Introduction to Verilog

Introduction to Verilog

- Overview
- Hierarchy – **`module`**
- Combinational circuits
  - Concurrent description (**`assign`**)
  - Built-in gates
  - Sequential description (**`always`**)
  - Signals, variables, wires, vectors
  - User defined primitives
- Sequential circuits: DFFs, latches, state machines
- Testbenches

The history of Verilog

- Verilog = **Veri**fying **Log**ic
- Developed by Gateway Design Automation in 1985 by Phil Moorby
  - Verilog was invented as simulation language
    - Use of Verilog for synthesis was a complete afterthought
  - bought by Cadence Design Systems in 1989
- Verilog opened to public in 1990
  - until that time, Verilog HDL was a proprietary language, being the property of Cadence Design Systems
  - In the late 1980's it seemed evident that designers were going to be moving away from proprietary languages like n dot, HiLo and Verilog towards the US Department of Defense standard VHDL
- Now an IEEE standard : IEEE-1364 in 1995 (revised in 2000)

A short introduction in Verilog

- We assume you are already familiar with digital electronics and to some extent with VHDL
- The slides concentrate on the Verilog constructs, and are not intended to explain the various basic digital circuits

module
input
output
wire
assign

Structure of a **module** in Verilog

```
module demo(A, B, C, Y);
   input  A;
   input  B;
   input  C;
   output Y;

   wire a_or_b;

   assign a_or_b = A | B;
   assign Y = a_or_b & (~C);
endmodule
```

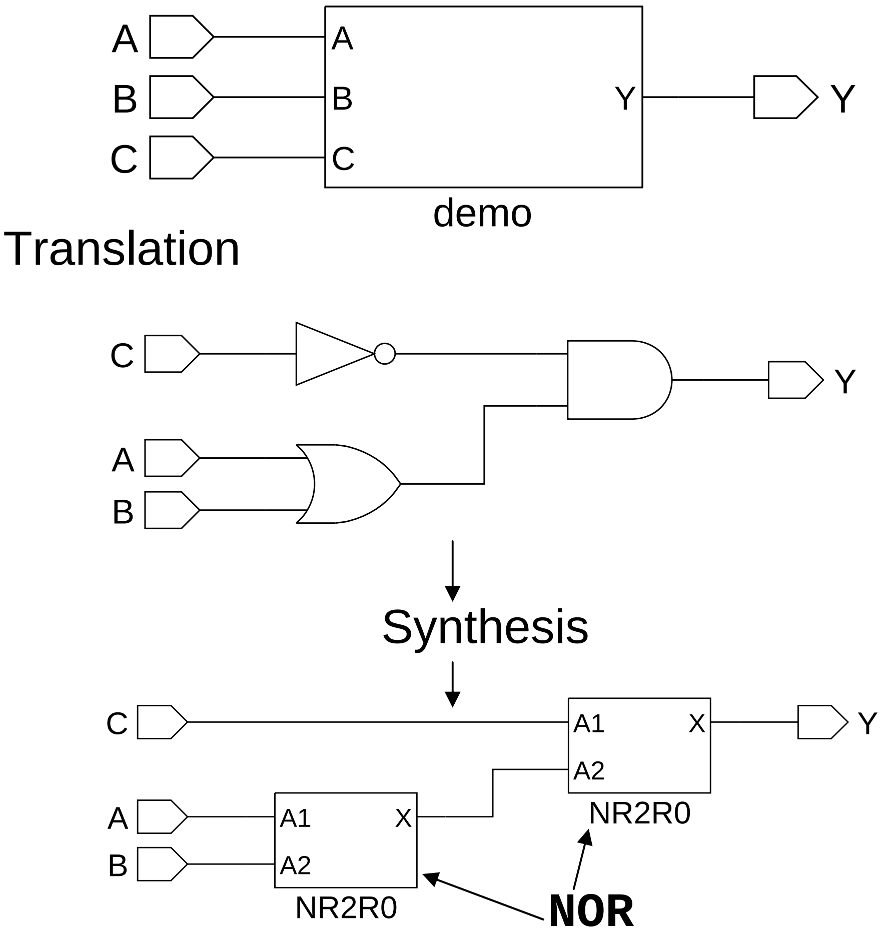
the type of the ports

internal signals

the order is not important

Verilog is case-sensitive!
All keywords are in lower case!
sig, SIG, Sig, sIg,
sIG, SiG are all different names!

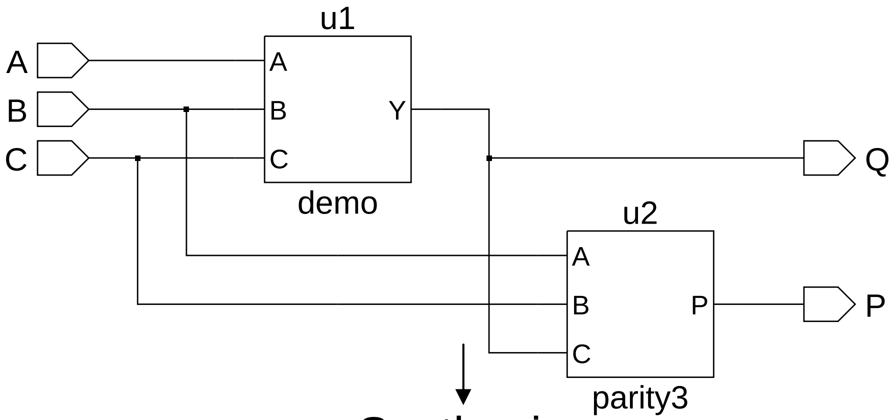
Recommendation: 1 file – 1 module
filename = name of the module



module
input
output
assign

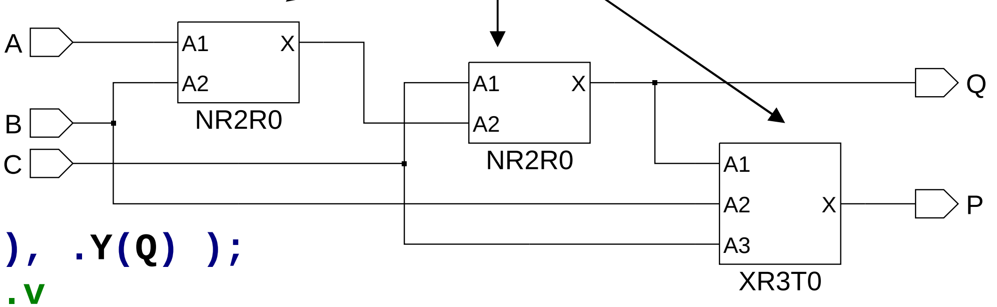
Instantiation of sub-blocks

```
module parity3(A, B, C, P);
  input   A;
  input   B;
  input   C;
  output  P;

  assign P = A ^ B ^ C;
endmodule
```



Synthesis

Library of components for ASIC



```
module top(A, B, C, P, Q);
  input   A;
  input   B;
  input   C;
  output  P;
  output  Q;
// instantiation of demo.v
demo u1 (.A(A), .B(B), .C(C), .Y(Q) );
// instantiation of parity3.v
parity3 u2 (.A(B), .B(C), .C(Q), .P(P) );
endmodule
```

module name

label

.portname(signalname)

Instantiation - mapping

```
demo u1 (.A(A),
         .B(B),
         .C(C),
         .Y(Q) );
```

There is a shorter way to connect the ports of the component to the signals, shown to the right. ***Not recommended***, as in this case the mapping depends on the correct order!

```
demo u1(A, B, C, Q);
```

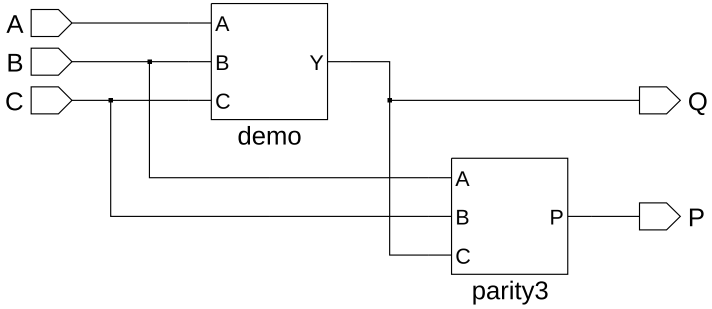
a wrong order of the signals here will change the circuit!!!

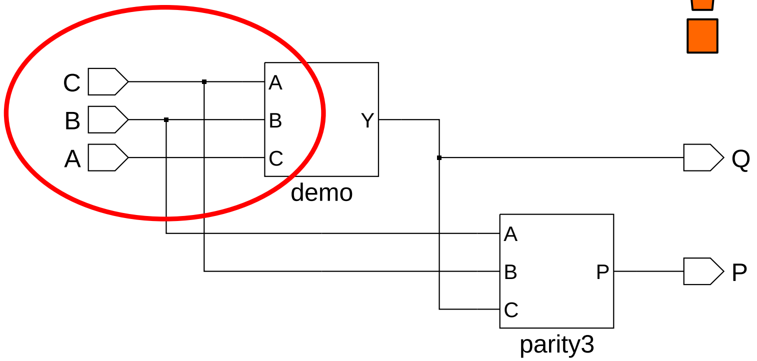
```
demo u1 (.C(C),
         .B(B),
         .A(A),
         .Y(Q) );
```

swapping of the pairs doesn't matter

```
demo u1(C, B, A, Q);
```

A and **C** are swapped



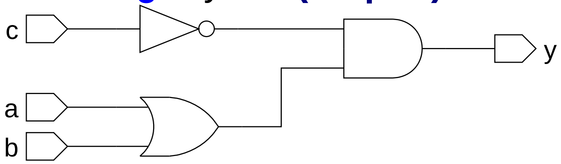


~
&
|
^
~^

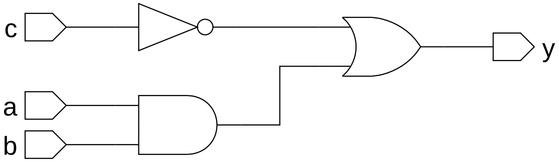
Combinational circuits – **assign**

Using concurrent assignments and simple bitwise operations:
~ not, & and, ^ xor, ~^ xnor , | or

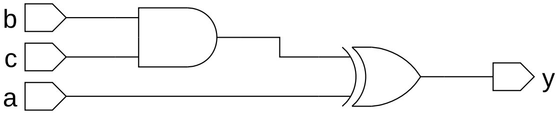
```
assign y = (a | b) & ~ c;
```



```
assign y = a & b | ~ c;
```

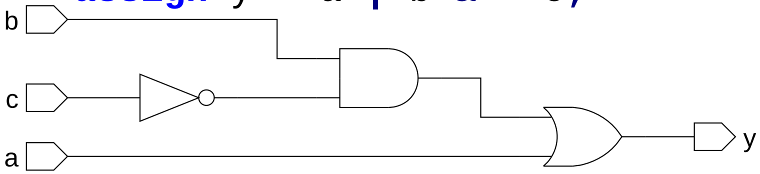


```
assign y = a ^ b & c;
```

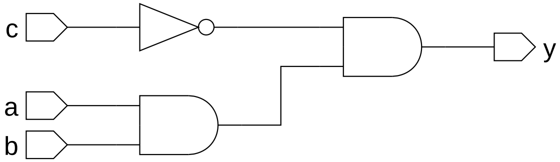


Conditional **assign** – see later

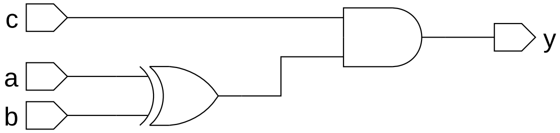
```
assign y = a | b & ~ c;
```



```
assign y = a & b & ~ c;
```

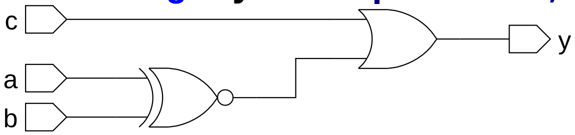


```
assign y = (a ^ b) & c;
```



```
assign y = a ~^ b | c;
assign y = c | a ~^ b;
```

} the same



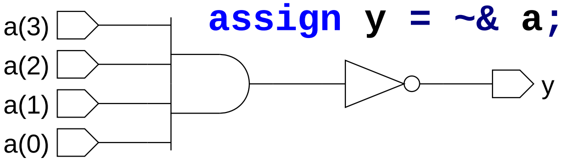
~
&
|
^
~^

Combinational circuits – reduction

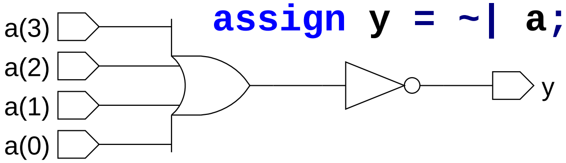
Using concurrent assignments and simple reduction operations:
& and, ~& nand, | or, ~| nor, ^ xor, ~^ xnor

```
module reduction(a, b, y);
parameter N = 4;
input [N-1:0] a;
input [N-1:0] b;
output y;
```

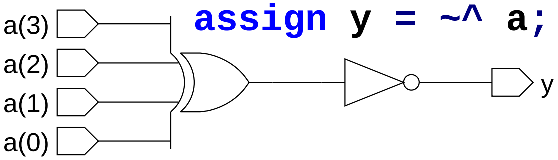
```
assign y = ~& a;
```



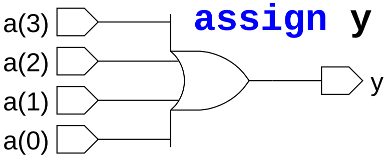
```
assign y = ~| a;
```



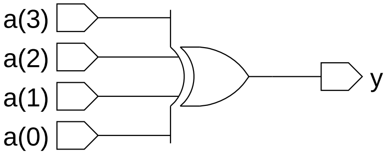
```
assign y = ~^ a;
```



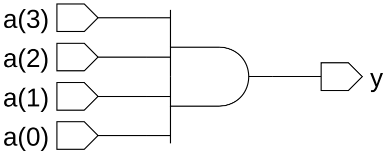
```
assign y = | a;
```

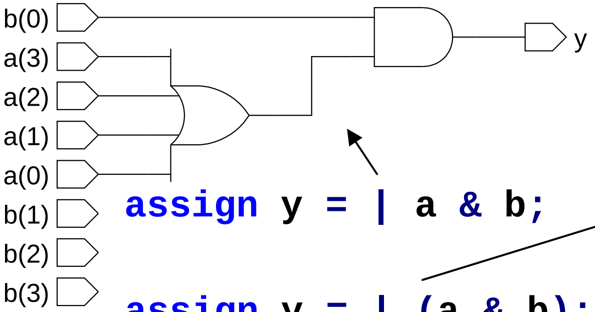


```
assign y = ^ a;
```



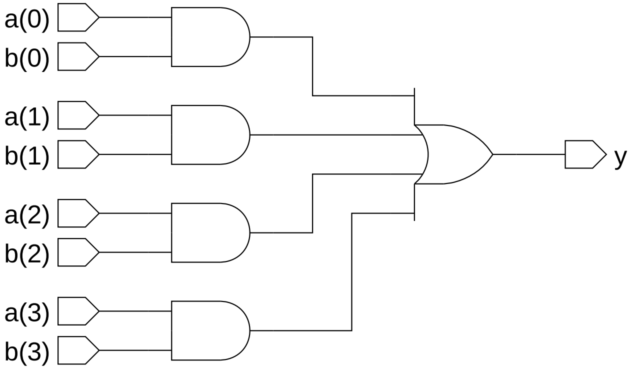
```
assign y = & a;
```





```
assign y = | a & b;
```

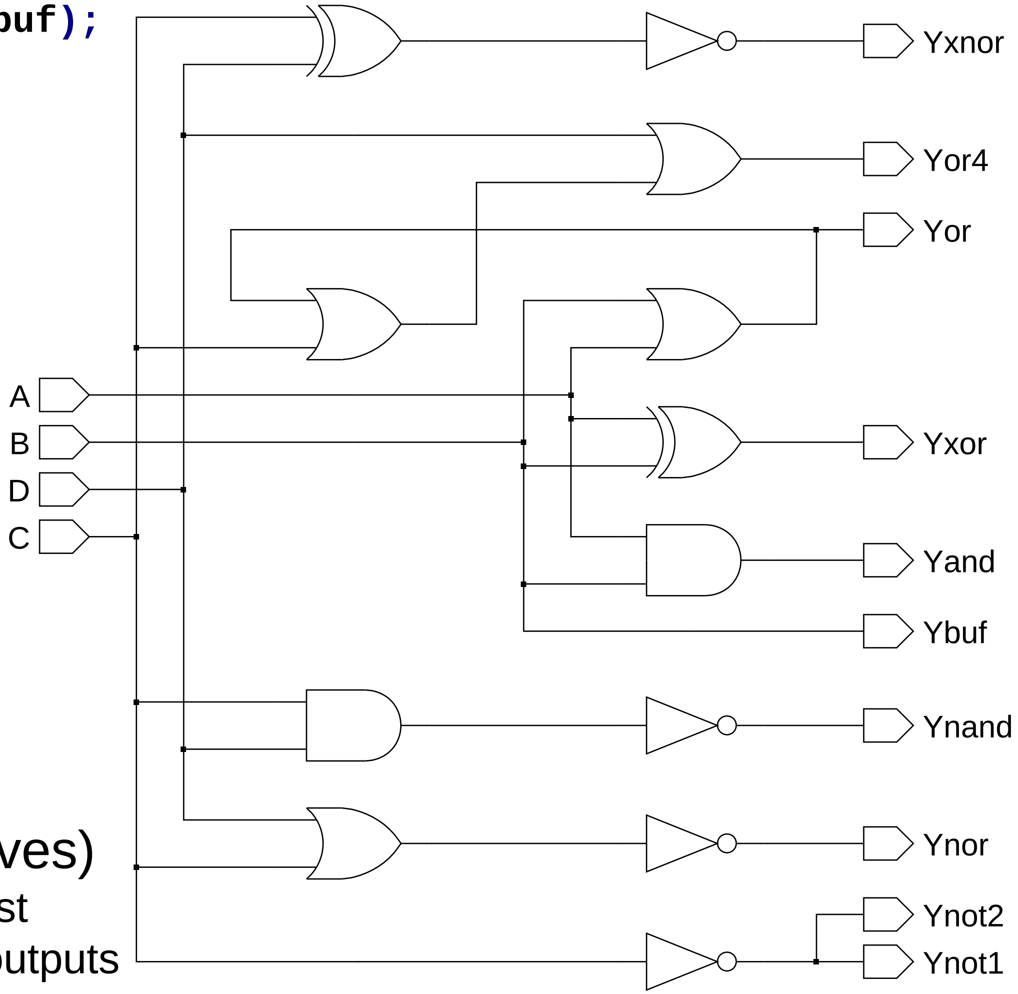
```
assign y = | (a & b);
```



and
nand
or
nor
xor
xnor
not
buf

Combinational circuits – gates

```
module gates(A, B, C, D, Yor, Ynor, Yxor, Yxnor, Yand, Ynand,
       Ynot1, Ynot2, Yor4, Ybuf);
// port types
   input  A, B, C, D;
   output Yor,   Yor4, Ynor;
   output Yxor,  Yxnor;
   output Yand,  Ynand;
   output Ynot1, Ynot2;
   output Ybuf;
// simple gates
   or (Yor,  A, B);
   or (Yor4, A, B, C, D);
   and (Yand, A, B);
   nand (Ynand, C, D);
   nor (Ynor, D, C);
   xor (Yxor, A, B);
   not (Ynot1, Ynot2, C);
   xnor (Yxnor, D, C);
   buf (Ybuf, B);
```



Built-in gates (primitives)

- the output is always the left-most
- **buf** and **not** can have several outputs

Combinational circuits – **always**

input
output
always
reg
if
else

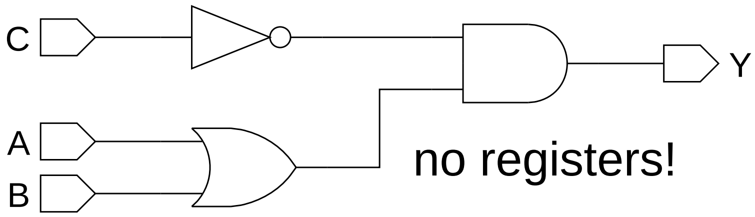
also known as **always procedure**

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

  reg y;

  always @(a or b or c)
  begin
      if (c == 1) y <= 0;
      else if (a==1 || b==1) y <= 1;
      else y <= 0;
  end
endmodule
```

or or ,

Note that **reg** is different from **wire**! The signals of type **reg** hold their values between the simulation deltas. This is necessary, as the **always** construct is executed only after an event on any of the signals in its **sensitivity list**. Verilog was developed for simulation and as interpreter language.

Note that the assignments to **y** use **<=** instead of just **=**. In this particular case we could use **=**, but mixing both types for the same signal is not allowed!



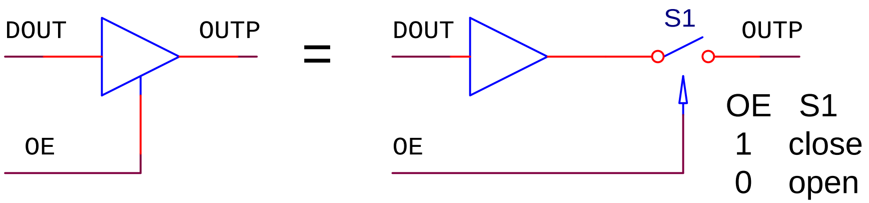
no registers!

Don't confuse **reg** with register!

module
inout

Tri-state and bidirectional ports

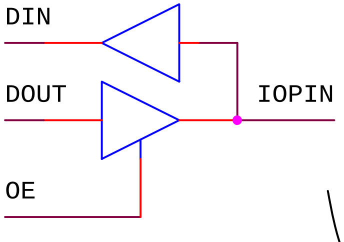
```
module tristate(OUTP, DOUT, OE);
  output OUTP;
  input  DOUT;
  input  OE  ;
  assign OUTP = (OE==1) ? DOUT : (OE==0)
                        ? 1'bz : 1'bx;
endmodule
```



OE	S1
1	close
0	open

compare!

```
module bidir(IOPIN, DIN, DOUT, OE);
  inout IOPIN;
  output DIN ;
  input  DOUT;
  input  OE  ;


  assign DIN = IOPIN;
  assign IOPIN = OE ? DOUT : 1'bz;
endmodule
```



This is more compact, but too optimistic for simulation, if **OE** is unknown, the result will be **Z** and a potential conflict on the bus will remain undiscovered.

When using tri-stated buses driven by multiple driver:

- Be sure that only one driver is active at the same time
- Insert turn-around cycles when changing the driver of the line with all drivers turned off

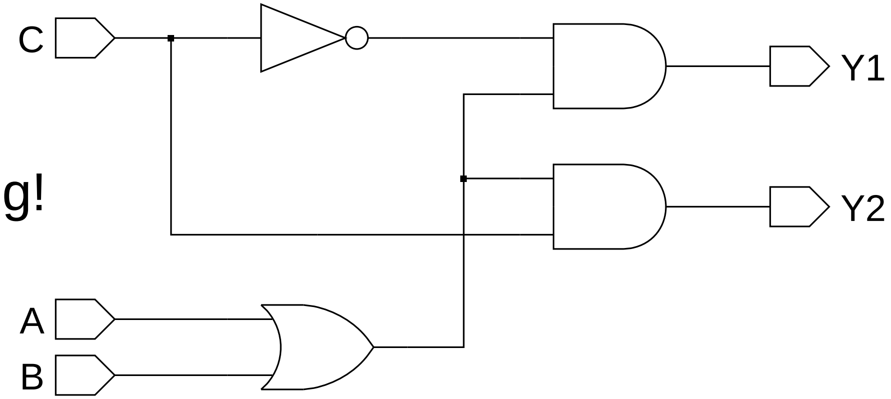
- Internal tri-state lines are typically not supported for FPGAs, some tools convert them to multiplexers.

and
not
wire
module
input
output

The **wire** in Verilog

```
module demo(A, B, C, Y1, Y2);
   input  A, B, C;
   output Y1, Y2;

   wire a_or_b;

   or myor2(a_or_b, A, B);
   not (nC, C);
   and (Y1, a_or_b, nC);
   and (Y2, C, a_or_b);
endmodule
```

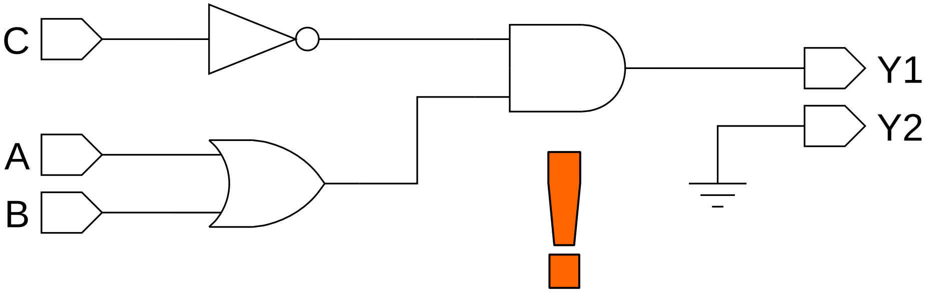
nC is missing!



Unfortunately the wire declaration is optional, as can be seen (**nC** is missing). If by error the last line with **and** uses the undeclared **wire** **a_or_d** instead of **a_or_b**,

```
and (Y2, C, a_or_d);
```

the result is undriven **Y2** and some warning.



parameter
defparam
wire

Working with vectors

```
module mux21_nbit(I0, I1, SEL, Y);
  parameter N = 8;
  input  [N-1:0] I0, I1;
  input  SEL;
  output [N-1:0] Y;

  assign Y = (SEL==1) ? I1 : I0;
endmodule

module top_mux(I0, I1, I2, I3, SEL, Y);
  parameter Width = 2;
  input [Width-1:0] I0, I1, I2, I3;
  input [1:0] SEL;
  output [Width-1:0] Y;

  wire [Width-1:0] Y01, Y23;

mux21_nbit #(Width) u1 (.I0(I0), .I1(I1), .SEL(SEL[0]), .Y(Y01));
mux21_nbit #(Width) u2 (.I0(I2), .I1(I3), .SEL(SEL[0]), .Y(Y23));
mux21_nbit u3 (.I0(Y01), .I1(Y23), .SEL(SEL[1]), .Y(Y));
defparam u3.N = Width;
endmodule
```

use **parameter** to set the size of the bit vectors

declare the input & output bus signals as vectors

Hierarchical 4:1 mux built with 3x **mux21_nbit** **module**s

The default value of the **parameter** **N** of **mux21_nbit** can be overwritten when instantiating

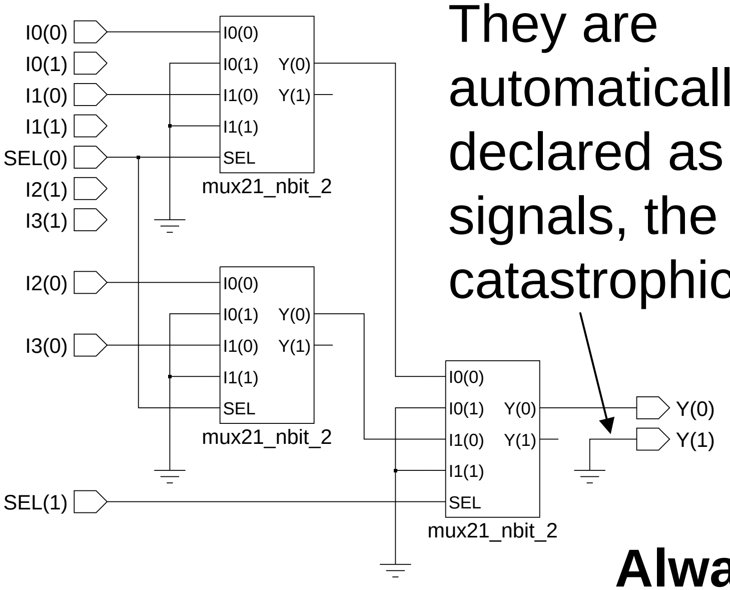
better when many parameters

module
parameter
wire

Working with vectors and wires

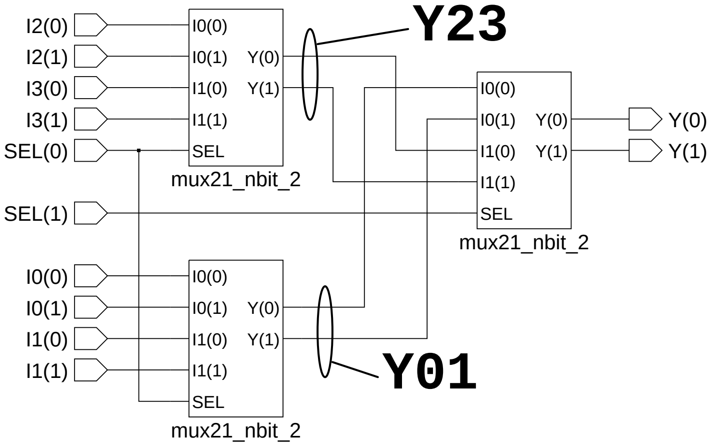
```
module top_mux(I0, I1, I2, I3, SEL, Y);
  parameter Width = 2;
  input [Width-1:0] I0, I1, I2, I3;
  input [1:0] SEL;
  output [Width-1:0] Y;
  wire [Width-1:0] Y01, Y23;
mux21_nbit #(Width) u1 (.I0(I0), .I1(I1), .SEL(SEL[0]), .Y(Y01));
mux21_nbit #(Width) u2 (.I0(I2), .I1(I3), .SEL(SEL[0]), .Y(Y23));
mux21_nbit #(Width) u3 (.I0(Y01), .I1(Y23), .SEL(SEL[1]), .Y(Y));
endmodule
```

What will happen if the internal signals **Y01** and **Y23** are not declared?



They are automatically declared as **scalar** signals, the result is catastrophic!

with proper **wire** declaration

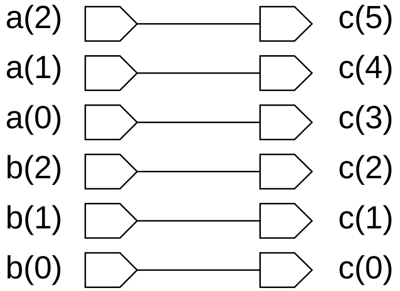


Always declare your internal wires!!!

Working with vectors - aggregates

```
input  [2:0] a, b;
output [5:0] c, d, e;
```

```
assign c = {a , b};
```

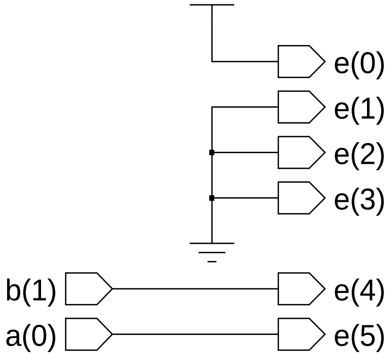


leftmost in **e**

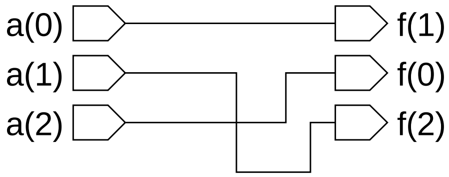
2 bit decimal

2 bit binary

```
assign e = {a[0], b[1], 2'd0, 2'b01};
```



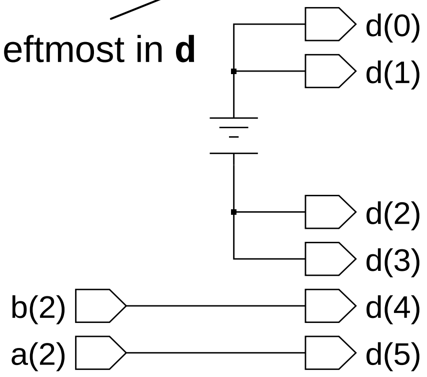
```
assign {f[0],f[2],f[1]} = a;
```



```
assign d = {a[2], b[2], 4'hC};
```

leftmost in **d**

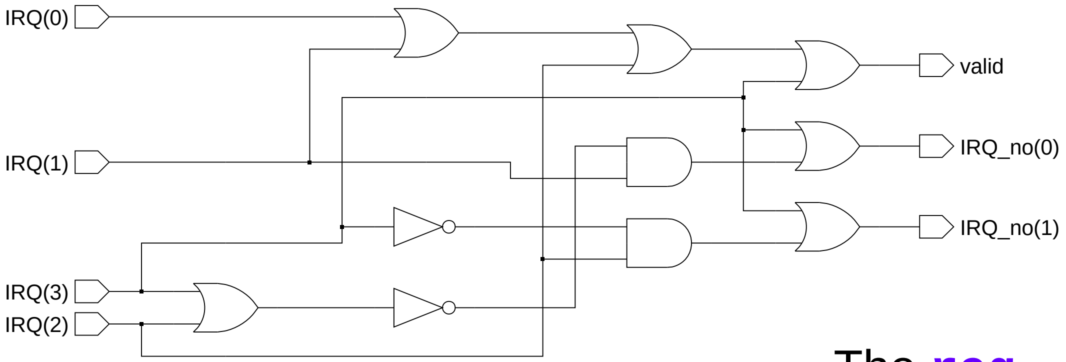
hex



reg
wire
always
if ... else
assign

Priority encoder

```
module priority(irq, valid, irq_no);
 input  [3:0] irq;
 output valid;
 output [1:0] irq_no;
 reg [1:0] irq_no;
 reg valid;
 always @(irq)
 begin
  irq_no <= 2'bxx;
  valid  <= 1'b0;
  if      (irq[3]==1) begin irq_no <= 2'd3; valid <= 1'b1; end
  else if (irq[2]==1) begin irq_no <= 2'd2; valid <= 1'b1; end
  else if (irq[1]==1) begin irq_no <= 2'd1; valid <= 1'b1; end
  else if (irq[0]==1) begin irq_no <= 2'd0; valid <= 1'b1; end
 end
endmodule
```



The **reg** signals in the **always** block should get value in all cases!

Or using only concurrent → (continuous) assignments

```
wire [1:0] irq_no;
wire valid;
assign valid = | irq;
assign irq_no = (irq[3]==1) ? 2'd3 :
                (irq[2]==1) ? 2'd2 :
                (irq[1]==1) ? 2'd1 :
                (irq[0]==1) ? 2'd0 : 2'bxx;
```

Note that **valid** and **irq_no** are **reg** in the **always** case and **wire** in the **assign** case!

reg
always
casex
casez

Priority encoder – **casex, casez**

```
module priority(irq, valid, irq_no);
  input  [3:0] irq;
  output valid;
  output [1:0] irq_no;
  reg [1:0] irq_no;
  reg valid;
always @(irq)
begin
    irq_no <= 2'bxx;
    valid  <= 1'b1;
    casex (irq)
    4'b1xxx : irq_no <= 2'd3;
    4'b01xx : irq_no <= 2'd2;
    4'b001x : irq_no <= 2'd1;
    4'b0001 : irq_no <= 2'd0;
    4'b0000 : valid  <= 1'b0;
    default : valid  <= 1'bx;
    endcase
end
```

The **reg** signals in the **always** block should get value in all cases, otherwise latches will be inferred (see later)

this value (don't care) remains if none of the inputs is active, the compiler is free to minimize the logic resources

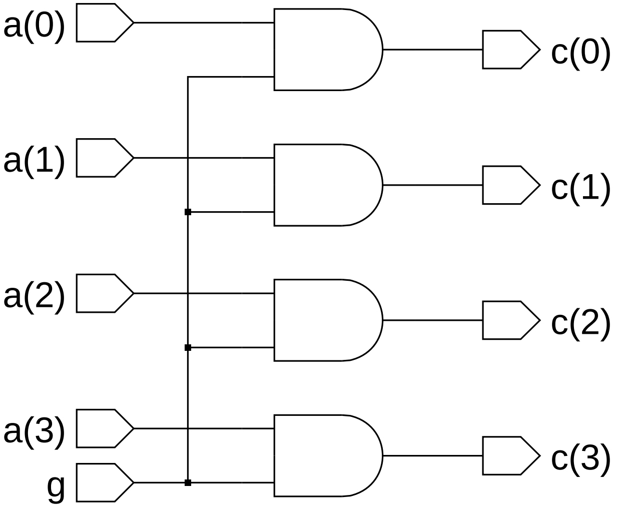
casex interprets **x** and **z** as don't care, any possible value
casez interprets only **z** as don't care

The values assigned to the **reg** signals inside the **always** are postponed until the end of the block. Only the last assigned value remains.

Gating of a vector

```
module gate_array(a, g, y);
parameter N = 4;
input g;
input  [N-1:0] a;
output [N-1:0] y;

reg [N-1:0] tmp, y;
integer i;

always @(a or g)
begin
    for (i = 0; i < N; i = i + 1)
      begin
          tmp[i] = a[i] & g;
      end
    y <= tmp;
end
endmodule
```

integer is used as index in the loop, not as a signal!



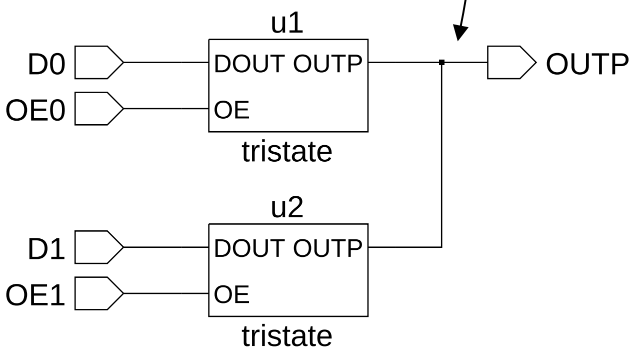
for loop, with fixed boundaries, otherwise not synthesizable

A much simpler solution

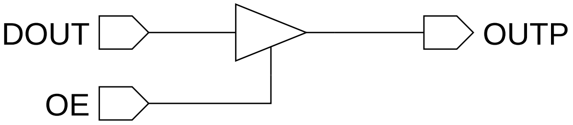
```
assign y = (g == 1'b1) ? a : 0;
```

Wire of type `tri`

```
module tri_top(OUTP, D0, D1, OE0, OE1);
output OUTP;
input D0, D1, OE0, OE1;

tri OUTP;

tristate u1(OUTP, D0, OE0);
tristate u2(OUTP, D1, OE1);
endmodule
```



```
module tristate(OUTP, DOUT, OE);
  output OUTP;
  input  DOUT;
  input  OE  ;

  assign OUTP = (OE==1) ? DOUT : (OE==0)
                        ? 1'bz : 1'bx;

endmodule
```



Multiple assignments short the `wire` together. The synthesis tools recognise the conflict and assert an error message. There are exceptions from this rule. We can connect tri-state drivers together by using the type `tri`. Most of the tools accept a normal `wire` here.

Note that the absence of warning or error in the synthesis doesn't guarantee that the logic controlling the output enable signals is correct! This is a responsibility of the designer!

wor
wand

Wired OR, AND using **wor**, **wand**

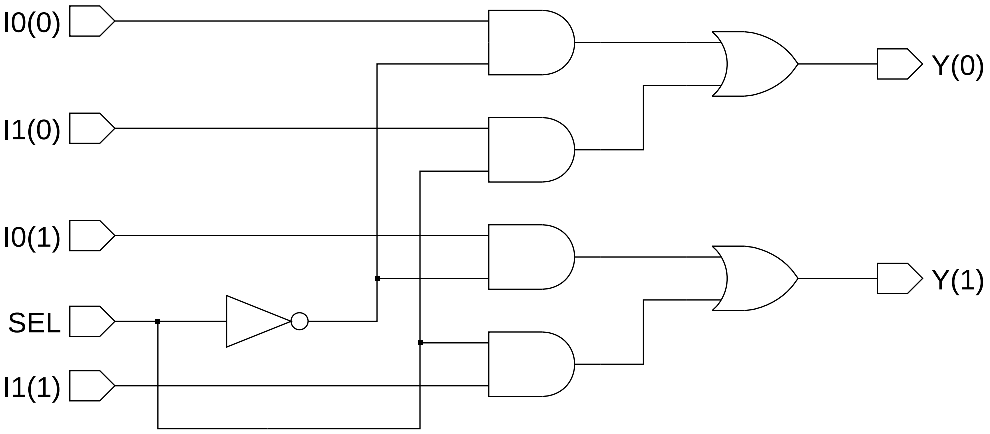
```
module mux21_nbit(I0, I1, SEL, Y);
  parameter N = 2;
  input  [N-1:0] I0, I1;
  input  SEL;
  output [N-1:0] Y;
  wor [N-1:0] Y;
  wire [N-1:0] I0g, I1g;
  assign I0g = (SEL==0) ? I0 : 0;
  assign I1g = (SEL==1) ? I1 : 0;
  assign Y = I0g;
  assign Y = I1g;
endmodule
```

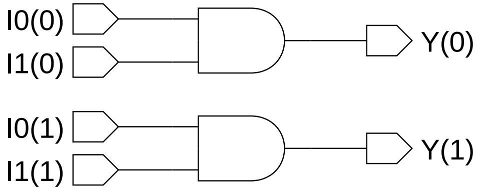
Another exception from this rule is for the two types of wire: **wor** / **wand**.

In this case multiple assignments create OR / AND gates.

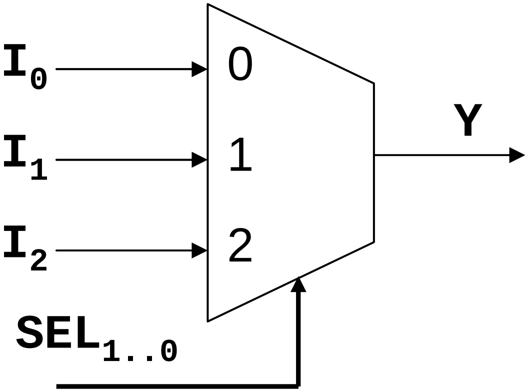
Very useful to implement muxes

```
module wired_and(I0, I1, Y);
  parameter N = 2;
  input  [N-1:0] I0, I1;
  output [N-1:0] Y;
  wand [N-1:0] Y;
  assign Y = I0;
  assign Y = I1;
endmodule
```





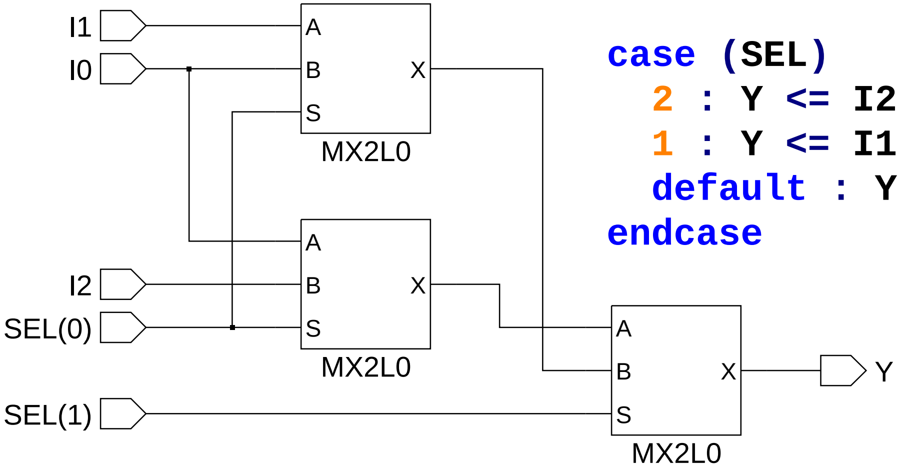
Example for don‘t care with a multiplexer 3:1



SEL	Y
0	I0
1	I1
2	I2
3	?

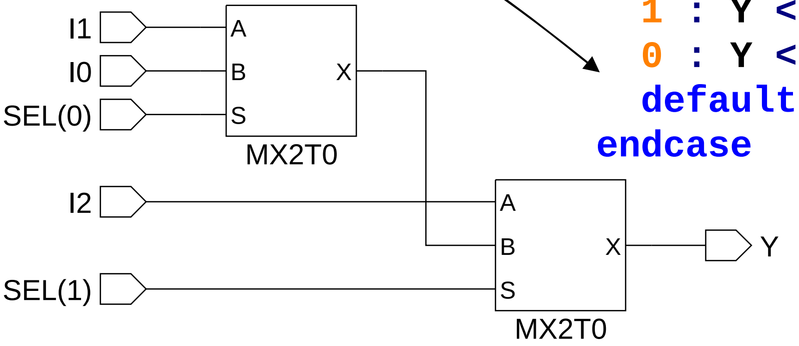
What is the benefit of **x ?**

- automatic choice of the best value
- undefined output when the input **SEL** is unexpected in the functional simulation



```
case (SEL)
  2 : Y <= I2;
  1 : Y <= I1;
  default : Y <= I0;
endcase
```

```
case (SEL)
  2 : Y <= I2;
  1 : Y <= I1;
  0 : Y <= I0;
  default : Y <= 1'bx;
endcase
```



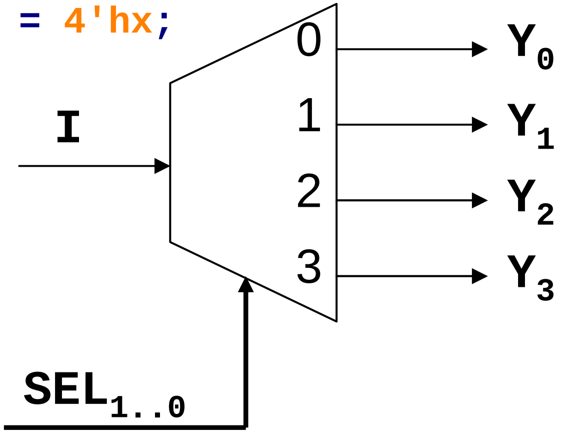
parameter
reg
always
case

Demultiplexer (decoder)

```
module demux2to4(I, SEL, Y);
  input  [1:0] SEL;
  input  I;
  output [3:0] Y;
  reg [3:0] Y;
always @(I or SEL)
begin
  Y = 4'd0;
  case (SEL)
  2'b00   : Y[0] = I;
  2'b01   : Y[1] = I;
  2'b10   : Y[2] = I;
  2'b11   : Y[3] = I;
  default : Y = 4'hx;
  endcase
end
endmodule
```

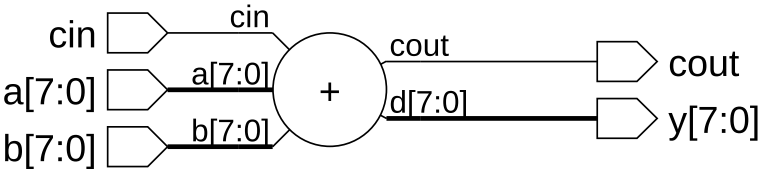
Fully scalable →

```
module demux(I, SEL, Y);

  parameter Nbit = 2;
  parameter Nout = 1 << Nbit;

  input  [Nbit-1 : 0] SEL;
  input  I;
  output [Nout-1 : 0] Y;

  reg [Nout-1 : 0] Y;

always @(I or SEL)
begin
  Y <= 0;
  Y[SEL] <= I;
end
endmodule
```



Adder

```
module adder(a, b, cin, y, cout);
  parameter N = 8;
  input cin;
  input  [N-1:0] a, b;
  output [N-1:0] y;
  output cout;

  wire [N:0] sum;

  assign sum = cin + a + b;
  assign y = sum[N-1:0];
  assign cout = sum[N];

  assign {cout, y} = cin + a + b;

endmodule
```



In Verilog the conversion from bitvectors to integers is done automatically, you can freely use arithmetic expressions. By default the bitvectors represent unsigned integers.

Automatic extension of the smaller vectors to the size of the target by padding 0 or with sign extension if the signals are signed:

```
input signed [N-1:0] a, b;
output signed [N-1:0] y;
```

for
if…else
always

Barrel shifter right

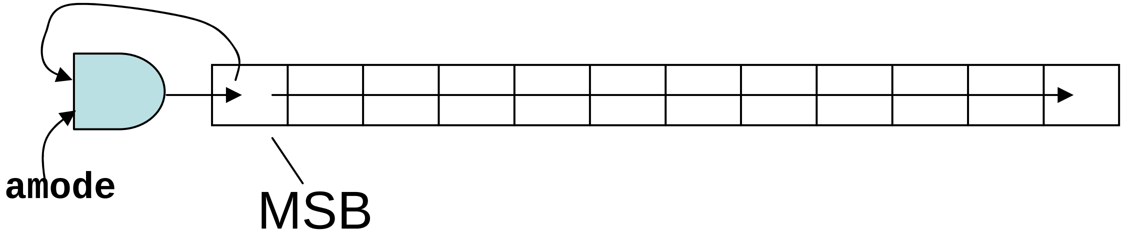
```
module barrel_v(a, shd, amode, y);

parameter N = 16;
parameter M = 3;
```

The size of **a** and **y**

The size of **shd**, 2^M-1 is the max shift distance

```
input amode;
input  [N-1:0] a;
input  [M-1:0] shd;
output [N-1:0] y;

reg [N-1:0] y;
wire msb;
```

In arithmetic mode the MSB remains, otherwise fill with 0 from the left



```
integer i;
```

Integer is used as index in the loop, not as a signal!

```
assign msb = amode & a[N-1];
always @(a or shd or msb)
begin
    for (i = 0; i < N; i = i + 1)
      begin
          if (i > (N - 1 - shd)) y[i] = msb;
          else y[i] = a[i + shd];
      end
end
endmodule
```

For synthesis only **for** loops with constant boundaries are allowed, for simulation one can use **while**

Sequential circuits – DFF and DFF with enable

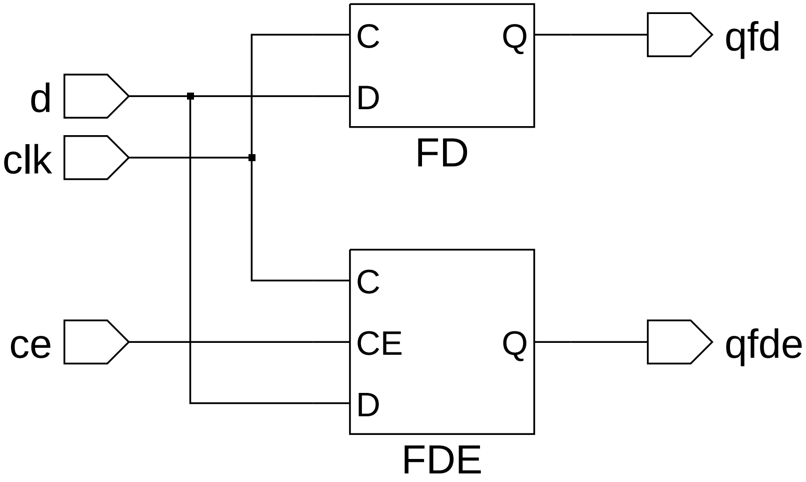
```
module dffe(d, clk, ce, qfd, qfde);

  input  d, clk, ce;
  output qfd, qfde;

  reg qfd, qfde;

  always @(posedge clk)
  begin
    qfd <= d;
  end

  always @(posedge clk)
  begin
    if (ce == 1) qfde <= d;
  end
endmodule
```

FD

FDE

the DFF outputs must be of type reg!

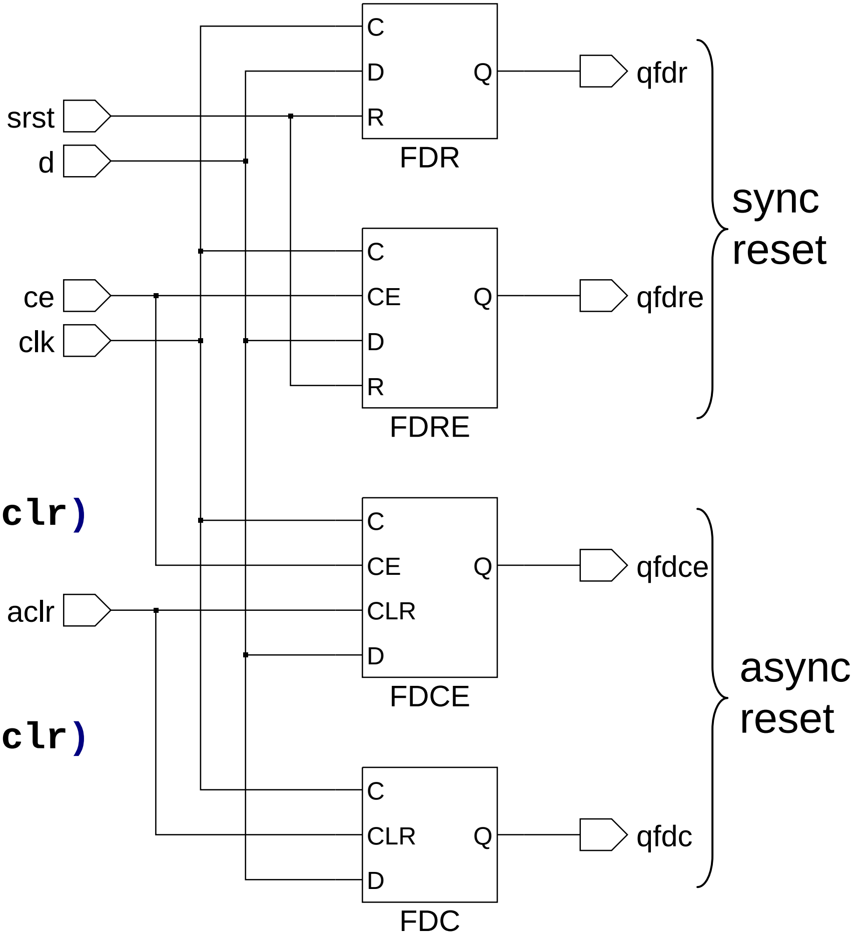
detect rising edge of the clock



The XILINX notations of the DFFs: F for flip-flop, D for D-type, E for enable…

DFFs with reset and enable

```
      reg qfdc, qfdr, qfdce, qfdre;
       always @(posedge clk)
       begin
FDR      if (srst == 1) qfdr <= 0;
         else           qfdr <= d;
       end
       always @(posedge clk)
       begin
FDRE     if     (srst == 1) qfdre <= 0;
         else if (ce == 1) qfdre <= d;
       end
       always @(posedge clk or posedge aclr)
       begin
FDC      if (aclr == 1) qfdc <= 0;
         else           qfdc <= d;
       end
       always @(posedge clk or posedge aclr)
       begin
FDCE     if     (aclr == 1) qfdce <= 0;
         else if (ce == 1) qfdce <= d;
       end
```



…E for enable, R for synchronous reset, C for asynchronous clear…

DFFs with async. set and reset

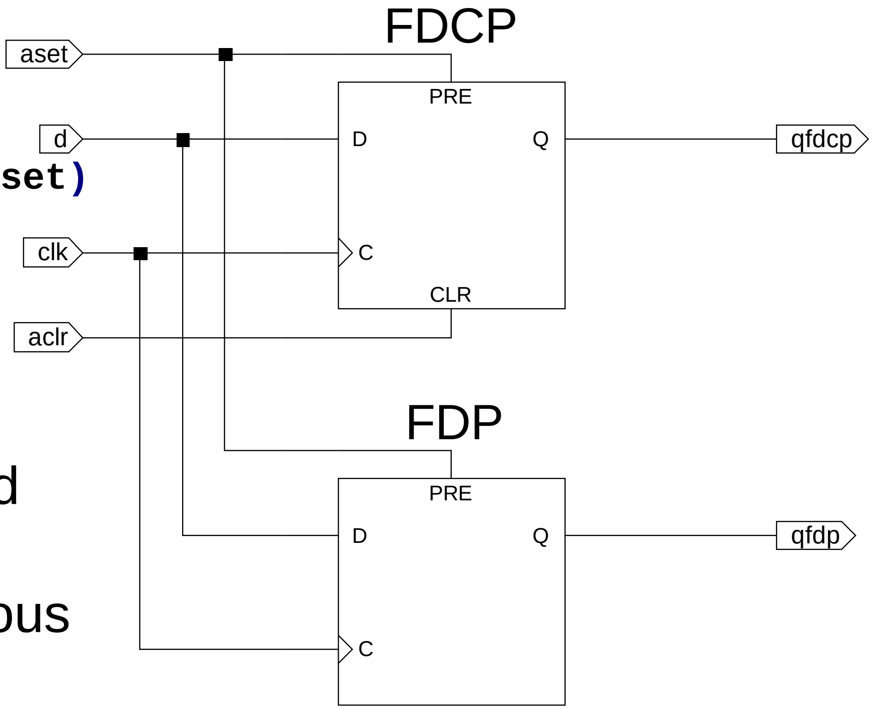
FDCP

```
reg qfdcp, qfdp;
always @(posedge clk or posedge aset or posedge aclr)
 begin
   if (aclr == 1) qfdcp <= 0;
   else if (aset == 1) qfdcp <= 1;
   else qfdcp <= d;
 end
```

the priority is important, first the clear!

FDP

```
 always @(posedge clk or posedge aset)
 begin
   if (aset == 1) qfdp <= 1;
   else           qfdp <= d;
 end
```



…P for asynchronous preset…

It is not recommended to:

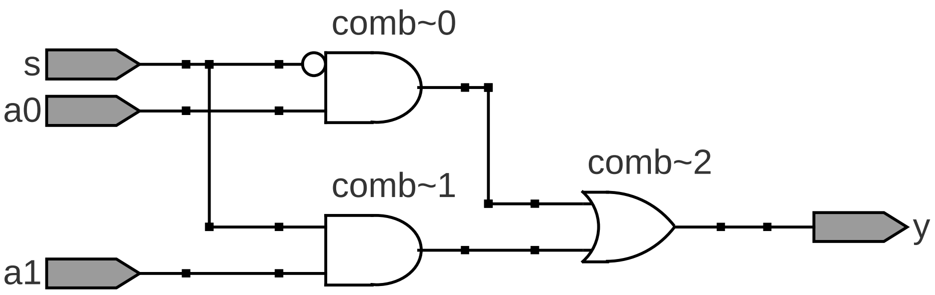
- use both asynchronous clear and preset
- mix synchronous and asynchronous clear/presets

Try to use only synchronous clear/presets!

User Defined Primitives (UDP) – combinational logic

```
/*
   Interpretation             Explanation
?  0, 1, X                    ? means the variable can be 0, 1 or x
b  0, 1                       Same as ?, but x is not included
f  (10)                       Falling edge
r  (01)                       Rising edge
p  (01),(0x),(x1),(1z),(z1) Rising edge including x and z
n  (10),(1x),(x0),(0z),(z0) Falling edge including x and z
*  (??)                       All transitions
-  no change                  No Change
*/
primitive mux2to1(y, a0, a1, s);
output y;
input a0, a1, s;
table
// a0, a1, s : y
    1   ?  0 : 1;
    0   ?  0 : 0;
    ?   1  1 : 1;
    ?   0  1 : 0;
endtable
endprimitive
```

The output is always the first port, the inputs are up to 10.
All unspecified combinations of inputs lead to undefined output (x).
Usage exactly as the built-in primitives.



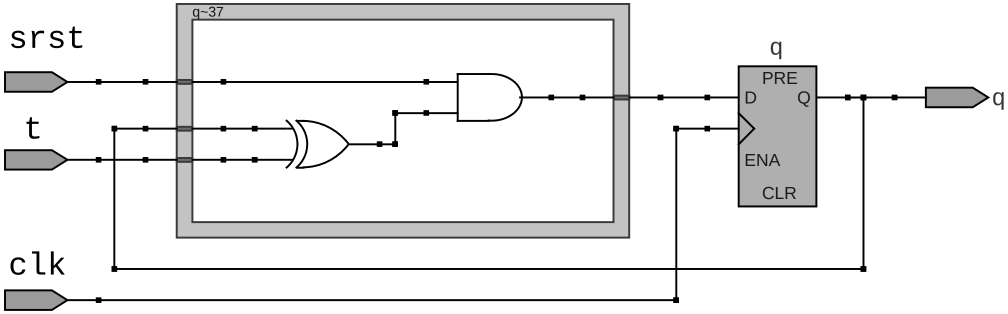
User Defined Primitives (UDP) – sequential logic

For **sequential logic** the table contains the inputs (up to 9), the present state and the next state. Note that **all valid combinations** of the inputs must be specified! Including the **edges** for non-clock signals (*)!

```
primitive dff_p(q, clk, d);
output q;
input clk, d;
reg q;
table
// clk, d : q : q+
    r   0 : ? : 0;
    r   1 : ? : 1;
    f   ? : ? : -;
    ?   * : ? : -;
endtable
endprimitive
```

```
primitive tff_p(q, clk, t, srst_n);
output q;
input clk, t, srst_n;
reg q;
table
// clk, t, srst : q : q+
    r   ?   0   : ? : 0;
    r   0   1   : ? : -;
    r   1   1   : 0 : 1;
    r   1   1   : 1 : 0;
    f   ?   ?   : ? : -;
    ?   *   ?   : ? : -;
    ?   ?   *   : ? : -;
endtable
endprimitive
```

TFF



Latches

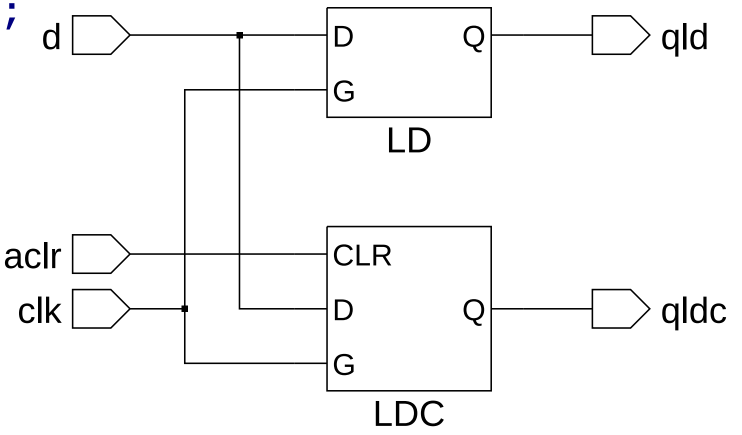
```
module latches(d, clk, aclr, qldc, qld);
  input  d, clk, aclr;
  output qldc, qld;

  reg qldc, qld;

  always @(clk or d)
  begin
    if (clk == 1) qld <= d;
  end
  always @(clk or aclr or d)
  begin
    if      (aclr == 1) qldc <= 0;
    else if (clk == 1) qldc <= d;
  end
endmodule
```

LD

LDC

no **posedge**!

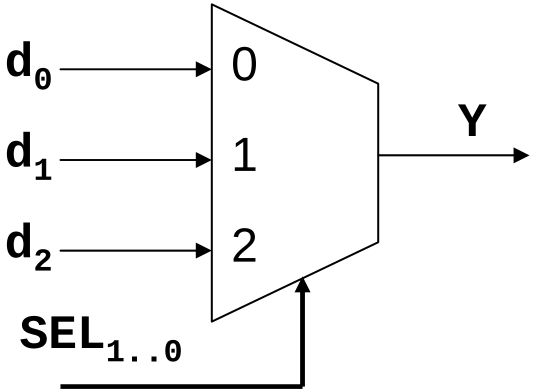
d is on the sensitivity list

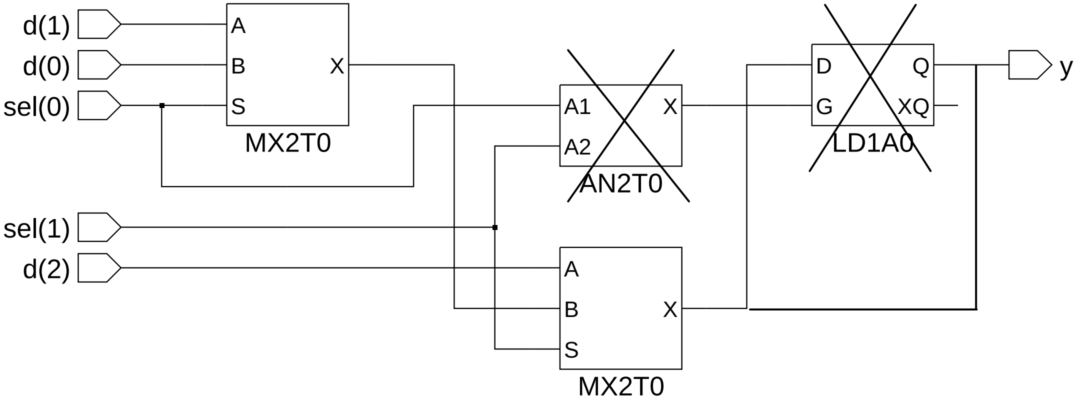


L for Latch, C for asynchronous clear

It is not recommended to use latches except in the rare cases when you really know what you are doing!
When the synthesis tools find latches, check your code!

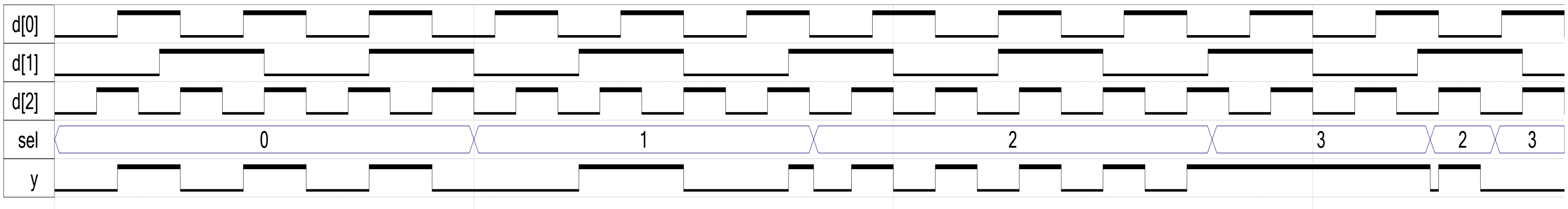
Unwanted Latch





```
always @(d or sel)
  begin
    if      (sel==0) y <= d[0];
    else if (sel==1) y <= d[1];
    else if (sel==2) y <= d[2];
    else             y <= 1'bx;
  end
```

If the output is not specified for all possible inputs, Latches are inferred!



A simple counter with synchronous reset & load

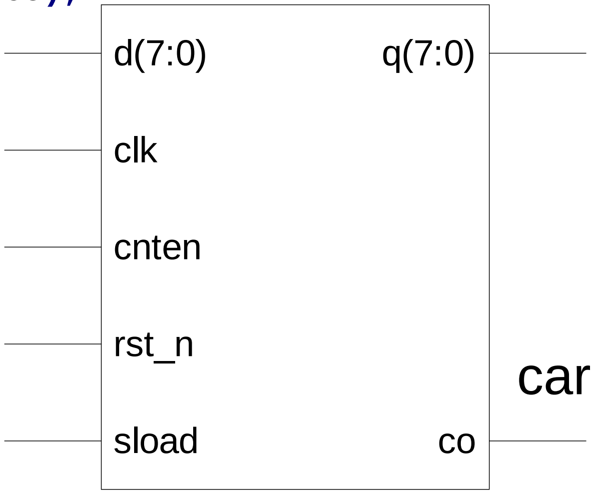
```
`timescale 1 ns / 1 ns
module counter(clk, cnten, rst_n, d, sload, q, co);
parameter del = 10;
parameter N = 4;
input  clk, cnten, rst_n, sload;
input  [N-1:0] d;
output co;
output [N-1:0] q;
reg [N-1:0] q_i;
always @(posedge clk)
begin
    if (rst_n == 0) q_i <= 0;
    else if (sload == 1) q_i <= d;
    else if (cnten == 1) q_i <= q_i + 1;
end
assign # del co = (& q_i) & (~sload) & cnten;
assign # del q = q_i;
```

Time_unit / *Time_precision* for simulation

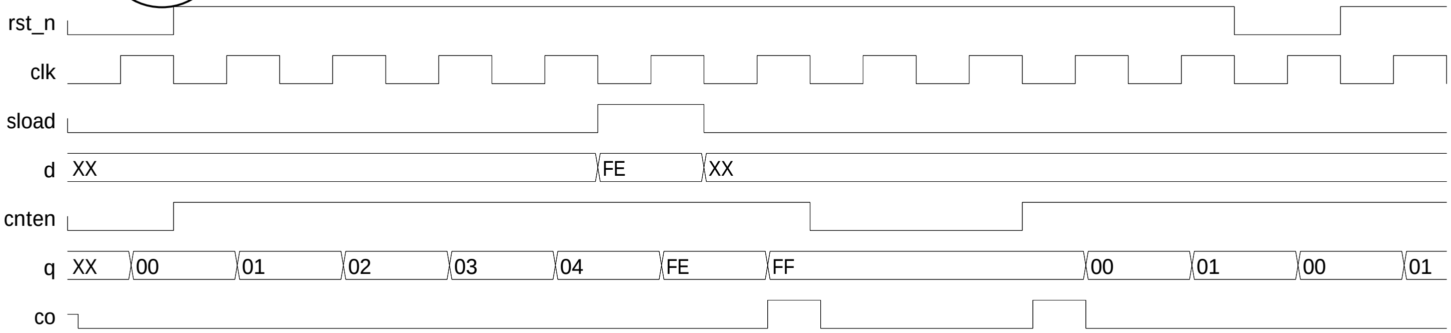
delay in **ns** (for simulation)

higher priority for reset

assign with delay

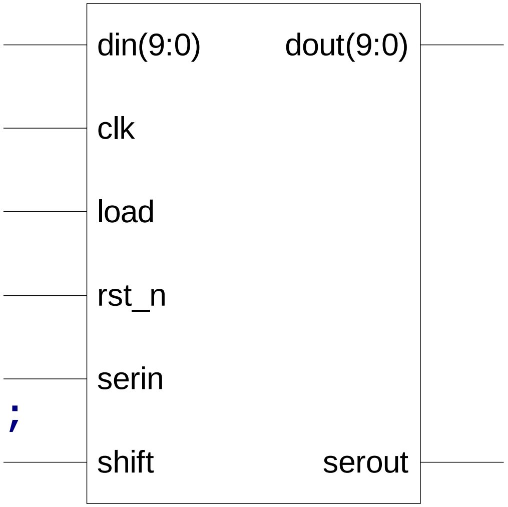


carry out

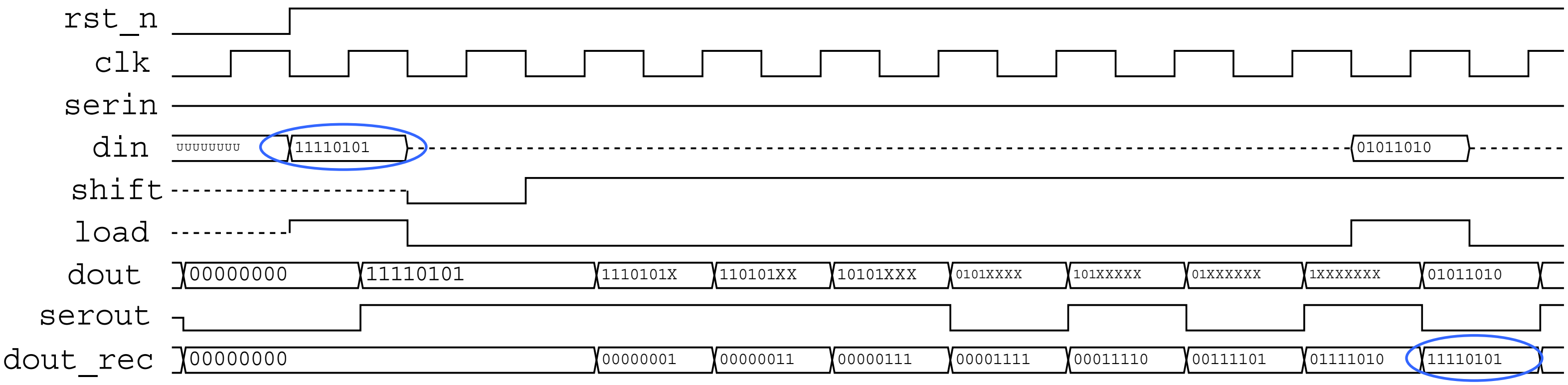


Shift register with parallel load

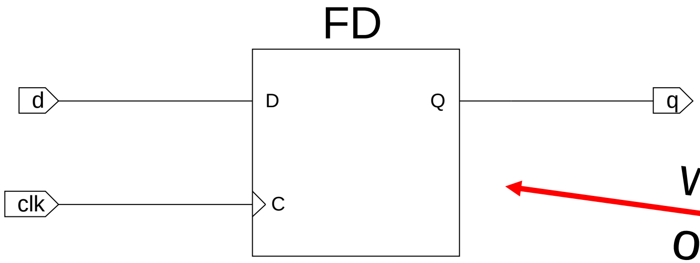
```
parameter N = 10;
…
reg [N-1:0] dout;
always @(posedge clk or negedge rst_n)
begin
  if (rst_n == 0) dout <= 0;
  else
  if (load == 1)  dout <= din;
  else
  if (shift == 1) dout <= {dout[N-2:0], serin};
end
assign serout = dout[N-1];
endmodule
```



two modules, one sends serially (**`serout`**) to the other



Shift register with blocking assignments



= in **`always`** means immediate update!

```
reg a, b, c;
always @(posedge clk)
begin
    a = d;     c = b;
    b = a;     b = a;
    c = b;     a = d;
end
assign q = c;
```

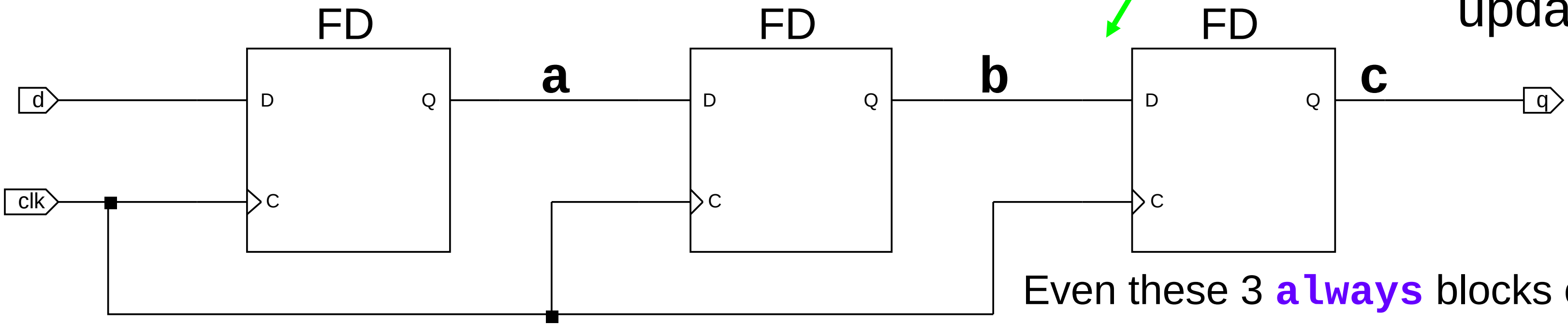
wrong order!

correct!

any order is correct

```
a <= d;
b <= a;
c <= b;
```

scheduled update



Even these 3 **`always`** blocks can be executed in a wrong order by the simulator!

```
always @(posedge clk) a = d;
always @(posedge clk) b = a;
always @(posedge clk) c = b;
```

If using non-blocking assignments (**<=**) instead of blocking assignments (**=**) in the **`always`**, the order of the three lines is **not** important! **Avoid using blocking assignments for real signals!**

`timescale
repeat
$display
initial
negedge

Simple simulation testbench

…of the two variants of the shift register with blocking assignments

```
`timescale 1 ns / 1 ns
module shift_reg_var_tb();
parameter Td2 = 5;
reg clk, d;
wire q1, q2;
initial
begin
  clk = 1'b0;
  @(negedge clk);
  repeat (2)
  begin
    d <= 1;
    repeat (6)
        @(negedge clk);
    d <= 0;
    repeat (6)
        @(negedge clk);
  end
end
```

2x

x6

```
always @ (negedge clk)
begin
  if (q1 !== q2)
   $display("%t : in %m q1 and q2 differ! q1=%b, q2=%b",
                        $time, q1, q2);
end
always # Td2 clk =  ~clk;
shift_reg_var_bad dut1(.clk(clk), .d(d), .q(q1));
shift_reg_var_ok  dut2(.clk(clk), .d(d), .q(q2));
endmodule
```

time

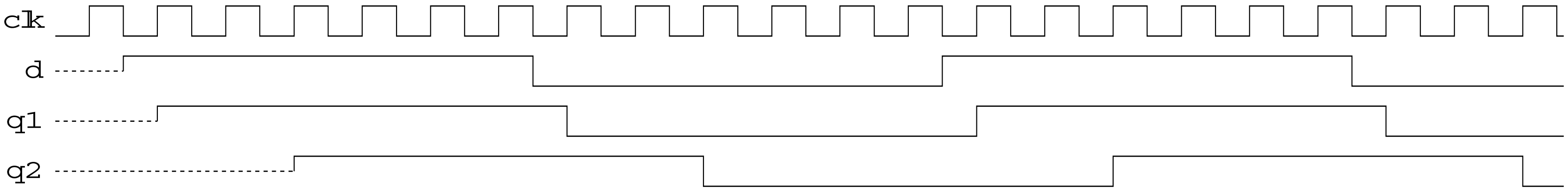
module

binary

periodical clock

Text output by the simulator

```
#   20 : in shift_reg_var_tb q1 and q2 differ! q1=1, q2=x
#   30 : in shift_reg_var_tb q1 and q2 differ! q1=1, q2=x
#   80 : in shift_reg_var_tb q1 and q2 differ! q1=0, q2=1
#   90 : in shift_reg_var_tb q1 and q2 differ! q1=0, q2=1
#  140 : in shift_reg_var_tb q1 and q2 differ! q1=1, q2=0
#  150 : in shift_reg_var_tb q1 and q2 differ! q1=1, q2=0
#  200 : in shift_reg_var_tb q1 and q2 differ! q1=0, q2=1
#  210 : in shift_reg_var_tb q1 and q2 differ! q1=0, q2=1
```



Conditional compilation, include

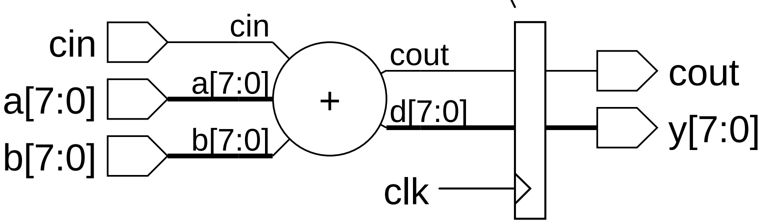
`include
`ifdef
`else
`endif
`define

```
`include "SRC_v/incl.v" → `define out_reg
`ifdef out_reg
module adder(clk, a, b, cin, y, cout);
input clk;
`else
module adder(a, b, cin, y, cout);
`endif
parameter N = 8;
input cin;
input [N-1:0] a, b;
output cout;
output [N-1:0] y;
`ifdef out_reg
reg cout;
reg [N-1:0] y;
  always @(posedge clk)
  begin
      {cout, y} = cin + a + b;
  end
`else
assign {cout, y} = cin + a + b;
`endif
endmodule
```

In many programming languages (in C with **`#define … #ifdef … #endif`**) one can compile some parts of the source code depending on some condition.

In Verilog, it is almost the same as in C, just replace # with \` . This is useful to configure the module, instead of creating many variants as different modules. Similarly with **\`include** one can include a source file.

depending on **out_reg**



parameter
localparam
if … else
for
reg
always

Registerfile – two-dimens. array

```
module regfile(clk, rst_n, we, din, waddr, raddra, raddrb, rdata, rdatb);
  parameter Na =  3;
  parameter Nd = 16;
  input  clk, rst_n, we;
  input  [Na-1:0] waddr, raddra, raddrb;
  input  [Nd-1:0] din;
  output [Nd-1:0] rdata, rdatb;
  localparam Nr = 1 << Na;
  reg [Nd-1:0] rf_data [0:Nr-1];
  reg [0:Nr-1] we_dec;
  integer i;

  always @(we or waddr) // the decoder
  begin
      we_dec <= 0;
      we_dec[waddr] <= we;
  end
  always @(posedge clk) // the registers
    for (i = 0; i < Nr; i = i + 1)
    begin
      if (rst_n == 0) rf_data[i] <= 0;
      else
      if (we_dec[i] == 1) rf_data[i] <= din;
    end

  assign rdata = rf_data[raddra]; // two read muxes
  assign rdatb = rf_data[raddrb];

endmodule
```

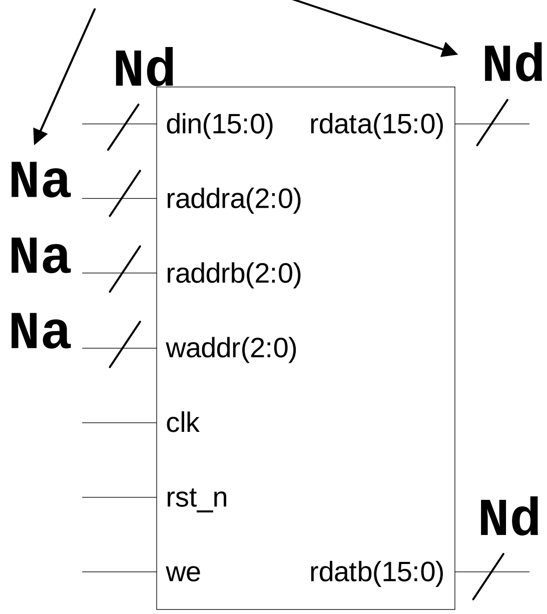
local parameter, can not be seen/modified from outside

two-dimensional array

used in the **for** loop

As a part of a RISC CPU core

Number and size of registers set through parameters



$dumpfile
$dumpvars
$time
$monitor
`timescale

Simulation of the regfile(1)

```
`timescale 1 ns / 1 ns
```

← *Time_unit* / *Time_precision* for simulation

```
module regfile_tb();

parameter Td2 = 5; // the half of the period
parameter Nd = 16; // the data size
parameter Na =  3; // the address size

localparam Nreg = (1 << Na); // the number of registers

reg clk, rst_n, we;
wire [Nd-1:0] rda, rdb; // read data
reg [Nd-1:0] d;         // write data
reg [Na-1:0] wa, ra, rb; // addresses
reg [Nd-1:0] regm [0:Nreg-1]; // shadow memory for the regs
integer i; // loop index
```

declare all variables

instantiate the regfile

```
regfile dut(.clk(clk), .rst_n(rst_n), .we(we), .din(d), .waddr(wa),
             .raddra(ra), .raddrb(rb), .rdata(rda), .rdatb(rdb));

defparam dut.Nd = Nd; // map the parameters
defparam dut.Na = Na;
initial begin
    $dumpfile("regfile.vcd");
    $dumpvars(1, regfile_tb.dut);
  end
initial
   $monitor("Time: %t, write enable changed to: %b", $time, we);
```

dump file, in .vcd (**V**alue **C**hange **D**ump) format

dump only the signals in the **dut** instance

print all changes of the signal **we**

Simulation of the regfile(2)

```
always # Td2 clk =  ~clk; // generate the clock

initial
begin
  clk =   0;     // initial value of the clock
  rst_n = 0;     // power up reset
  repeat (2) @(negedge clk);
  rst_n = 1;     // deactivate the reset

  for (i = 0; i < Nreg; i = i + 1) // write some pattern
    write_reg(i, i | (i+1) << 8);
  for (i = 0; i < Nreg; i = i + 1) // read back
    read_reg(0, i, regm[i]);       // using port A
  for (i = 0; i < Nreg; i = i + 1)
    read_reg(1, i, regm[i]);       // and port B

  rst_n = 0;                       // reset
  @(negedge clk);
  rst_n = 1;
  for (i = 0; i < Nreg; i = i + 1) // read and expect 0
    read_reg(0, i, 0);
  for (i = 0; i < Nreg; i = i + 1) // from both ports
    read_reg(1, i, 0);
end
```

blocks of type initial will be executed only once

tasks

read back and compare with the expected values

tasks

check if all registers are cleared

Simulation of the regfile(3) - tasks

```
task write_reg;
  input [Na-1:0] rega;
  input [Nd-1:0] regd;
  begin
    we = 1;
    d  = regd;
    wa = rega;
    @ (negedge clk);
    regm[rega] = regd;
    we = 0;
  end
endtask
```

The **task** can have **inputs**, **outputs** and **inouts**, but all **module** variables and **wires** are visible and can be accessed!
The **task** can contain time-controlling statements.

```
task read_reg;
  input prt; // 0 or 1
  input [Na-1:0] rega;
  input [Nd-1:0] regd;
  begin
    we = 0;
    if (prt==0) ra = rega;
    else        rb = rega;
    @ (negedge clk);
    if ((prt==0) && (rda != regd))
      $display("%t : expected 0x%04x, readA 0x%04x, reg# %d", $time, regd, rda, rega);
    if ((prt==1) && (rdb != regd))
      $display("%t : expected 0x%04x, readB 0x%04x, reg# %d", $time, regd, rdb, rega);
end
endtask
endmodule
```

port, 0 for A and 1 for B

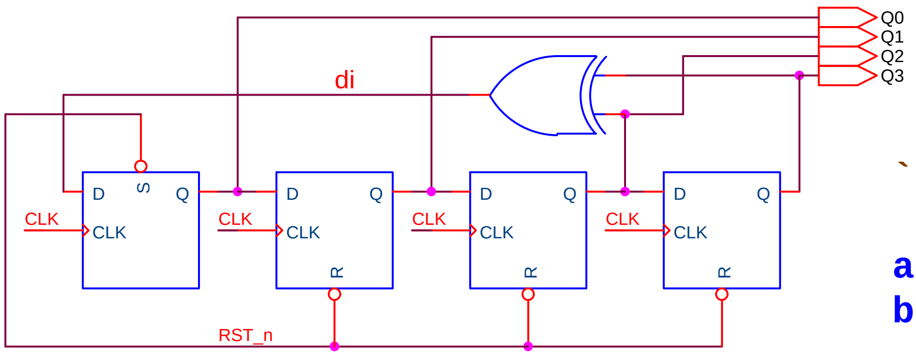
read address

expected data

Check if the read data correspond to the expected and print a message if not

Pseudorandom generators(1)

N-bit shift register with a feedback, used to generate pseudorandom numbers. Note that the number of possible states is 2^N-1, not 2^N! The example below is for N=4.



(the load network is not shown)

```
module psrg_gen(clk, load, din, q);
  parameter N = 4;
  input clk, load;
  input [N-1:0] din;
  output [N-1:0] q;
`include "../SRC_v/psrg_func.v"
  reg [N-1:0] q;
always @(posedge clk)
begin
  if (load == 1'b1) q <= din;
  else
  q <= {q[N-2:0], psrg_func(N, q)};
end
endmodule
```

next slide

calculates the new bit 0

Used: for built-in tests of memories; to generate input data in testbenches; for Monte Carlo simulations; as simple frequency divider

function
case

Pseudorandom generators(2)

```
function psrg_func;
input integer N;
input [31:0] sr; ←— for up to 32 bit input
case (N)
   32 : psrg_func = sr[31] ^ sr[21] ^ sr[ 1] ^ sr[ 0];
   31 : psrg_func = sr[30] ^ sr[27];
   30 : psrg_func = sr[29] ^ sr[ 5] ^ sr[ 3] ^ sr[ 0];
   29 : psrg_func = sr[28] ^ sr[26];
   28 : psrg_func = sr[27] ^ sr[24];
   27 : psrg_func = sr[26] ^ sr[ 4] ^ sr[ 1] ^ sr[ 0];
   26 : psrg_func = sr[25] ^ sr[ 5] ^ sr[ 1] ^ sr[ 0];
   25 : psrg_func = sr[24] ^ sr[21];
   24 : psrg_func = sr[23] ^ sr[22] ^ sr[21] ^ sr[16];
   23 : psrg_func = sr[22] ^ sr[17];
   22 : psrg_func = sr[21] ^ sr[20];
   21 : psrg_func = sr[20] ^ sr[18];
   20 : psrg_func = sr[19] ^ sr[16];
   19 : psrg_func = sr[18] ^ sr[ 5] ^ sr[ 1] ^ sr[ 0];
   18 : psrg_func = sr[17] ^ sr[10];
   17 : psrg_func = sr[16] ^ sr[13];
   16 : psrg_func = sr[15] ^ sr[14] ^ sr[12] ^ sr[ 3];
   15 : psrg_func = sr[14] ^ sr[13];
   14 : psrg_func = sr[13] ^ sr[ 4] ^ sr[ 2] ^ sr[ 0];
   13 : psrg_func = sr[12] ^ sr[ 3] ^ sr[ 2] ^ sr[ 0];
   12 : psrg_func = sr[11] ^ sr[ 5] ^ sr[ 3] ^ sr[ 0];
   11 : psrg_func = sr[10] ^ sr[ 8];
   10 : psrg_func = sr[ 9] ^ sr[ 6];
    9 : psrg_func = sr[ 8] ^ sr[ 4];
    8 : psrg_func = sr[ 7] ^ sr[ 5] ^ sr[ 4] ^ sr[ 3];
    7 : psrg_func = sr[ 6] ^ sr[ 5];
    6 : psrg_func = sr[ 5] ^ sr[ 4];
    5 : psrg_func = sr[ 4] ^ sr[ 2];
    4 : psrg_func = sr[ 3] ^ sr[ 2];
   default : psrg_func = sr[ 2] ^ sr[ 1];
 endcase
endfunction
```

The `function` must be included in the body of the `module`! A separate compilation is not possible! The `function` is used only for computations, no timing modelling is allowed there!

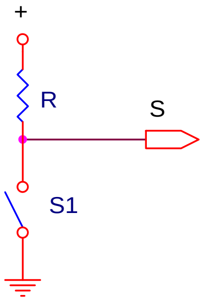
The feedback expressions for some sizes of the shift register, based on:

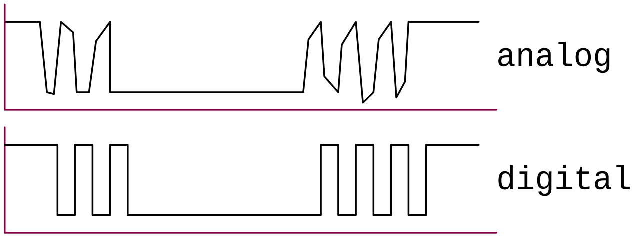
Xilinx APP 052 July 7, 1996, ver. 1.1
- table for n=3...168

The generator can not exit the state `"00..00"`!

Filter for noisy or slow signals(1)

```
module filt_shortv(clk, d, q);
parameter N = 3;

input clk, d;
output q;

reg q;

reg [N-1:0] samples;
wire all_high, all_low_n;

assign all_high  = & samples;
assign all_low_n = | samples;

always @(posedge clk)
begin
  samples <= {samples[N-2:0], d};
  q <= (q | all_high) & all_low_n;
end

endmodule
```

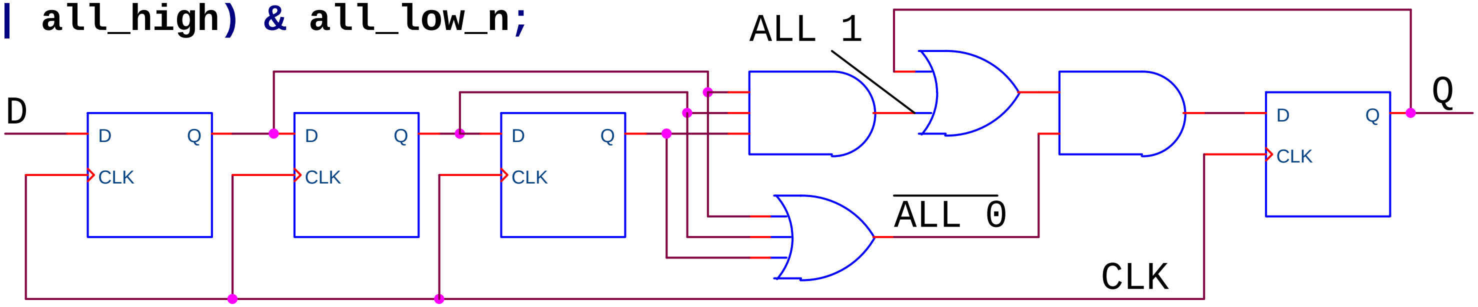




Using a shift register

The input is shifted in, the output is changed only if all bits in the shift register are equal

The clock frequency must be low enough, for mechanical buttons in the order of 100Hz or below



© V. Angelov

function
localparam

Filter for noisy or slow signals(2)

```
module filt_longv(clk, d, q);
parameter N = 3;
input clk, d;
output q;
```

the number of bits necessary to store ni

the type of the return

```
function integer log2;
input integer ni;
begin
   log2 = 0;
   while (ni > 0)
   begin
       log2 = log2 + 1;
       ni = (ni >> 1);
   end
end
endfunction
```

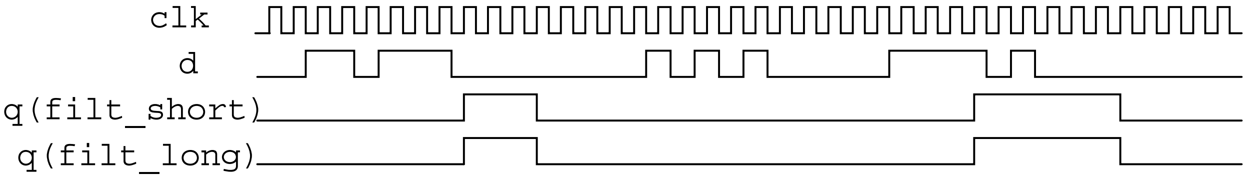
the type of the input

If more than one statement – use **begin end**!

Using a counter

The input signal **d** must remain unchanged for **N clk** cycles in order to be copied to the output **q**, otherwise the counter will be reloaded

```
localparam cnt_size = log2(N-2);
reg [cnt_size-1:0] counter;
reg [1:0] samples;
reg q;
always @(posedge clk)
begin
  samples <= {samples[0], d};
  if (^ samples) counter <= N-2;
  else if (counter > 0) counter <= counter - 1;
  else q <= samples[1];
end
endmodule
```



Mealy State Machine(1)

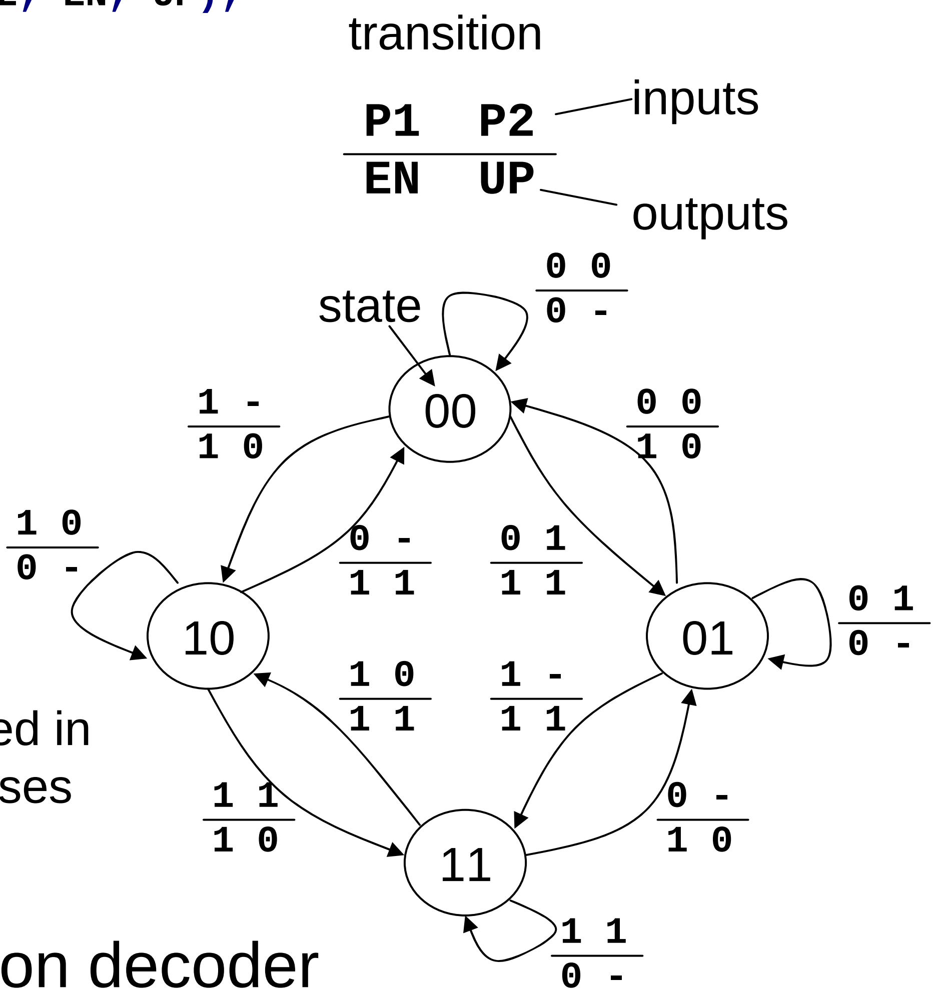
```
module state_2phv(clk, rst_n, P1, P2, EN, UP);

input  clk;
input  rst_n;
input  P1;
input  P2;
output EN;
output UP;

parameter S00 = 2'b00, S01 = 2'b01,
          S10 = 2'b10, S11 = 2'b11;

reg [1:0] present_st;
reg [1:0] next_st;

reg [1:0] p12s;
reg p1s, p2s;

reg EN, UP;
```

type of the ports

state names

the encoding

signals used in the processes



position decoder

Mealy State Machine(2)

Next state and outputs

```
always @(present_st or p1s or p2s)
begin
  EN <= 2'b0; UP <= 1'bx; // the default is no counting and UP = don't care
  next_st <= present_st;
  case (present_st)
  S00: if (p1s == 1'b1) begin
          next_st <= S10; EN <= 1'b1; UP <= 1'b0;
        end else if (p2s == 1'b1) begin
          next_st <= S01; EN <= 1'b1; UP <= 1'b1;
        end
  S01: if (p1s == 1'b1) begin
          next_st <= S11; EN <= 1'b1; UP <= 1'b1;
        end else if (p2s == 1'b0) begin
          next_st <= S00; EN <= 1'b1; UP <= 1'b0;
        end
  S11: if (p1s == 1'b0) begin
          next_st <= S01; EN <= 1'b1; UP <= 1'b0;
        end else if (p2s == 1'b0) begin
          next_st <= S10; EN <= 1'b1; UP <= 1'b1;
        end
  S10: if (p1s == 1'b0) begin
          next_st <= S00; EN <= 1'b1; UP <= 1'b1;
        end else if (p2s == 1'b1) begin
          next_st <= S11; EN <= 1'b1; UP <= 1'b0;
        end
  default : next_st <= 2'bxx;
  endcase
end
```

Note that for the case of no transition, the default values of the inputs and of the state are assigned at the beginning

Calculate the next state and the outputs as function of the present state and inputs

Mealy State Machine(3)

Register for the state and synchronization of the inputs

```
always @(posedge clk)
begin
    p1s <= P1; // synchronize the inputs
    p2s <= P2; // to the Mealy machine!
    p12s = {p1s, p2s};
    if (rst_n == 1'b0) // jump to the correct state
        case (p12s)
        S00 : present_st <= S00;
        S01 : present_st <= S01;
        S10 : present_st <= S10;
        S11 : present_st <= S11;
        default : present_st <= 2'bxx;
        endcase
    else
        present_st <= next_st; // store the next state
end
endmodule
```

this is very important!

this is no register!

a somehow unusual reset,
asynchronous would be dirty

Verilog – VHDL

	Verilog	VHDL
data types	only built-in	complex, abstract, flexible, own definitions possible
packages	no: only include files	yes: for functions, constants, types etc.
generate	no (only conditional)	yes (for …)
attributes	no	yes
strictly typed	no: automatic conversions and declarations	yes: conversion functions, all signals must be declared
easy to learn and to write	yes: syntax similar to C, short code	no: syntax similar to ADA and PASCAL, longer code