Logic synthesis

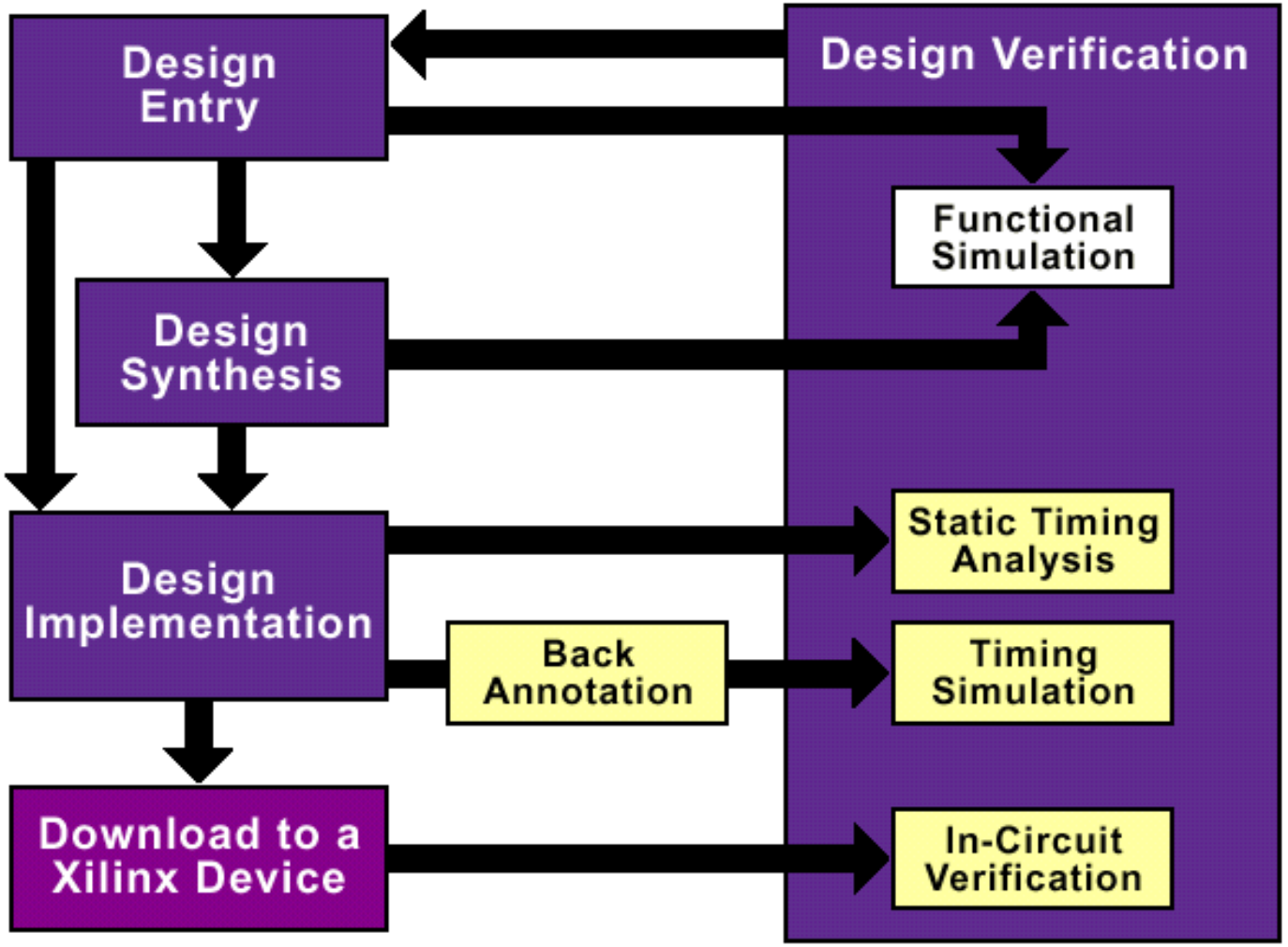
Agenda

- Introduction to the synthesis
- Synthesis of the basic elements
- Hardware representation of VHDL objects
- Synthesis of complex circuits
- Synthesis of types
- Non-synthesizable structures

Literature

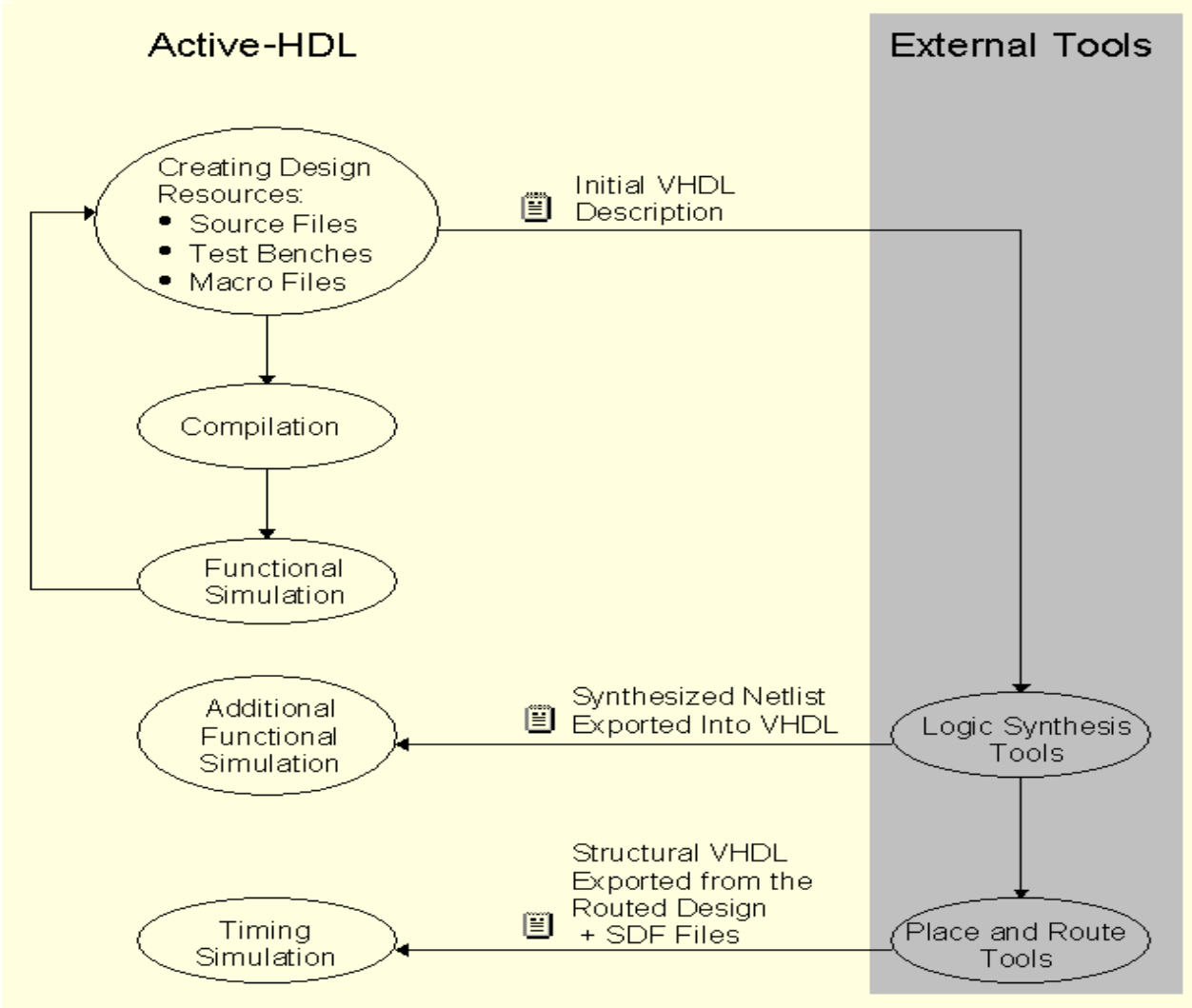
- **„*A VHDL Synthesis primer*” J.Bhasker,**
- **„*VHDL A Logic Synthesis Approach*” D.Naylor, S.Jones,**
- **„*VHDL Coding and Logic Synthesis with SYNOPSIS*” W.F.Lee,**
- **„*Reuse Methodology Manual*” M.Keating, P.Bricaud,**
- **„*Synthesis and Simulation Design Guide” (Xilinx manual)***
- **„*Xilinx Synthesis Technology (XST) User Guide” (Xilinx manual)***
- **„*Digital System Design with VHDL*” M.Zwoliński (translated: „*Projektowanie układów cyfrowych z wykorzystaniem języka VHDL*”)**

Synthesis & implementation
Xilinx CPLD/FPGA Design



Synthesis & implementation

Design in Active-HDL environment



Design in Active-HDL environment: Design Flow

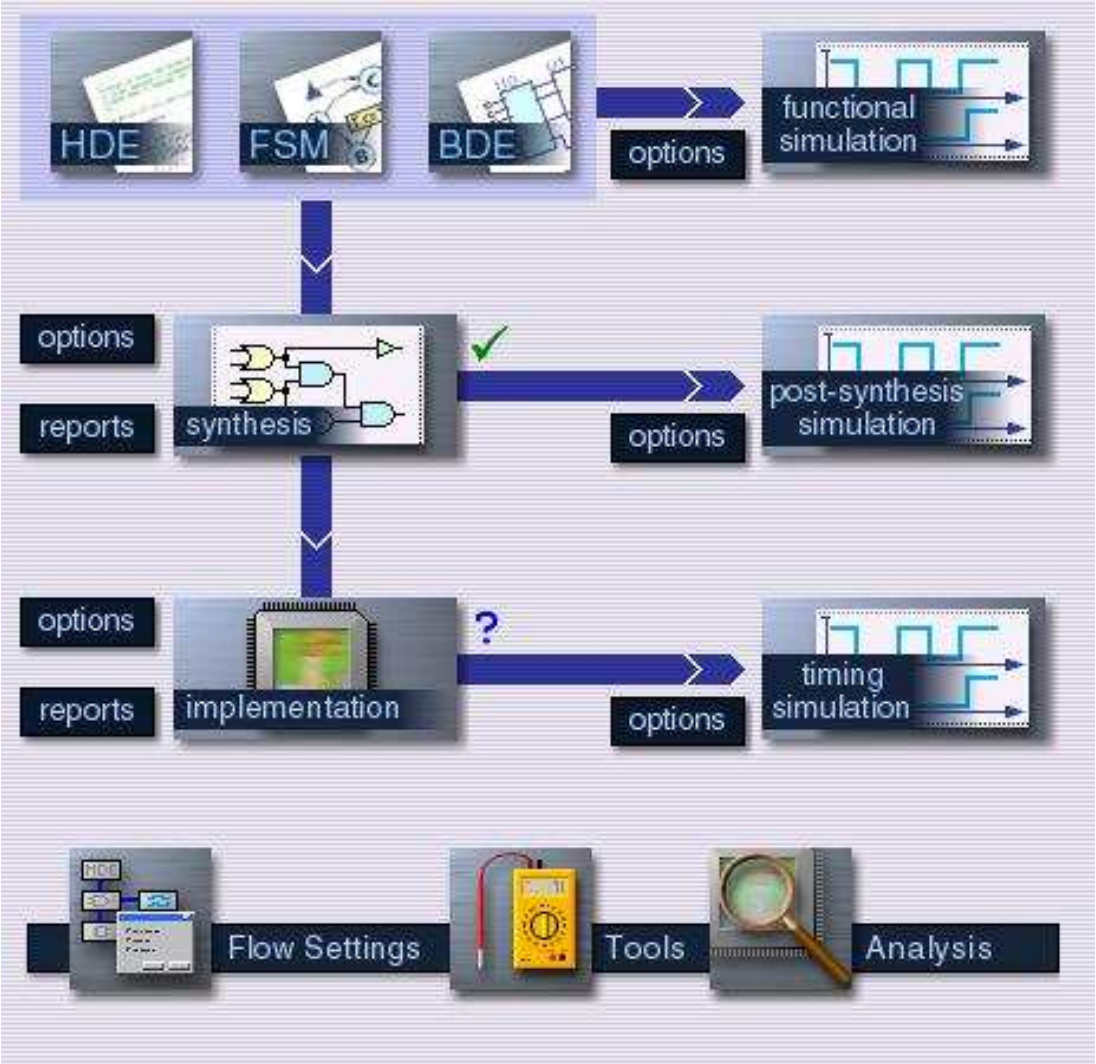
External tools for synthesis and implementation

Graphical shell by selecting option:

Tools/Preferences/ Environment/Flows/ Integrated Tools

and

View/Flow



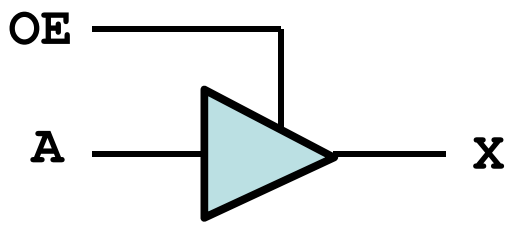
Synthesis of the basic elements
Arithmetic and logical operators

```
architecture behavioral of AND_gate is
begin
  process (A, B)                    -- synthesized AND gate
  begin
    if A = '1' and B = '1' then
      X <= '1';
    else
      X <= '0';
    end if;
end behavioral;

architecture inferred of AND_gate is
begin                               -- inferred AND gate
  X <= A and B;                     -- from the library of
end inferred;                       -- presynthesized elements
```

Inferred operators : `and or xor not nor nand xnor`
`+ - * / = /= > >= < <= ...`

Tri-state buffer

```
architecture behavioral of TRI_STATE_buffer is
begin
  process (A, OE)
  begin
    if OE = '1' then
      X <= A;
    else
      X <= 'Z';
    end if;
end behavioral;
```

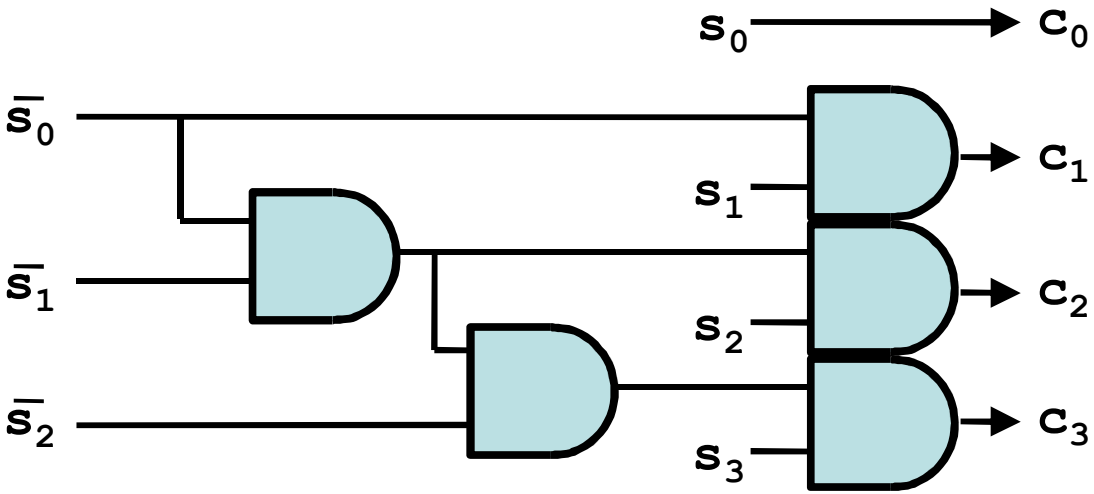



```
architecture inferred of TRI_STATE_buffer is
begin
  X <= A when OE = '1' else 'Z';
end inferred;
```

Priority encoder

```
architecture concurrent of PRIORITY_encoder is
begin
code <= "0001" when sel(0) = '1' else
        "0010" when sel(1) = '1' else
        "0100" when sel(2) = '1' else
        "1000" when sel(3) = '1' else
        "0000";
end concurrent;
```

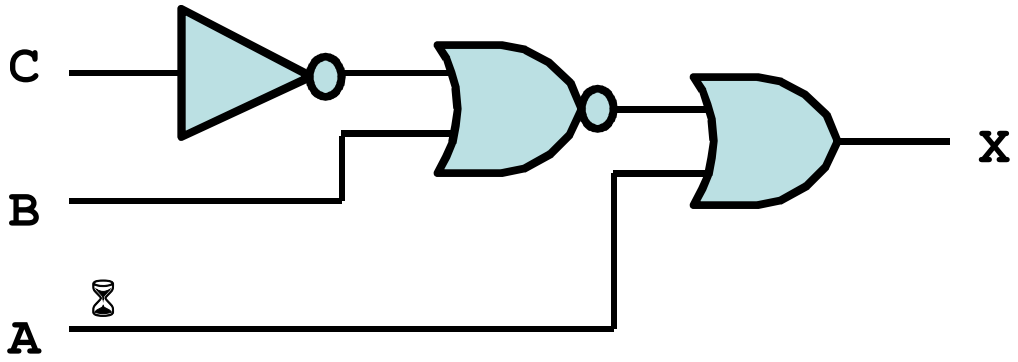
$$c_0 = s_0$$
$$c_1 = \bar{s}_0 s_1$$
$$c_2 = \bar{s}_0 \bar{s}_1 s_2$$
$$c_3 = \bar{s}_0 \bar{s}_1 \bar{s}_2 s_3$$



Note the logical functions implemented in LUTs!

Priority encoder

```
architecture behavioral of PRIORITY_encoder is
begin
  process (A, B, C)
  begin
    if A = '1' then          -- A: signal coming with
      X <= '1';              --    large delay
    elsif B = '1' then
      X <= '0';
    elsif C = '1' then
      X <= '1';
    else
      X <= '0';
    end if;
end behavioral;
```



Note the logical functions implemented in LUTs!

Hardware representation of VHDL objects

An object - signal or variable - may be represented as:

- **flip-flop**
  (edge-triggered memory element)
- **latch**
  (level-triggered memory element)
- **wire**
  (combinatorial element)

Hardware representation of VHDL objects

```
if C = 1 and
   C'event
then
   Q <= A;
end if;
```

signal Q:
flip-flop

```
if C = 1then
     Q <= A;
else
-- do nothing
end if;
```

signal Q:
latch

```
if C = 1 then
     Q <= A;
else
     Q <= B;
end if;
```

signal Q:
multiplexer

Synthesis of D-type flip-flop

Process containing sensitivity list:

```
process (CLK)
begin                                          --rising edge
  if CLK = '1' and CLK'event then
    Q <= A and B;
  end if;
end process;
```

Process containing `wait` statement

```
process
begin                                          --falling edge
  wait until not CLK = '1' and CLK'event;
    Q <= A and B;
end process;
```

Do not combine edge detection with other conditions:

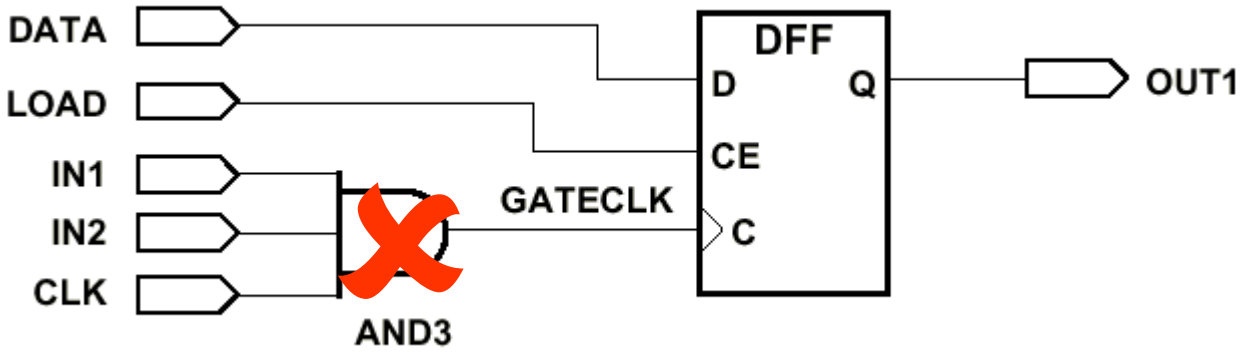
```
if CLK'event and CLK = '1' and CE = '1' then ...
```

Concurrent conditional assignment (not recommended):

```
Q <= A and B when CLK'event and CLK = '1';
```

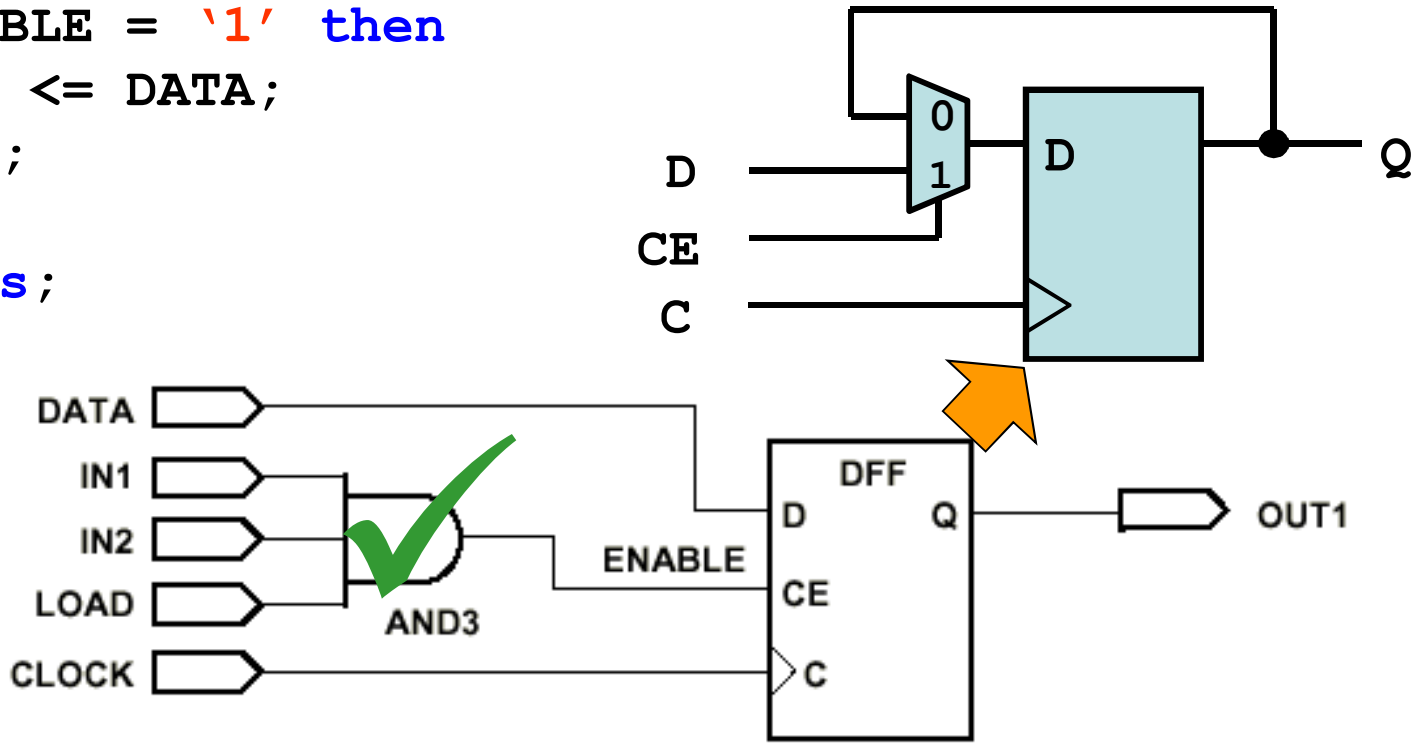
Flip-flop with Clock Enable input

```
GATECLK <= IN1 and IN2 and CLK;
process (GATECLK)
begin
  if GATECLK'event and GATECLK = '1' then
    if LOAD = '1' then
      OUT1 <= DATA;
    end if;
  end if;
end process;
```



Flip-flop with Clock Enable input

```
ENABLE <= IN1 and IN2 and LOAD;
process (CLK)
begin
  if CLOCK'event and CLOCK = '1' then
    if ENABLE = '1' then
      OUT1 <= DATA;
    end if;
  end if;
end process;
```



Synthesis of Set / Reset inputs

```
process(CLK)
begin
  if CLK and CLK'event then
    if SET = '1' then      -- synchronous SET
      Q <= '1';
    else
      Q <= A and B;
    end if;
  end if;
end process;

process(CLK, RESET)      -- RESET in sensitivity list!
begin
  if RESET = '1' then    -- asynchronous RESET
    Q <= '0';
  elsif CLK and CLK'event then
    Q <= A and B;
  end if;
end process;
```

Synthesis of D-type latch

Process containing sensitivity list :

```
process (CLK, A, B)       -- list of all signals...
begin                     -- ...used in the process
  if CLK = '1' then       -- active high
    Q <= A and B;
  end if;
end process;
```

Concurrent conditional assignment (not recommended):

```
Y <= A and B when CLK = '1';
```

Unintentional synthesis of latch

```
process (A, B, C, SEL)
begin
  if SEL = "00" then
    Y <= A;
  elsif SEL = "01" then
    Y <= B;
  elsif SEL = "10" then
    Y <= C;
  end if;
end process;
```

Problem:

synthesis of latch as a result of incomplete specification of options (for `SEL = "11"` combination, the last value will be hold by default, which will imply a memory element).

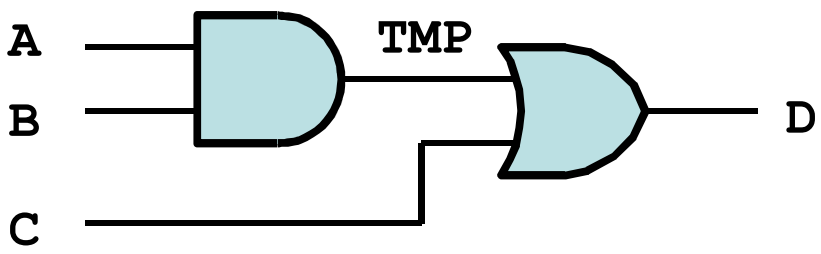
```
process (A, B, C, SEL)
begin
  if SEL = "00" then
    Y <= A;
  elsif SEL = "01" then
    Y <= B;
  elsif SEL = "10" then
    Y <= C;
  elsif SEL = "11" then
    Y <= '0';
  end if;
end process;
```

Solution:

specifying all the possibilities (also in the `case` statement - here you can use the *default* specification clause `when others`).

Signals and variables

```
signal A,B,C,D: bit
...
NO_MEMORY: process (A,B,C)
 variable TMP: bit;
begin
      TMP:= A and B;
      D <= TMP or C;
end process;
```



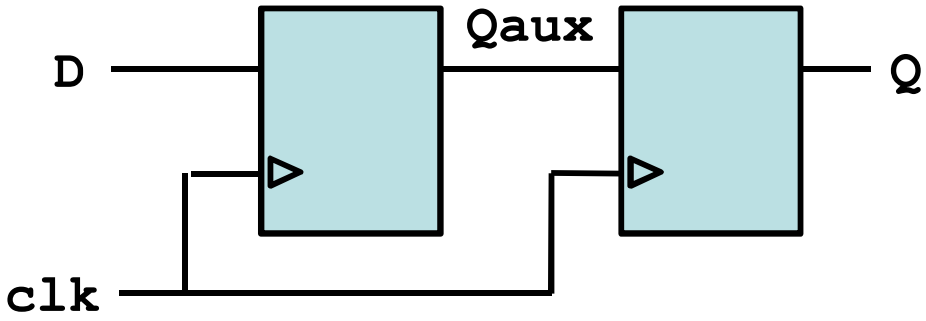
Synthesis of TMP : wire

```
signal A,B,C: bit
...
IS_IT_LATCH: process (A,B,C)
 variable TMP: bit;
begin
      C <= TMP and B;
      TMP := A or C;
end process;
```

Synthesis of TMP : latch ?!

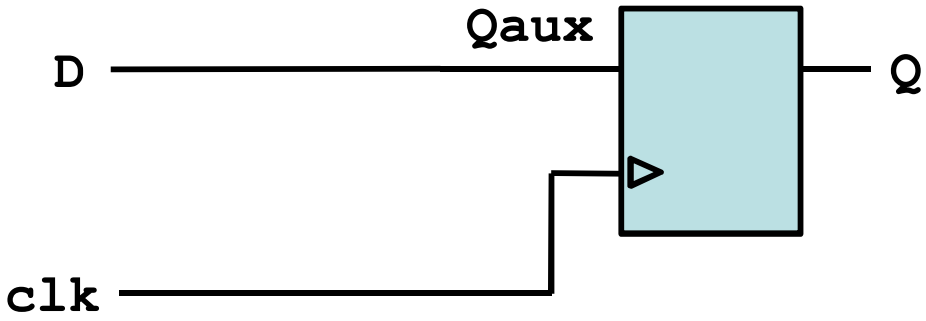
Signals and variables

```
  signal Qaux: ...
FFx2: process (clk)
begin
  …
  Qaux <= D;
  Q <= Qaux;
end process;
```



Synthesis : 2 flip-flops

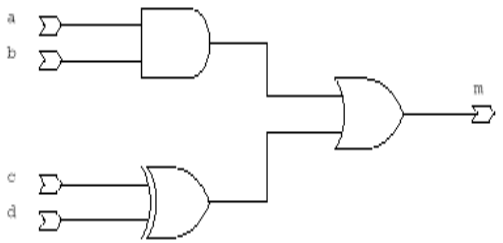
```
FFx1: process (clk)
  variable Qaux: ...
begin
  …
  Qaux := D;
  Q <= Qaux;
end process;
```



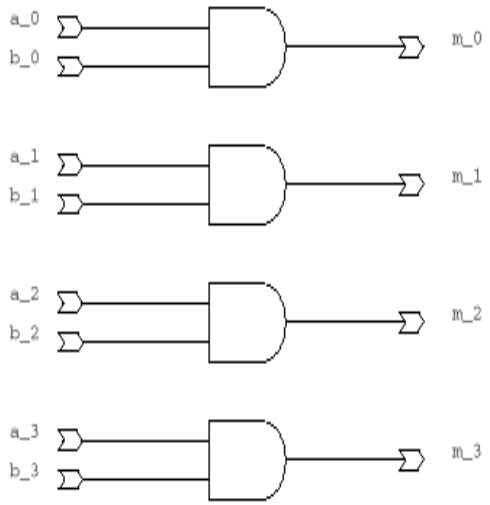
Syntesis : 1 flip-flop

Logical operators

```
entity logical_ops is
  port (a, b, c, d: in bit;  m: out bit);
end logical_ops;

architecture example of logical_ops is
  signal e: bit;
begin
  m <= (a and b) or e;
  e <= c xor d;
end example;
```

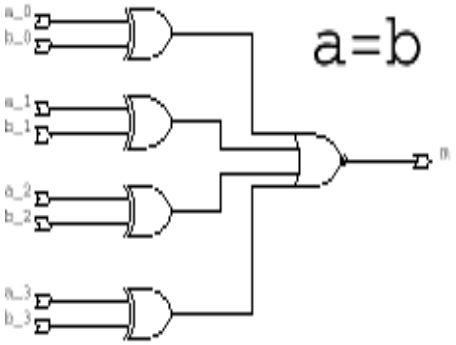



```
entity logical_bit is
  port (a, b: in bit_vector (0 to 3);
        m: out bit_vector (0 to 3));
end logical_bit

architecture example of logical_bit is
begin
  m <= a and b;
end example;
```

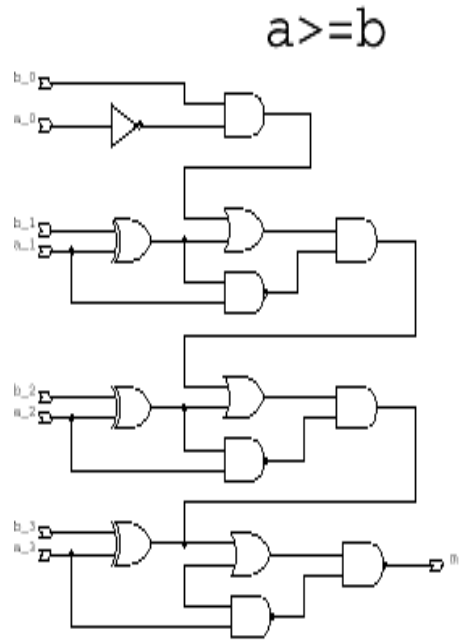


Relational operators

```
entity relational_equ is
  port (a, b: in bit_vector (0 to 3);
        m: out boolean);
end relational_equ;

architecture example of relational_equ is
begin
  m <= a = b;
end example;
```



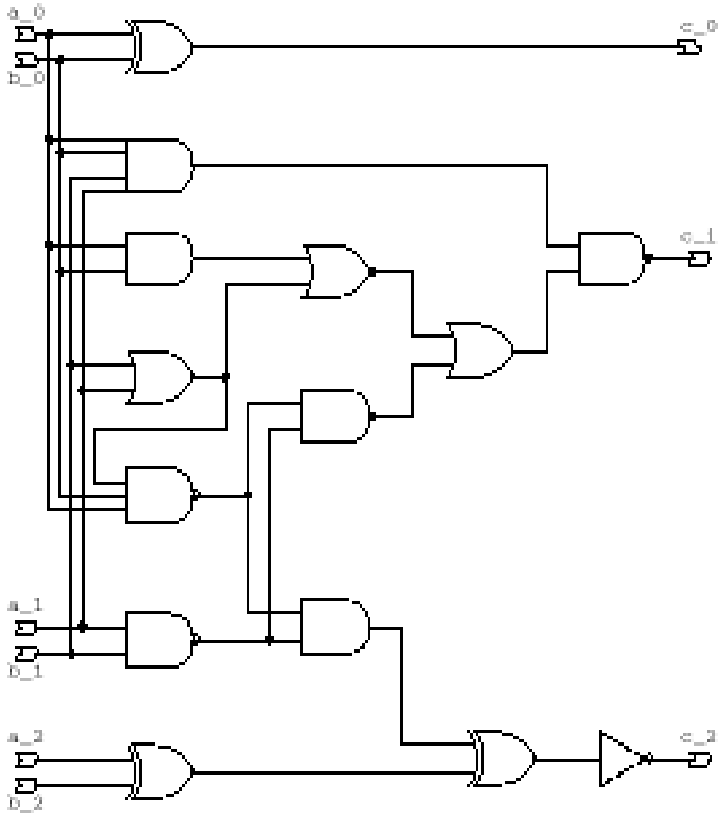

```
entity relational_mag is
  port (a, b: in integer range 0 to 15;
        m: out boolean);
end relational_mag;

architecture example of relational_mag is
begin
  m <= a >= b;
end example;
```



Arithmetic operators

```
package example_arithmetic is
  type small_int is range 0 to 7;
end example_arithmetic;
use work.example_arithmetic.all;

entity arith is
port (a, b: in small_int;
      m: out small_int);
end arith;

architecture example of arith is
begin
  m <= a + b;
end example;
```

" + "



Note the hard-macros of adders (and multipliers!)

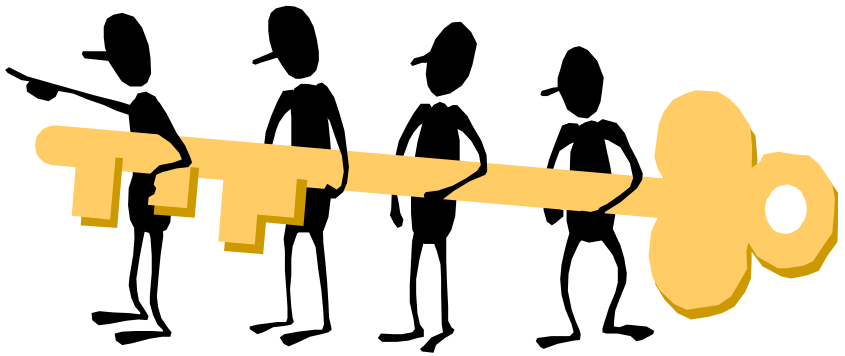
Selection statements

Sequential selection statements

- **conditional signal assignment:** `if...`
- **selected signal assignment:** `case...`

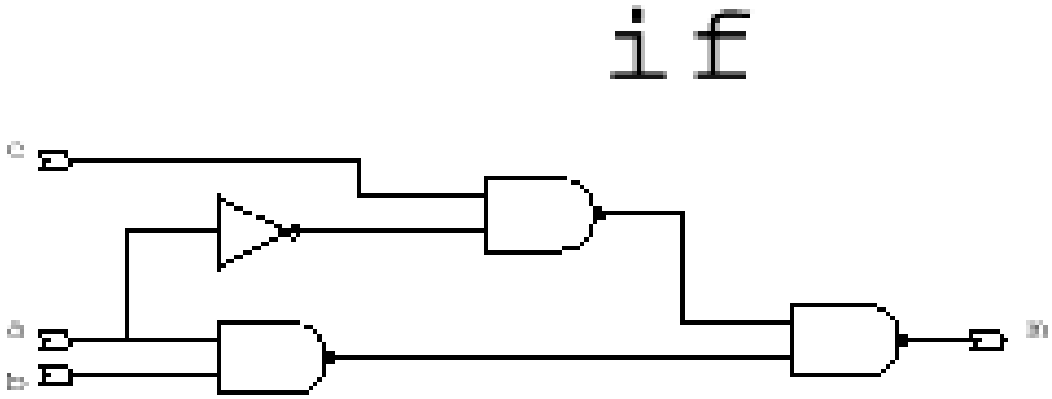
Concurrent selection statements

- **conditional signal assignment:** `when...`
- **selected signal assignment:** `with...`



Sequential selection statements: if

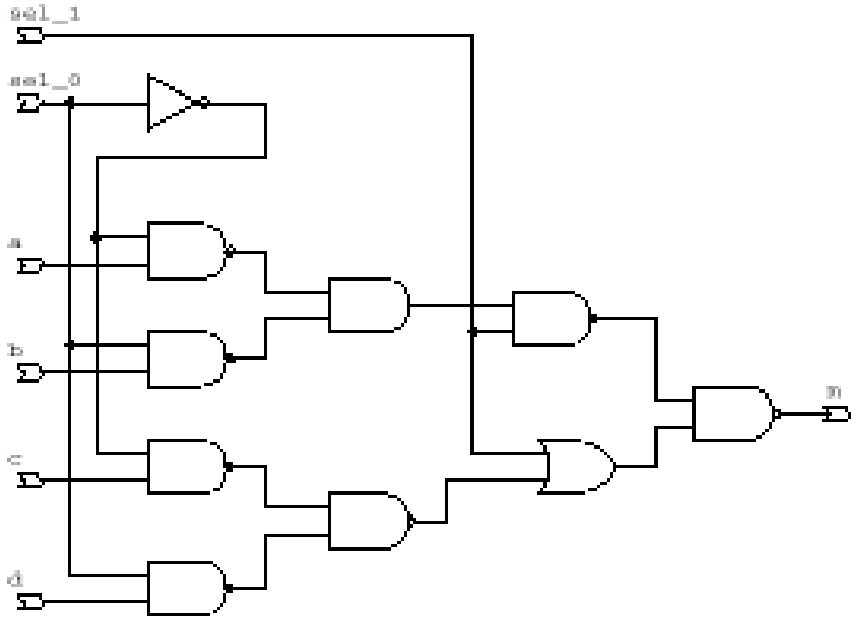
```
entity control_stmts is
port (a, b, c: in boolean; m: out
boolean);
end control_stmts;

architecture example of control_stmts
is
begin
  process (a, b, c)
    variable n: boolean;
    begin
    if a then
      n := b;
    else
      n := c;
    end if;
    m <= n;
  end process;
end example;
```



Sequential selection statements : case

```
entity control_stmts is
port (sel: bit_vector (0 to 1); a,b,c,d: bit;
      m: out bit);
end control_stmts;

architecture example of control_stmts is
begin
  process (sel,a,b,c,d)
  begin
  case sel is
    when b"00"   => m <= c;
    when b"01"   => m <= d;
    when b"10"   => m <= a;
    when others => m <= b;
  end case;
end process;
end example;
```

case



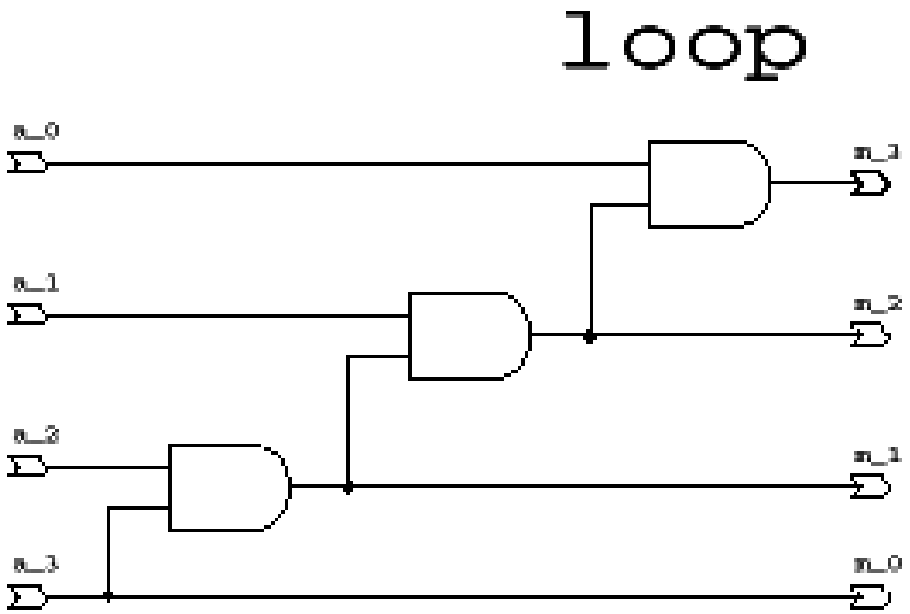
Concurrent selection statements : when / with

```
entity control_stmts is
  port (a, b, c: in boolean; m: out boolean);
end control_stmts;
architecture example of control_stmts is
begin
  m <= b when a else c;
end example;
```

```
entity control_stmts is
  port (sel: bit_vector (0 to 1); a,b,c,d: bit; m: out bit);
end control_stmts;
architecture example of control_stmts is
begin
  with sel select
       m <= c when b"00",
       m <= d when b"01",
       m <= a when b"10",
       m <= b when others;
end example;
```

Logic replication – loop

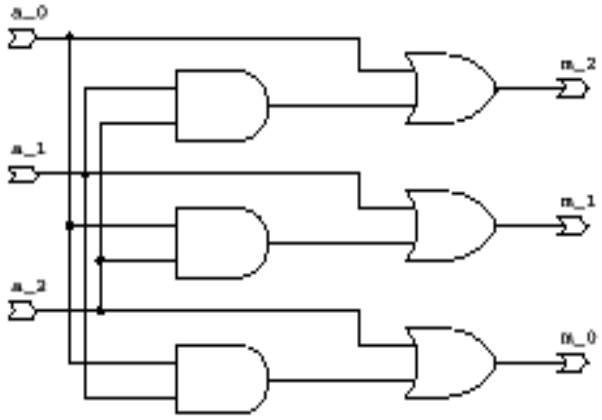
```
entity loop_stmt is
port (a: bit_vector (0 to 3);
      m: out bit_vector (0 to 3));
end loop_stmt;

architecture example of loop_stmt is
begin
process (a)
  variable b: bit;
  begin
    b := '1';
    for i in 0 to 3 loop
      b := a(3-i) and b;
      m(i) <= b;
    end loop;
end process;
end example;
```



Logic replication - subprogram

```
entity subprograms is
port (a: bit_vector (0 to 2);
      m: out bit_vector (0 to 2));
end subprograms;

architecture example of subprograms is

function simple (w, x, y: bit) return bit is
begin
  return (w and x) or y;
end;

begin
  process (a)
  begin
    m(0) <= simple(a(0), a(1), a(2));
    m(1) <= simple(a(2), a(0), a(1));
    m(2) <= simple(a(1), a(2), a(0));
  end process;
end example;
```

subprogram



Shifters

Sequential (shift register):

- **with concatenation operator (&)**

```
shreg <= shreg (6 downto 0) & SI;
```

- **with loop statemnet for...loop**

```
for i in 0 to 6 loop
  shreg(i+1) <= shreg(i);
end loop;
shreg(0) <= SI;
```

- **with shift operators (sll, srl, ...)**

Combinatorial (barrel shifter):

- **with shift operators (sll, srl, ...)**

```
with SEL select
  SO <= DI when "00",
        DI sll 1 when "01",
        DI sll 2 when "10",
        DI sll 3 when others;
```

- **with concatenation operator (&)**

Memories

- inferred or instantiated
- implemented as a distributed or block memories
  (depending on the size, speed and occupied area)
- synchronous (optionally with asynchronous read – distributed)
- RAM (also initialized) or ROM
- also used for combinatorial logic and FSMs

Method	Advantages	Disadvantages
Inference	• Most generic way to incorporate RAMs into the design, allowing easy/automatic design migration from one FPGA family to another • FAST simulation	• Requires specific coding styles • Not all RAMs modes are supported • Gives you the least control over implementation
CORE Generator software	Gives more control over the RAM creation	• May complicate design migration from one FPGA family to another • Slower simulation comparing to Inference
Instantiation	Offers the most control over the implementation	• Limit and complicates design migration from one FPGA family to another • Requires multiple instantiations to properly create the right RAM configuration

Synthesis of types
integer type

- **Types and subtypes, which may represent negative values in range, are encoded in the Two's Complement code.**
- **Types and subtypes, which represent only positive values in range, are encoded in the Natural Binary Code.**
- **The number of bits used depends on the largest allowable value for the object.**

```
-- binary encoding having 7 bits
  type int0 is range 0 to 100;
  type int1 is range 10 to 100;
-- 2's complement encoding having 8 bits (including sign)
  type int2 is range -1 to 100;
-- binary encoding having 3 bits
  subtype int3 is int2 range 0 to 7;
```

Synthesis of types
integer type

```
type short is integer 0 to 255;
subtype shorter is short range 0 to 31;
subtype shortest is short range 0 to 15;

signal op1, op2, res1: shortest;
signal res2: shorter;
signal res3: short;
begin
  res1 <= op1 + op2;  -- truncate carry
  res2 <= op1 + op2;  -- use carry
  res3 <= op1 + op2;  -- use carry and zero extend
```

Synthesis of types
Enumerated types

Declaration: **... is synthesized as:**

```
type direction is (left, right, up, down);          -- two wires
type cpu_op is (execute, load, store);              -- two wires
                        -- the encoding of 11 is a "don't care"

subtype mem_op is cpu_op range load to store;       -- two wires
                -- the encodings of 00 and 11 are "don't cares"

-- User Defined Encoding
attribute enum_encoding: string;
attribute enum_encoding of cpu_op: type is
  "001" &     -- execute
  "010" &     -- load
  "100";      -- store
```

Enumerated and other types

- During synthesis the enumerated types are coded binary by default. Subsequent elements (L) of the enumerated type receive subsequent values, the first from the left receives value zero.
- Number of bits (N) of an object that represents the enumerated type will be the smallest possible number, satisfying the condition: $L <= 2^N$

- **`bit`** & **`boolean`** types are synthesized as scalar
- **`character`** type is synthesized as 8-bit vector

Synthesis of types
`std_logic_1164` type

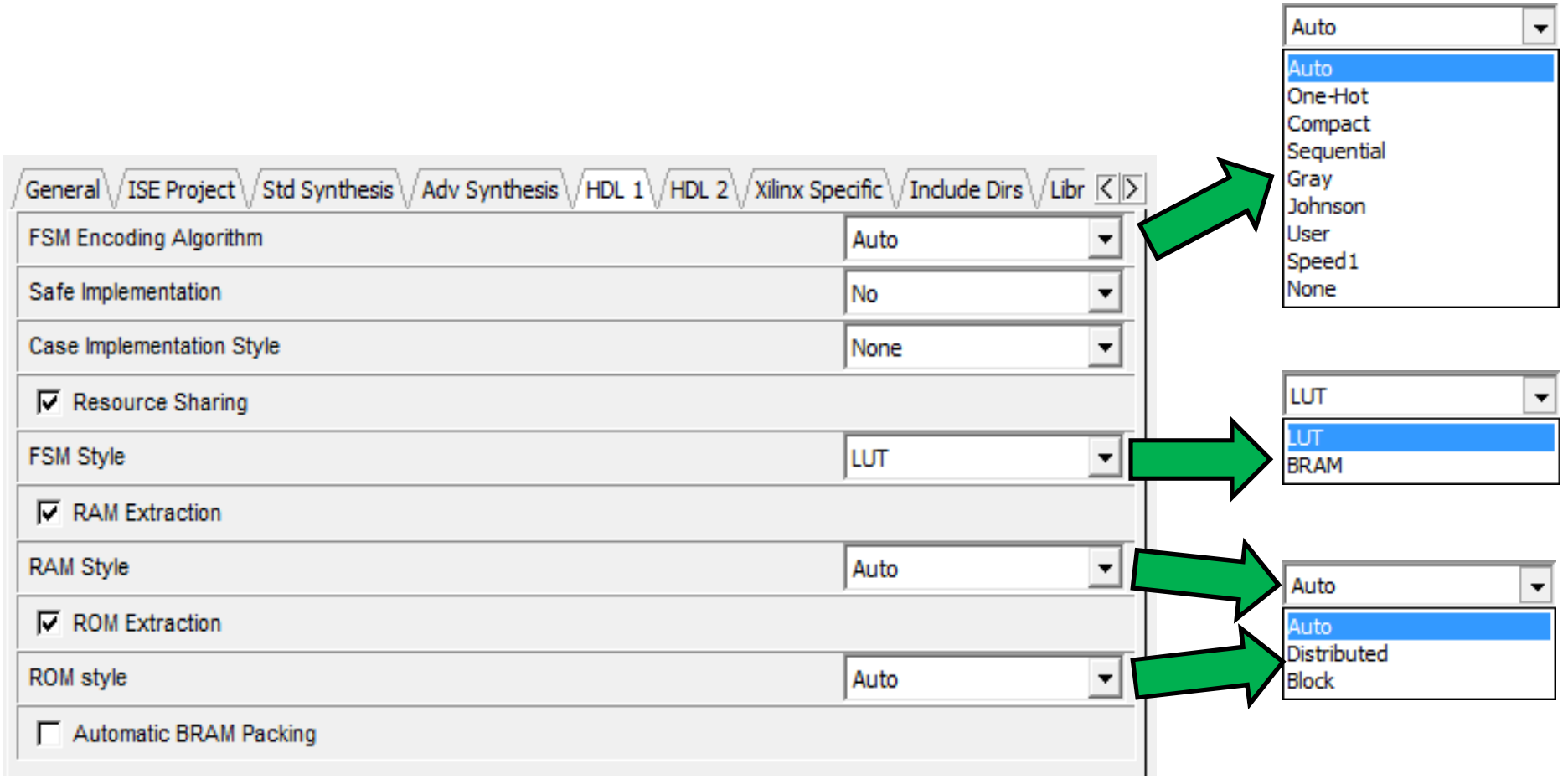
Recommended, because:

- large number of values (9), describing most of the real states in digital systems,
- is automatically initialized to the value of 'U' – this forces a designer to initialize objects explicitly.
  Do not overcome this feature by initialization of signals and variables in their declaration - circuit that cannot be initialized may be obtained after synthesis in the result of such an approach.
- easy integration with other modules - eg. `integer` type can be synthesized, but the simulation will require executing time-consuming conversion functions.
- **after synthesis and implementation there is only `std_logic` type**

Synthesis of FSMs
FSM states encoding algorithms

- *Auto*
  selects the needed optimization algorithms during the synthesis process
- *One-Hot*
  ensures that an individual state register is dedicated to one state. Only one flip-flop is active (hot) at any time. Very appropriate with most FPGAs where a large number of flip-flops are available. Also a good alternative to optimize speed or to reduce power.
- *Compact*
  minimizes the number of state variables and flip-flops. Appropriate when optimizing area.
- *Sequential*
  consists of identifying long paths and applying successive radix two codes to the states on these paths. Next state equations are minimized
- *Gray*
  guarantees that only one state variable switches between two consecutive states. Appropriate for controllers exhibiting long paths without branching
- *Johnson*
  much like Gray option. Shows benefits with FSM containing long paths with no branching.
- *User*
  causes the synthesis tool to use the encoding defined in the source file
- *Speed1*
  oriented for speed optimization. The number of bits for a state register depends on the particular FSM, but generally it is greater than the number of FSM states.

XST Synthesis Options



Non-synthesizable structures

- **timing clauses:**
  - **assignments (`after`, `transport`, `inertial`)**
  - **`wait for`**
- **floating-point data types (`real`)**
- **file operations – reduced:**
  - **`read`: memory initialization from file**
  - **`write`: debug**

To be continued…