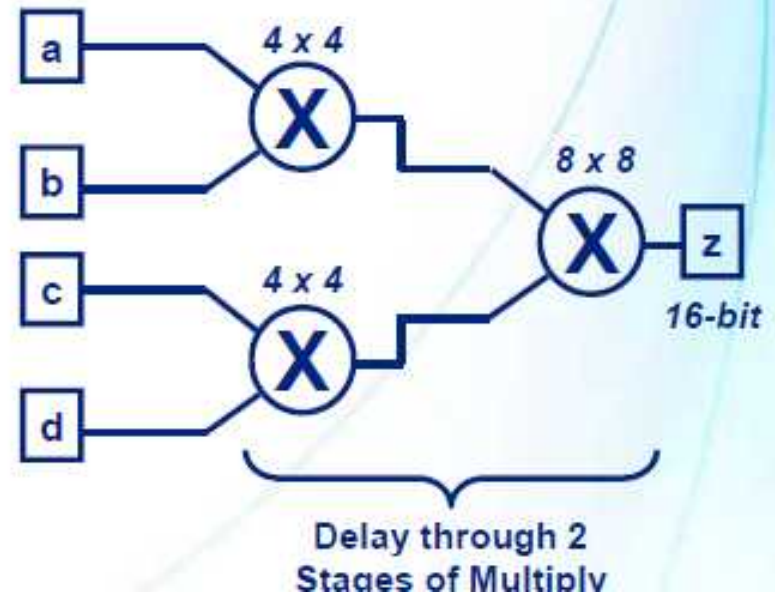
### Unbalanced

$$\text{out} = a * b * c * d$$



### Balanced

$$\text{out} = (a * b) * (c * d)$$





## Compartilhamento de recursos

Reduces number of operators needed

- Reduces area

Two types

- Sharing operators among mutually exclusive functions
- Sharing common sub-expressions

Synthesis tools can perform automatic resource sharing

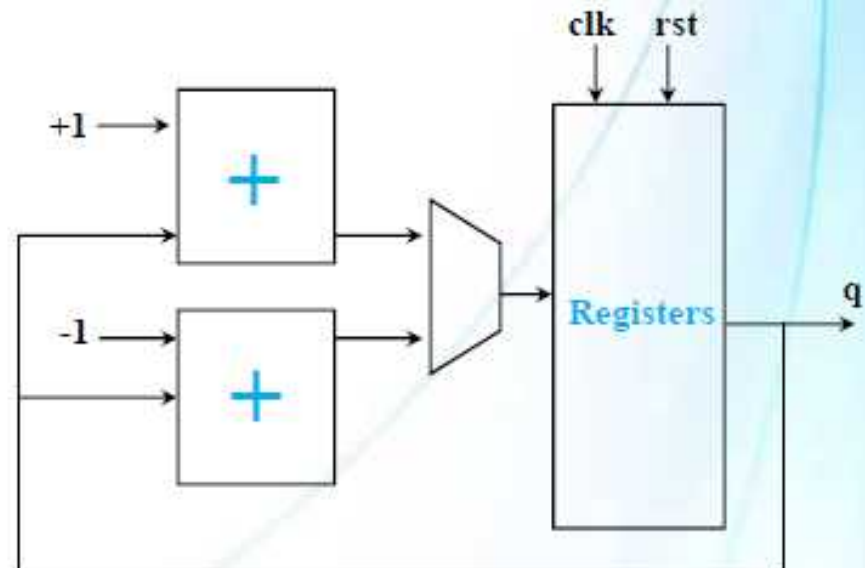
- Feature can be enabled or disabled



## Operadores mutuamente exclusivos

```
module test (  
    input rst, clk, updn,  
    output [7:0] q,  
    reg [7:0] q  
);  
  
always@(posedge clk, negedge rst)  
begin  
    if (!rst)  
        q<=0;  
    else if (updn)  
        q <= q + 1;  
    else  
        q <= q - 1;  
end  
endmodule
```

- Up/down counter
- 2 adders are mutually exclusive & can be shared (typically if-else with same operator in both choices)





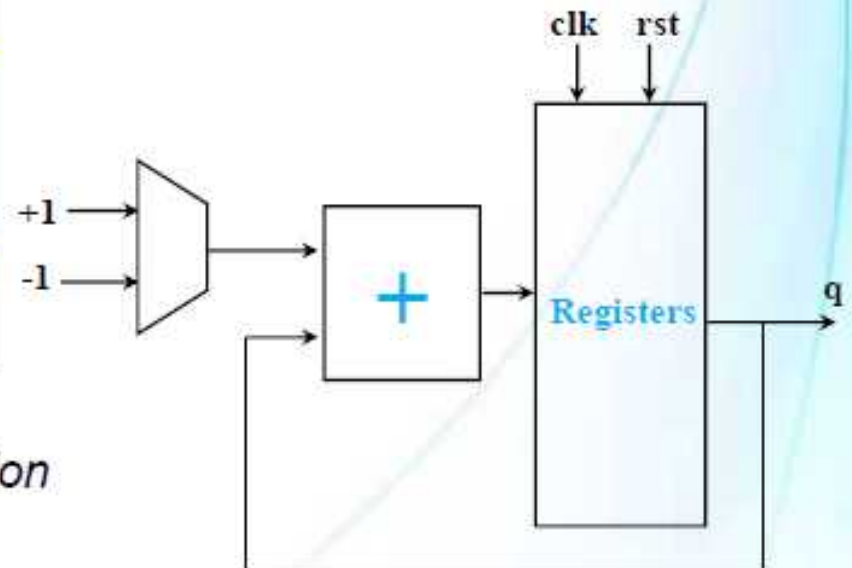


## Compartilhando recursos com operadores Mutuamente exclusivos

```
module test (  
    input rst, clk, updn,  
    output reg [7:0] q  
);  
  
always @ (posedge clk, negedge rst) begin  
    if (!rst)  
        q <= 0;  
    else  
        q <= q + ( updn ? 1 : -1);  
    end  
endmodule
```

*Single add operation*

- Up/down counter
- Only one adder required







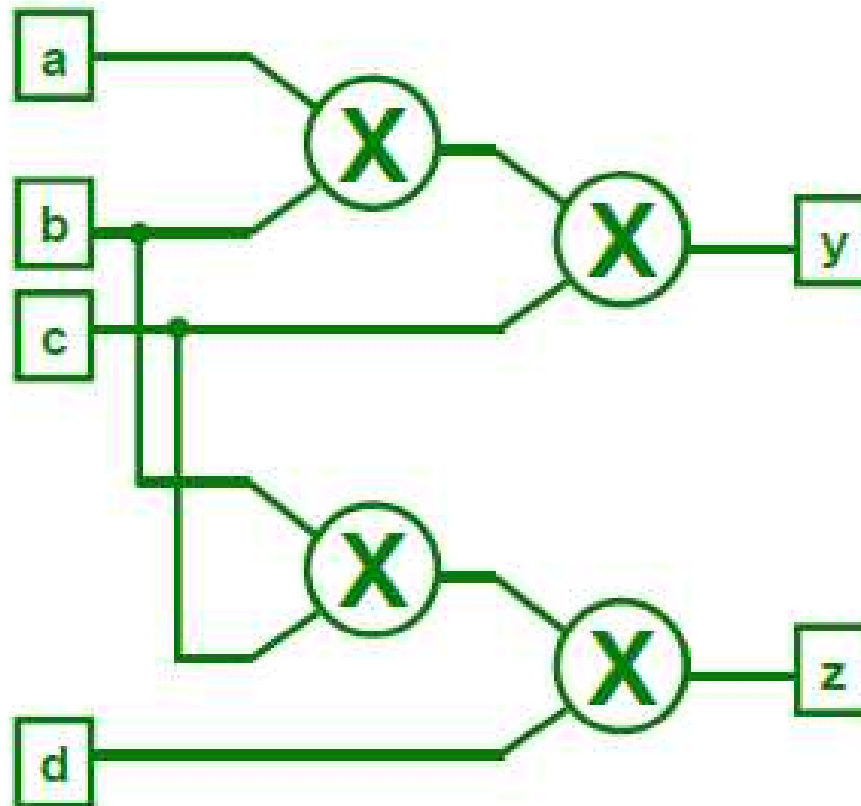
## Quantos Multiplicadores são necessários?

$$y = a * b * c$$

$$z = b * c * d$$



## Quantos Multiplicadores são necessários?





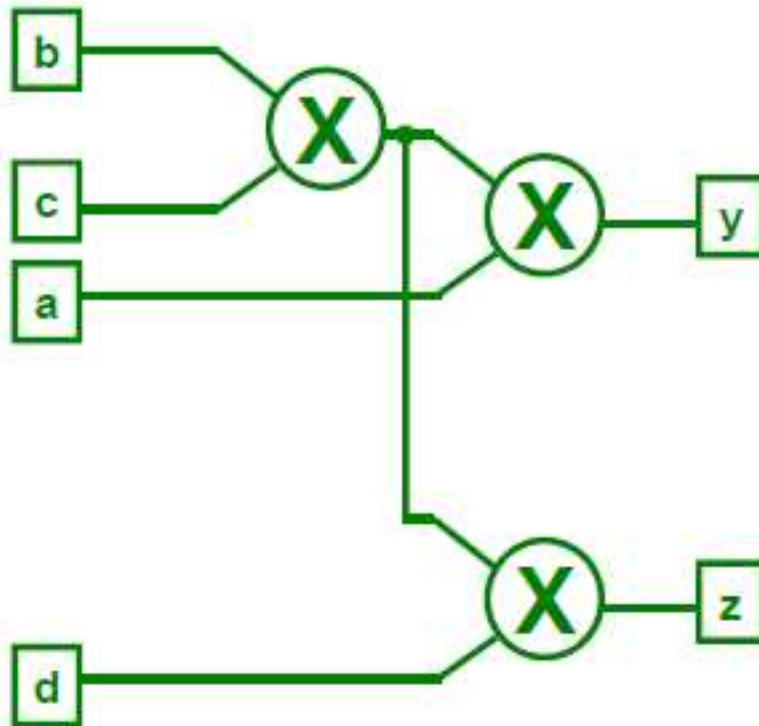
## Quantos Multiplicadores são necessários?

$$y = a * (b * c)$$

$$z = (b * c) * d$$



## Quantos Multiplicadores são necessários?



3 Multipliers!

- This is called sharing common subexpressions
- Some synthesis tools do this automatically, but some don't!
- Parentheses guide synthesis tools
- If  $(b*c)$  is used repeatedly, assign to temporary signal



## Duplicação de Lógica

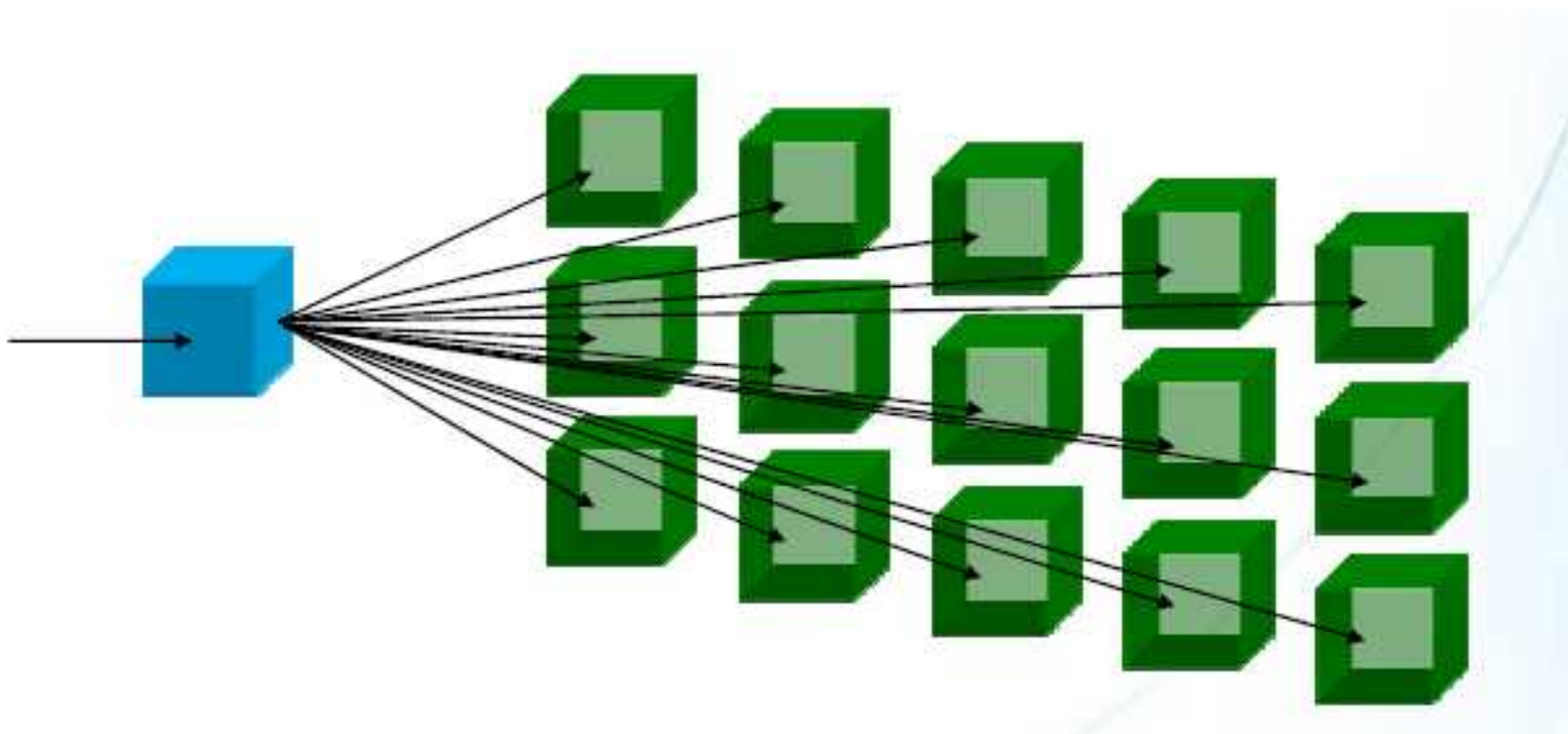
Intentional duplication of logic to reduce fan-out  
Synthesis tools can perform automatically  
- User sets maximum fan-out of a node



## Problemas de Fan-Out

High fan-out increases placement difficulty

- High fan-out node cannot be placed close to all destinations
- Ex: Fan-out of 1 & 15

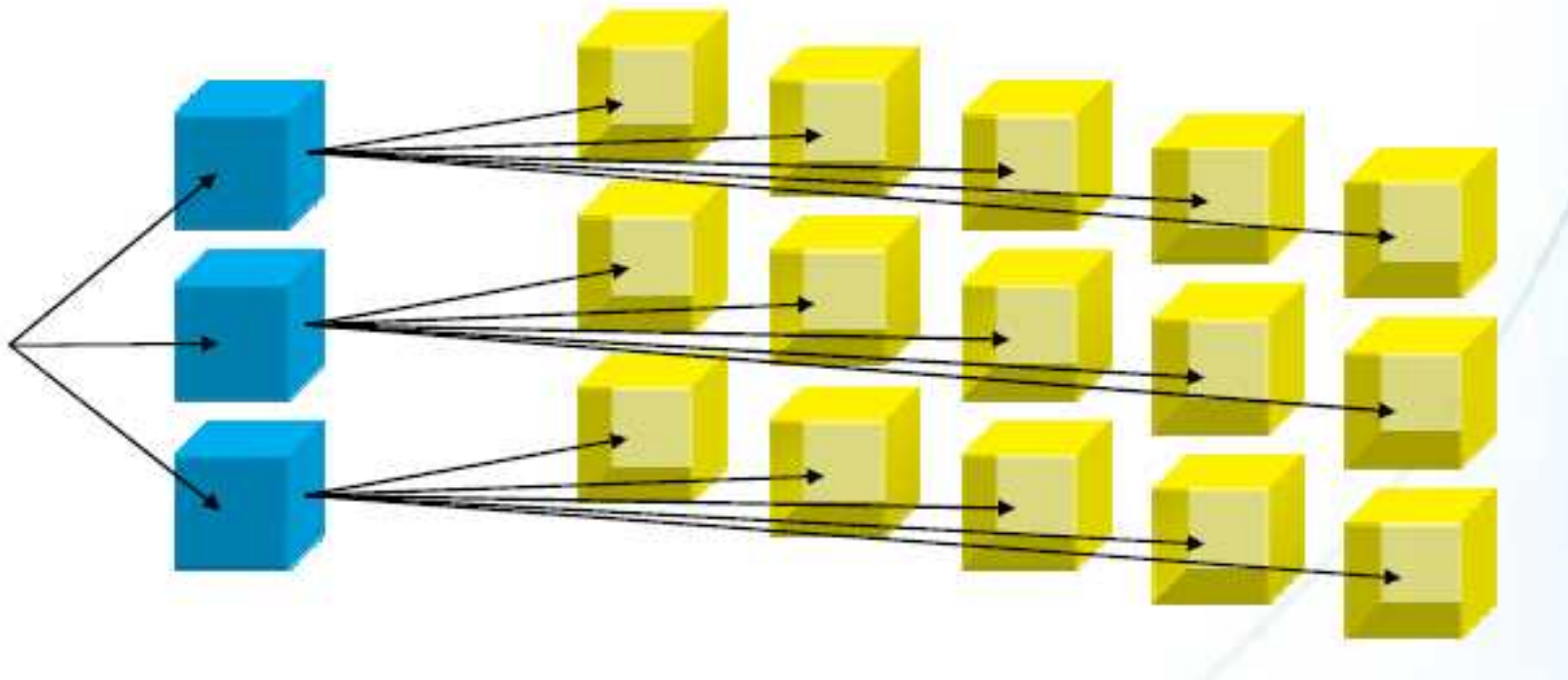




## Controlando o Fan-Out

By replicating logic fan-out can be reduced

- Worst case path now contains fan-out 3 and 5

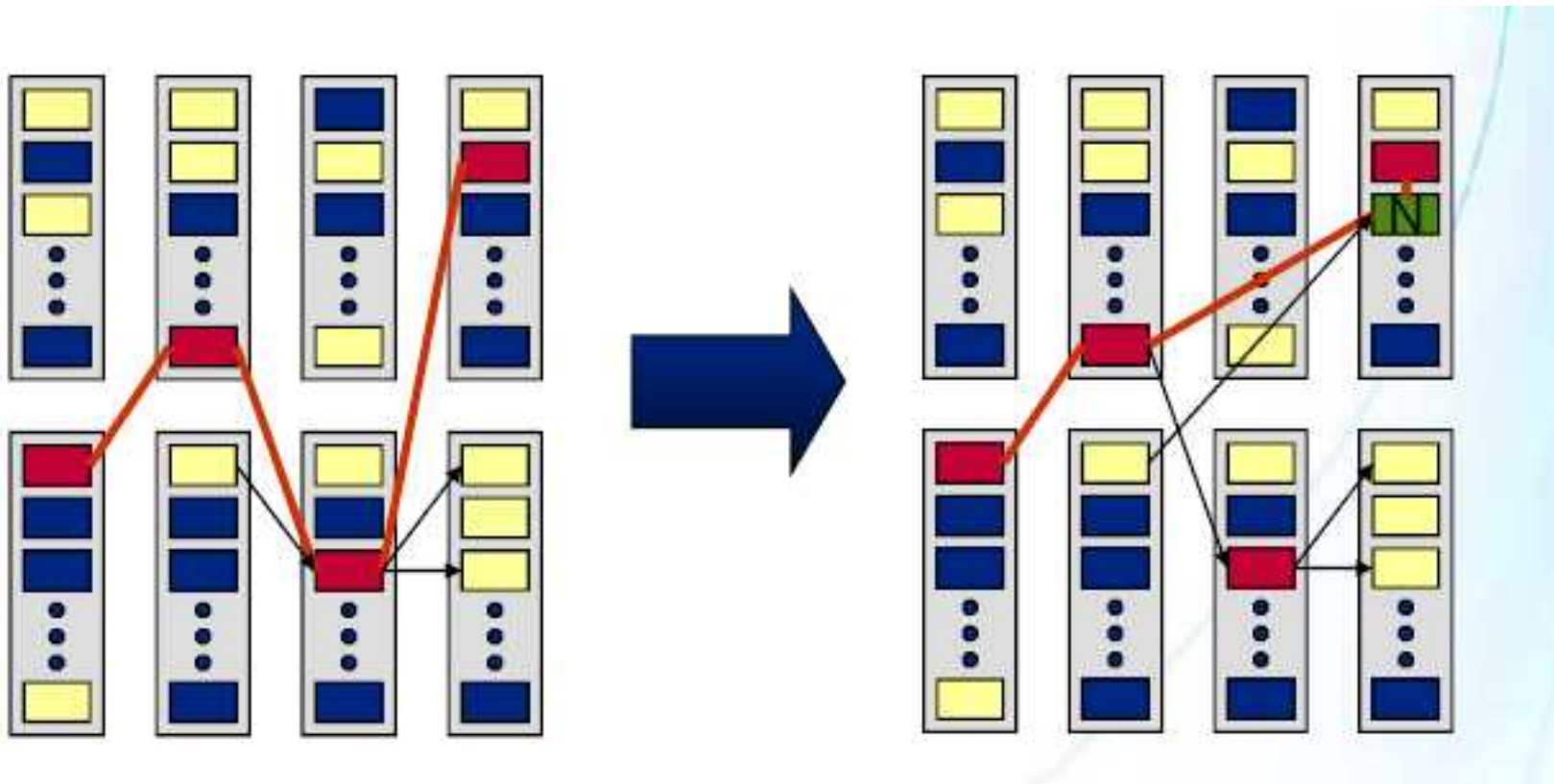






## Exemplo de duplicação de Lógica

- High fan-out node duplicated & placed to reduce delay







## Controle automático de Fan-out

Most synthesis tools feature options which limit fan-out

Advantage: Easy experimentation

Disadvantage: Less control over results

- Knowing which nodes have high fan-out & their destination helps floor-planning



## Controle automático Quartus

The screenshot shows the Quartus Assignment Editor window. The left pane lists various assignment categories, with 'Locations' selected. The right pane shows the 'Maximum Fan-Out' assignment. A red circle highlights the 'Maximum Fan-Out' assignment in the table, and a red arrow points from the text 'Select Signal Details' to the 'From' column of the table.

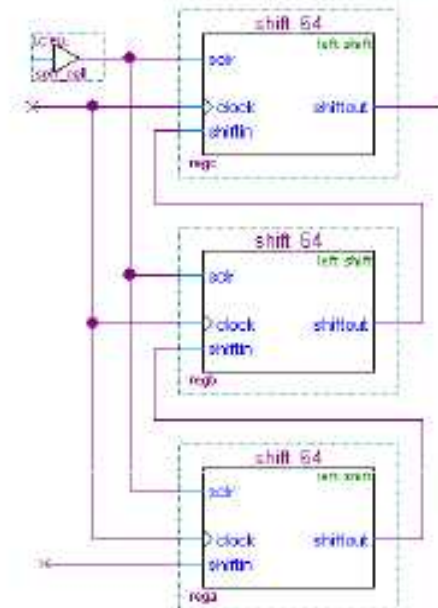
	From	To	Assignment Name	Value	Enabled
1		start	Maximum Fan-Out	50	Yes
2	<new>	<new>	<new>	<new>	<new>

Select Signal Details



## Exemplo : Shift Register

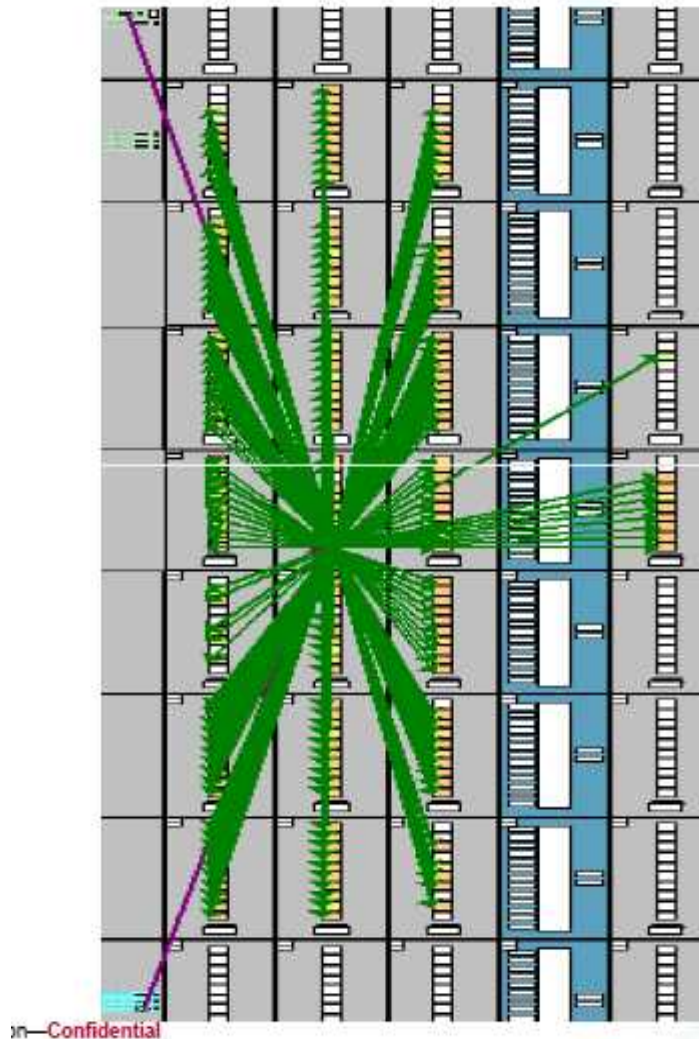
```
module test (  
    input clk, sclr_in, shiftin,  
    output shiftout  
);  
reg [63:0] rega, regb, regc;  
reg sclr;  
always @(posedge clk)  
    sclr <= sclr_in;  
always @(posedge clk) begin  
    if (sclr)  
        regc <= 0;  
    else  
        regc <= {regc[62:0], regb[63]};  
    if (sclr)  
        regb <= 0;  
    else  
        regb <= {regb[62:0], rega[63]};  
    if (sclr)  
        rega <= 0;  
    else  
        rega <= {rega[62:0], shiftin};  
end  
assign shiftout = regc[63];  
endmodule
```



- *sclr fans out to each DFF within 3 64 bit shift registers*
- *The shift registers are cascaded to produce one 192 bit shift register*
- *sclr provides a synchronous clear function*



## Fan-Out para 192 Registradores

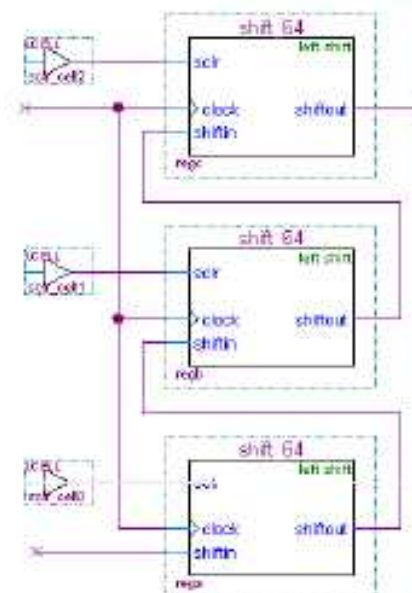






## Exemplo : Shift Register com Fan-Out Reduzido

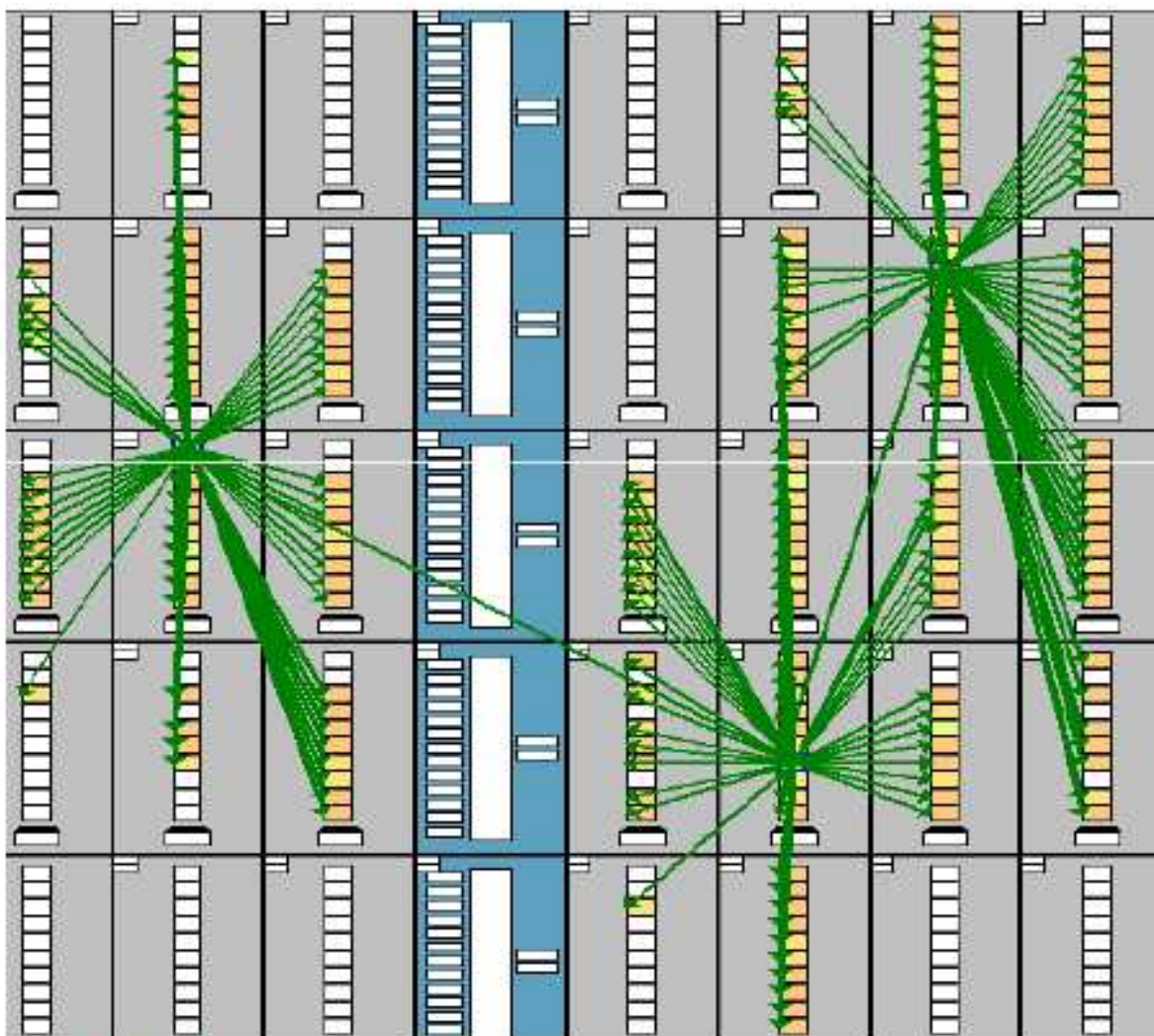
```
always @(posedge clk) begin
  if (sclr2)
    regc <= 0;
  else
    regc <= {regc[62:0], regb[63]};
  if (sclr1)
    regb <= 0;
  else
    regb <= {regb[62:0], rega[63]};
  if (sclr0)
    rega <= 0;
  else
    rega <= {rega[62:0], shiftin};
end
assign shiftout = regc[63];
```



- *sclr is replicated so that it appears 3 times*
- *Fan-out from the previous cell has gone from 1 to 3 but this is insignificant*



## Fan-Out para 64 Registradores







## Pipelining

Purposefully inserting register(s) into middle of combinatorial data (critical) path

Increases clocking speed

Adds levels of latency

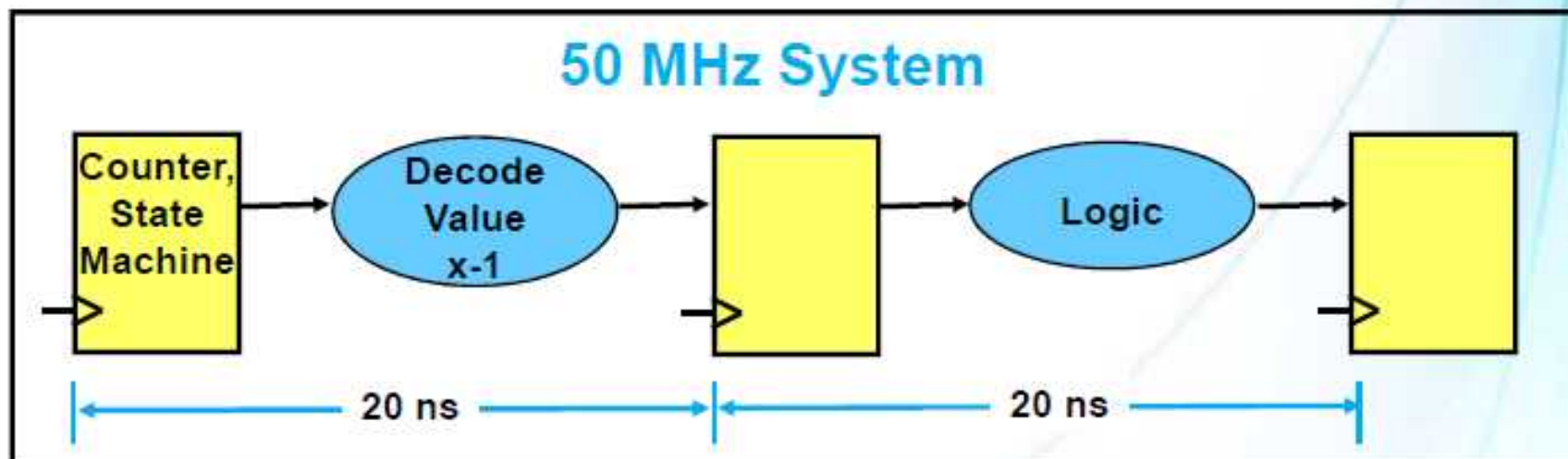
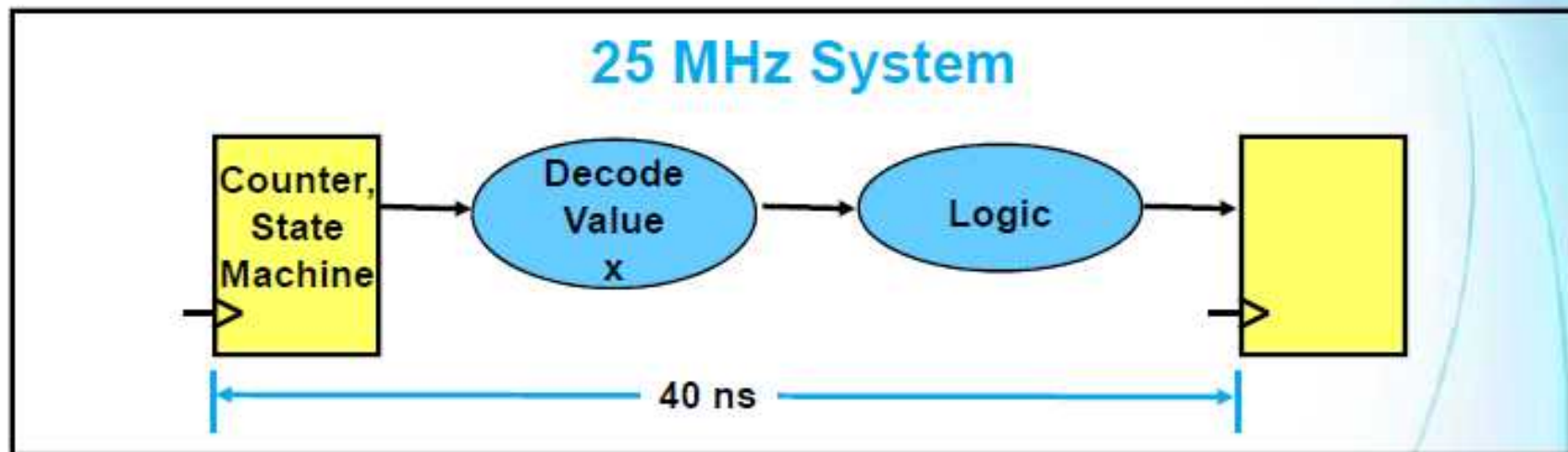
- More clock cycles needed to obtain output

Some tools perform automatic pipelining

- Same advantages/disadvantages as automatic fan-out



## Adicionando um nível de Pipeline





# Adicionando um nível de Pipeline com Verilog

## Non-Pipelined

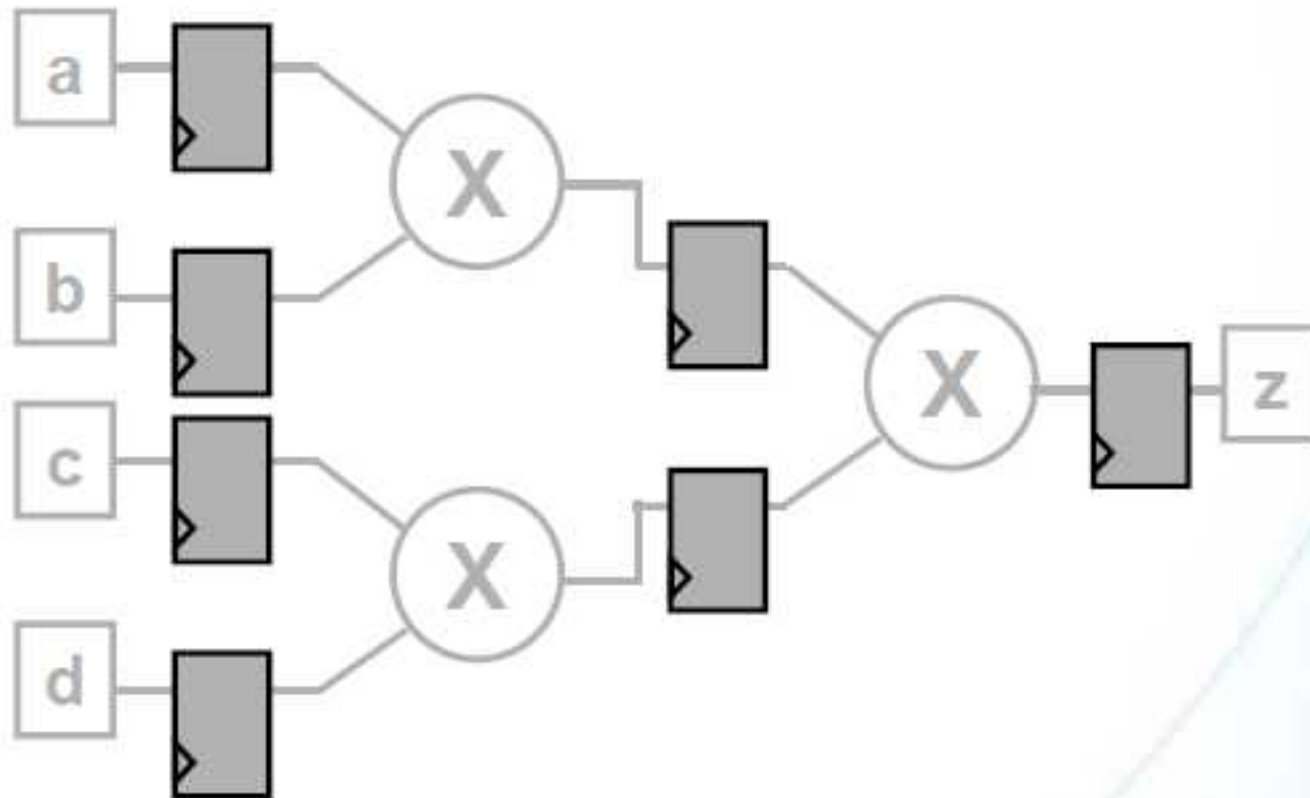
```
module test (  
    input clk, clr_n,  
    input [7:0] a, b, c, d,  
    output reg [31:0] result  
);  
reg [7:0] atemp, btemp, ctemp, dtemp;  
  
always @(posedge clk, negedge clr_n)  
begin  
    if (!clr_n) begin  
        atemp <= 0;  
        btemp <= 0;  
        ctemp <= 0;  
        dtemp <= 0;  
        result <= 0;  
    end else begin  
        atemp <= a;  
        btemp <= b;  
        ctemp <= c;  
        dtemp <= d;  
        result <= (atemp * btemp) * (ctemp * dtemp);  
    end  
end  
endmodule
```

## Pipelined

```
module test (  
    input clk, clr_n,  
    input [7:0] a, b, c, d,  
    output reg [31:0] result  
);  
reg [7:0] atemp, btemp, ctemp, dtemp;  
reg [15:0] int1, int2;  
  
always @(posedge clk, negedge clr_n)  
begin  
    if (!clr_n) begin  
        atemp <= 0;  
        btemp <= 0;  
        ctemp <= 0;  
        dtemp <= 0;  
        int1 <= 0;  
        int2 <= 0;  
        result <= 0;  
    end else begin  
        atemp <= a;  
        btemp <= b;  
        ctemp <= c;  
        dtemp <= d;  
        int1 <= atemp * btemp;  
        int2 <= ctemp * dtemp;  
        result <= int1 * int2;  
    end  
end  
endmodule
```



## Multiplicador de 4 entradas com Pipeline





## Referências

- Curso oficial da Altera "Advanced Verilog Design Techniques"