VHDL
Synthesis

Franco Fummi
Alessia Bozzini

UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

Version 1.1

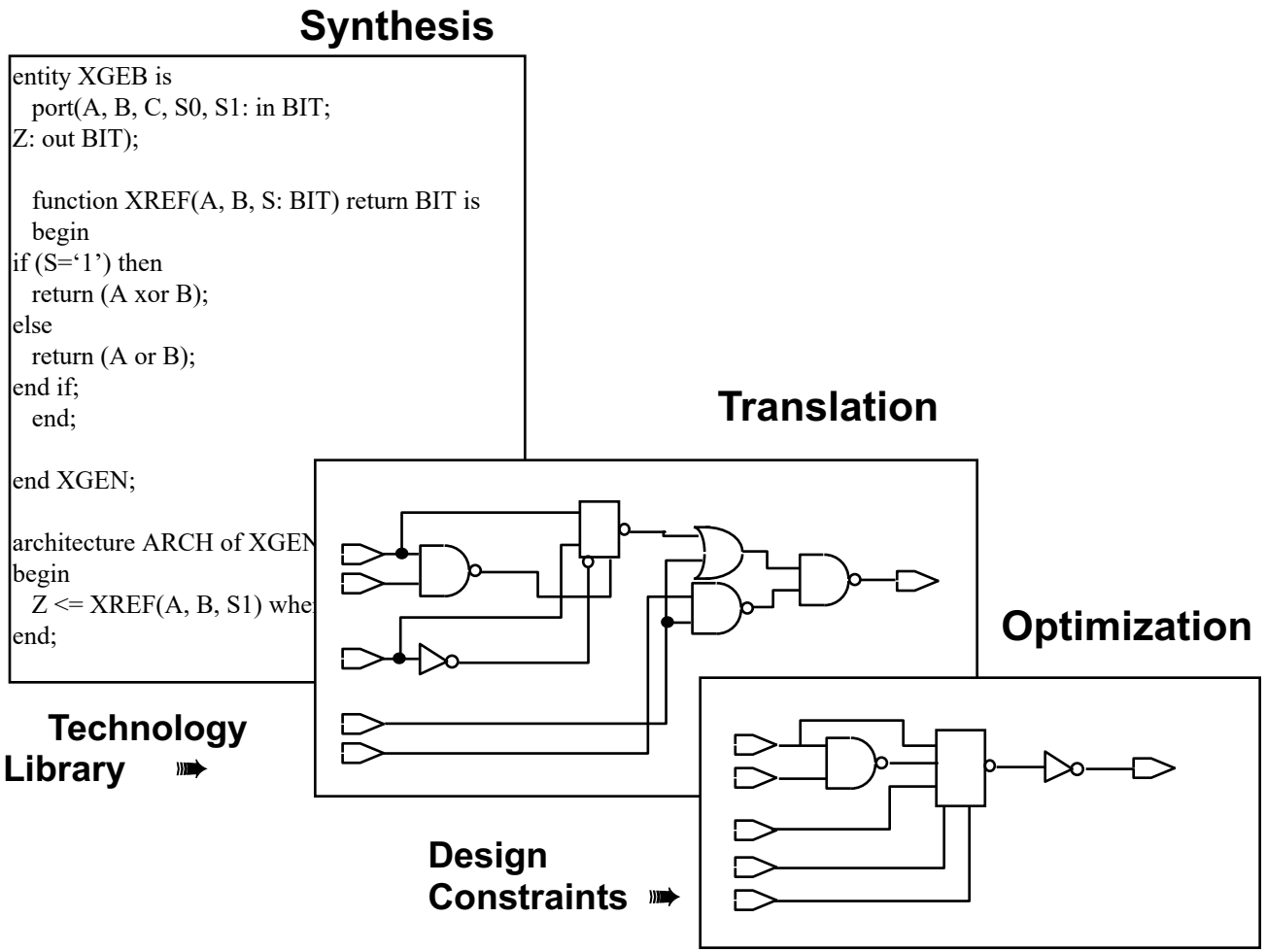
Contents

1. The synthesis process
2. Synthesis semantics
3. VHDL styles
   - behavioral
   - data flow
   - structural
4. Synthesis aspects
5. Synthesis restrictions
4. Synthesized circuits
   - Combinational synthesis
   - Sequential synthesis
     - 4 processes styles
5. FSMs in VHDL

Synthesis Process

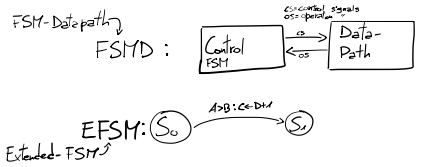
- Process of turning an abstract, technology independent, text description of a design into gates
- Two crucial steps:
  - Translation ⇒ CRUCIAL STEP (Eg: after the process, check if we have registers where there shouldn't be)
    - acts as the automated bridge between two levels of abstraction
    - In general, from an algorithmic-like description of the behavior to the register transfer level (RTL) to gate level
  - Optimization
    - technology-specific design transformation to meet user's goals for the given design
    - Introduction of user's constraints
    - Significant aspects: performance, area and test

Synthesis Process



Synthesis Process

- The logic synthesis process begins with a validated behavioral design description
- The design is translated into a design composed of:
  - control part, and
  - data path
- At the RTL level, the design description has been partitioned down to the implementation primitives of ALUs, ROMs, RAMs, data-path and control functions



Synthesis Process

- A second component to synthesis design policy in the VHDL style that has been adopted for the specification of the device.
- Different VHDL descriptions of the same functionality can yield radically different results.
- The more the VHDL description yields an implementation (structural description style) the easier the translation phase is; the higher the abstraction level, the more fundamental the translation phase is.

Synthesis Process

- VHDL was aimed at simulation purposes
  - The semantic is uniquely defined
    - The same code processed with two different commercial simulators will provide the same results
    - The simulation task requires no "interpretation" of the code
  - The semantics of VHDL is expressed in terms of a canonical simulator and not in terms of equivalent hardware constructions

Synthesis semantics

- VHDL lacks of synthesis semantics
  - The synthesis task requires an "interpretation" of the code to determine which hardware device can realize the desired function
- Being synthesis a difficult task, commercial tools usually restrict the set of accepted statements to a tool-dependent subset
  - No portability
  - No reusability

Synthesis semantics

- To obtain satisfable results from the synthesis process, the user has to 'deeply' know the commercial tool.
- A basic idea of what VHDL constructs lead to is also required.
- A "Guide to VHDL synthesized constructs" is useful for selecting the appropriate constructs and having a general feeling of the possible results.

STANDARD :

Level-0 VHDL Synthesis Subset

Synthesis semantics

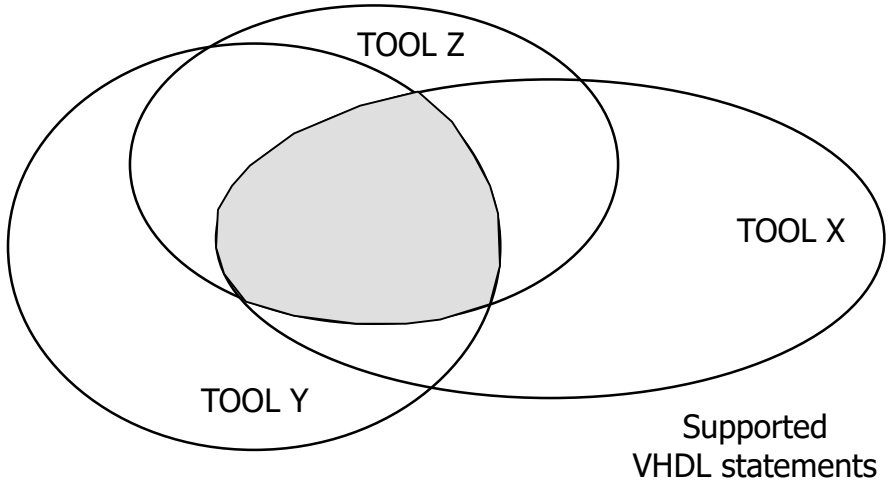
- VHDL Synthesis Subset
  - definition of a stable VHDL synthesis usage guide
  - allow portability
  - interchange format between tools, designers, companies …
  - allow VHDL to be used as an RT/logic specification language
  - facilitate reusability of descriptions

Synthesis semantics

- Stable VHDL synthesis usage guide
  - It is important to understand how the tool interprets the VHDL source code
  - It is useful to agree on what the expected results of the synthesis of the VHDL source code should be:
    - less confusion
    - no misunderstanding

Synthesis semantics

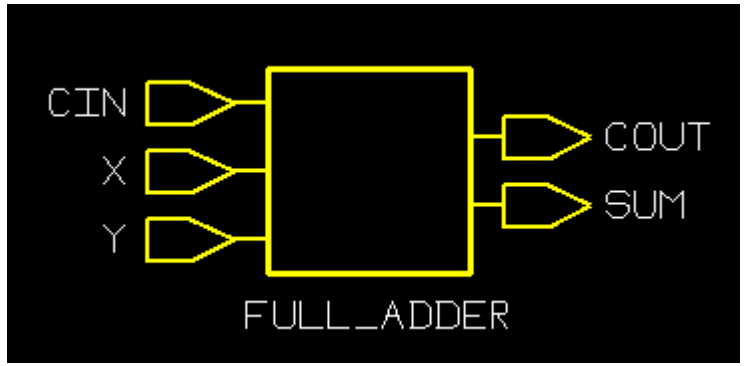
- VHDL portability



- Usually there are several possible styles to describe a behavior:
  - It is advisable to select the statement which is more widely adopted.

VHDL styles

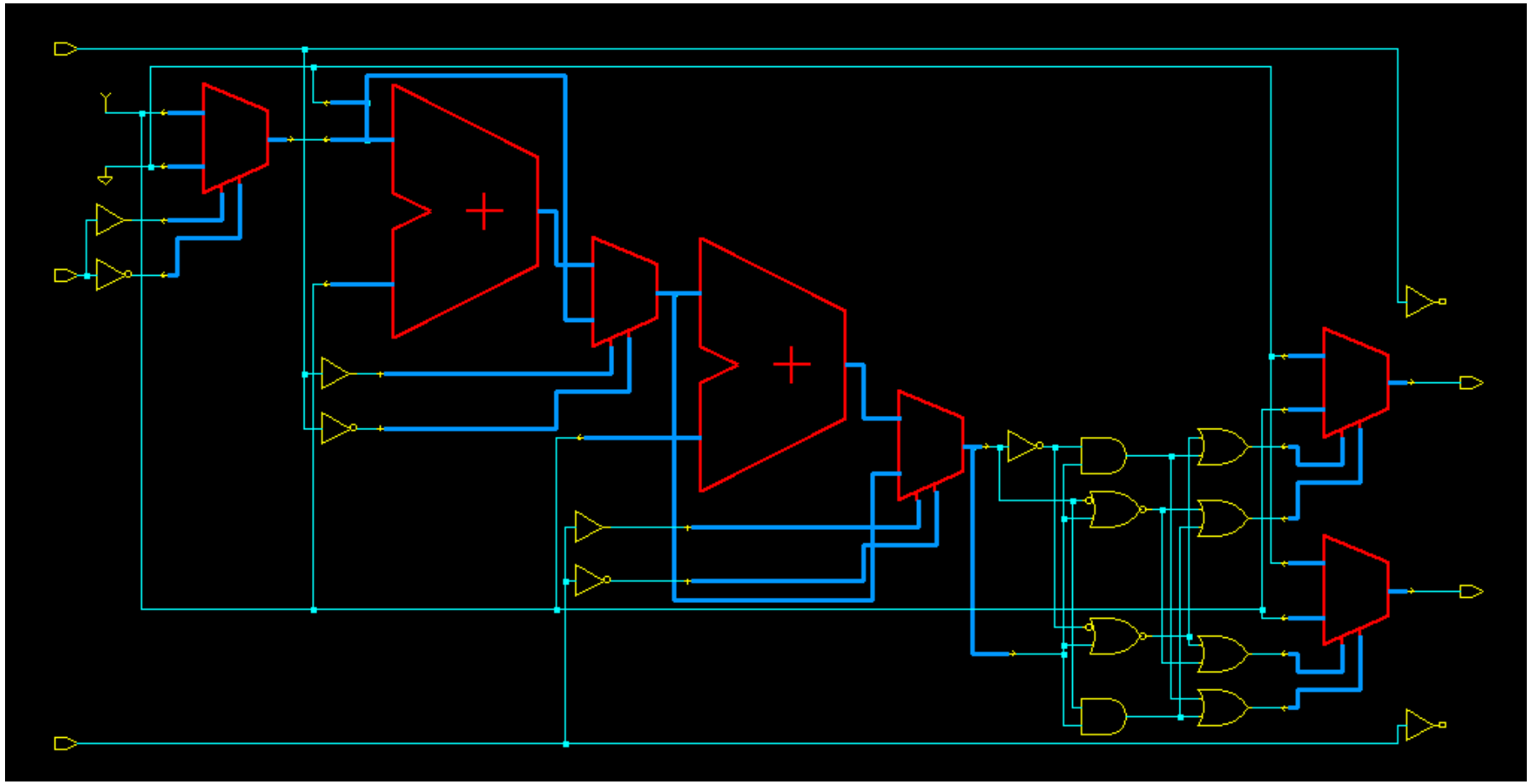
- Different “styles” describing the same behavior may lead to different realizations.
- An example: Full Adder
  1. behavioral
  2. data flow
  3. structural



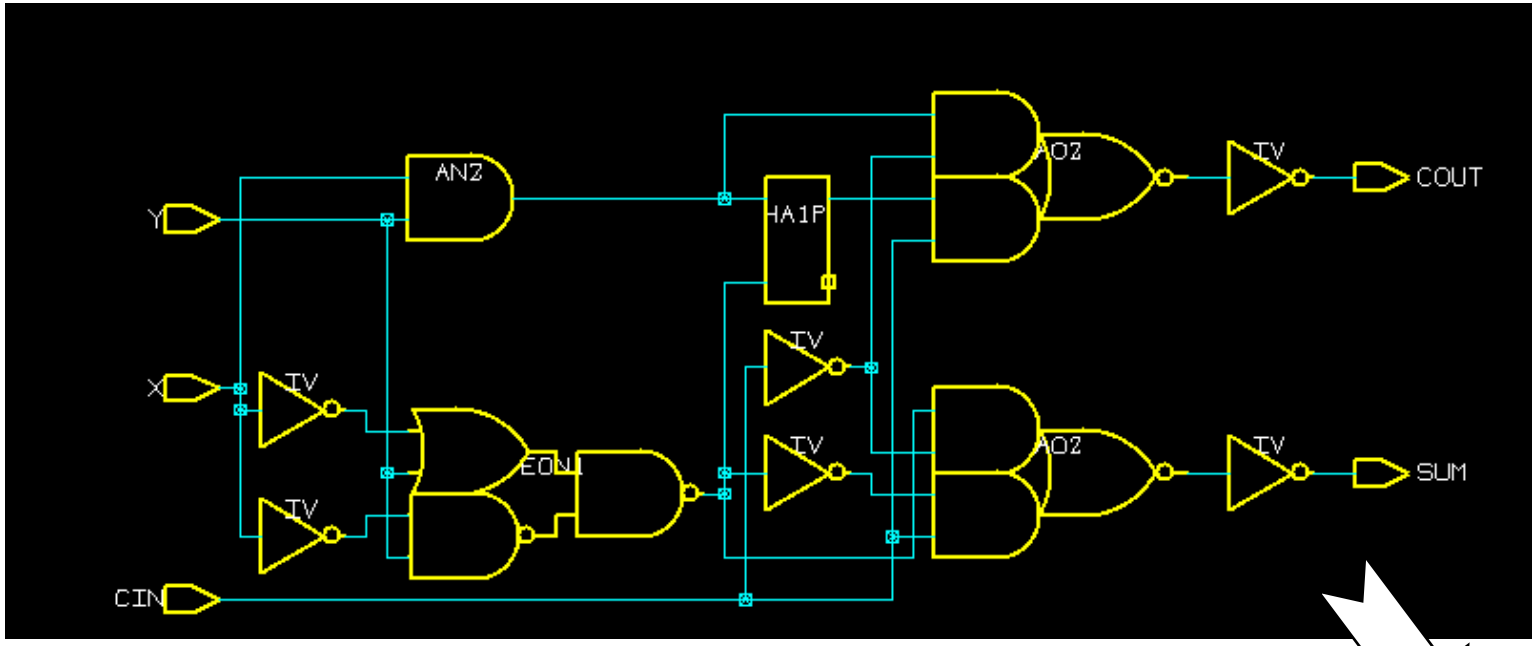
VHDL styles: behavioral

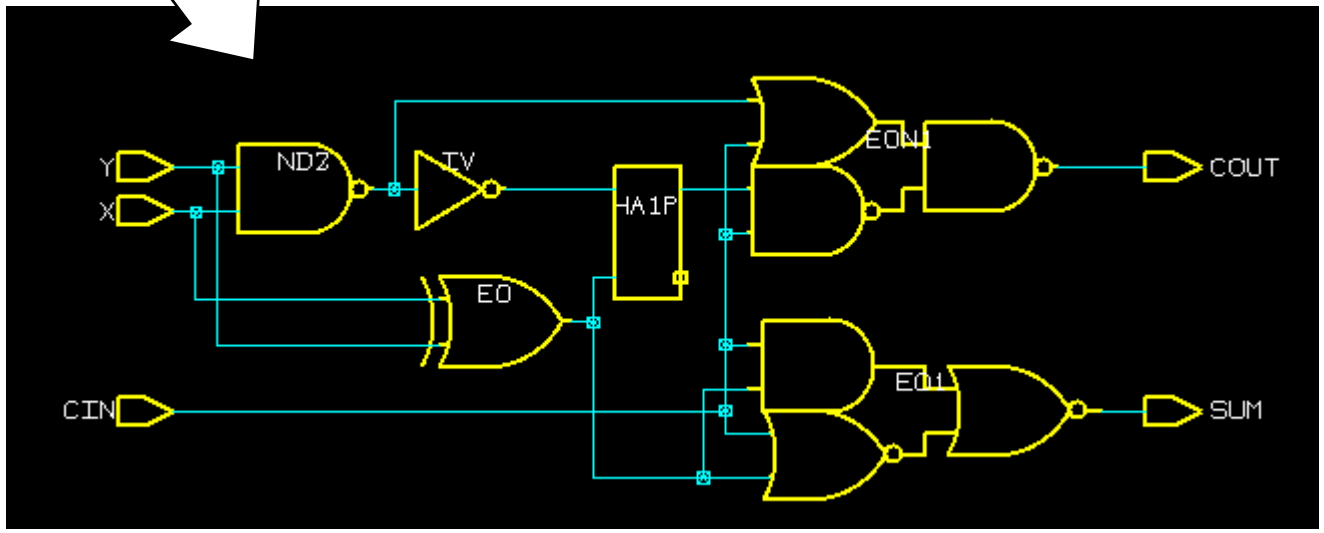
```
architecture FA_BEH of FULL_ADDER is
begin
   process (X, Y, CIN)
      variable BV: BIT_VECTOR(1 to 3);
      variable NUM: INTEGER range 0 to 3;
      variable STemp, CTemp: BIT;
    begin
          NUM := 0;
          BV := X & Y & CIN;
          for I in 1 to 3 loop
                if (BV(I) = '1') then
                      NUM := NUM + 1;
                end if;
          end loop;
          case NUM is
                when 0 => CTemp:='0'; STemp:='0';
                when 1 => CTemp:='0'; STemp:='1';
                when 2 => CTemp:='1'; STemp:='0';
                when 3 => CTemp:='1'; STemp:='1';
          end case;
          SUM <= STemp ;
          COUT <= CTemp ;
   end process;
end FA_BEH;
```

VHDL styles: behavioral



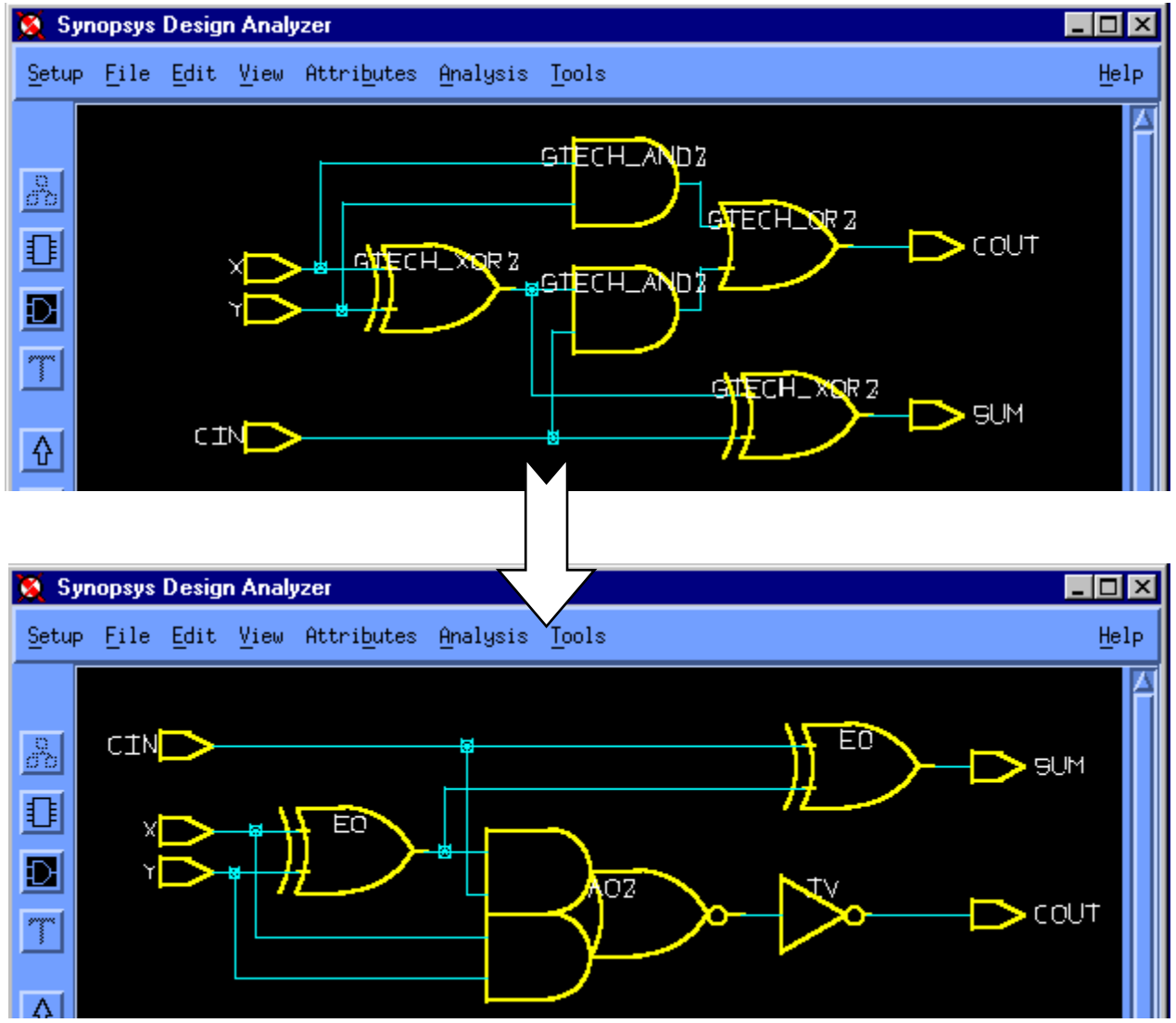
VHDL behavioral





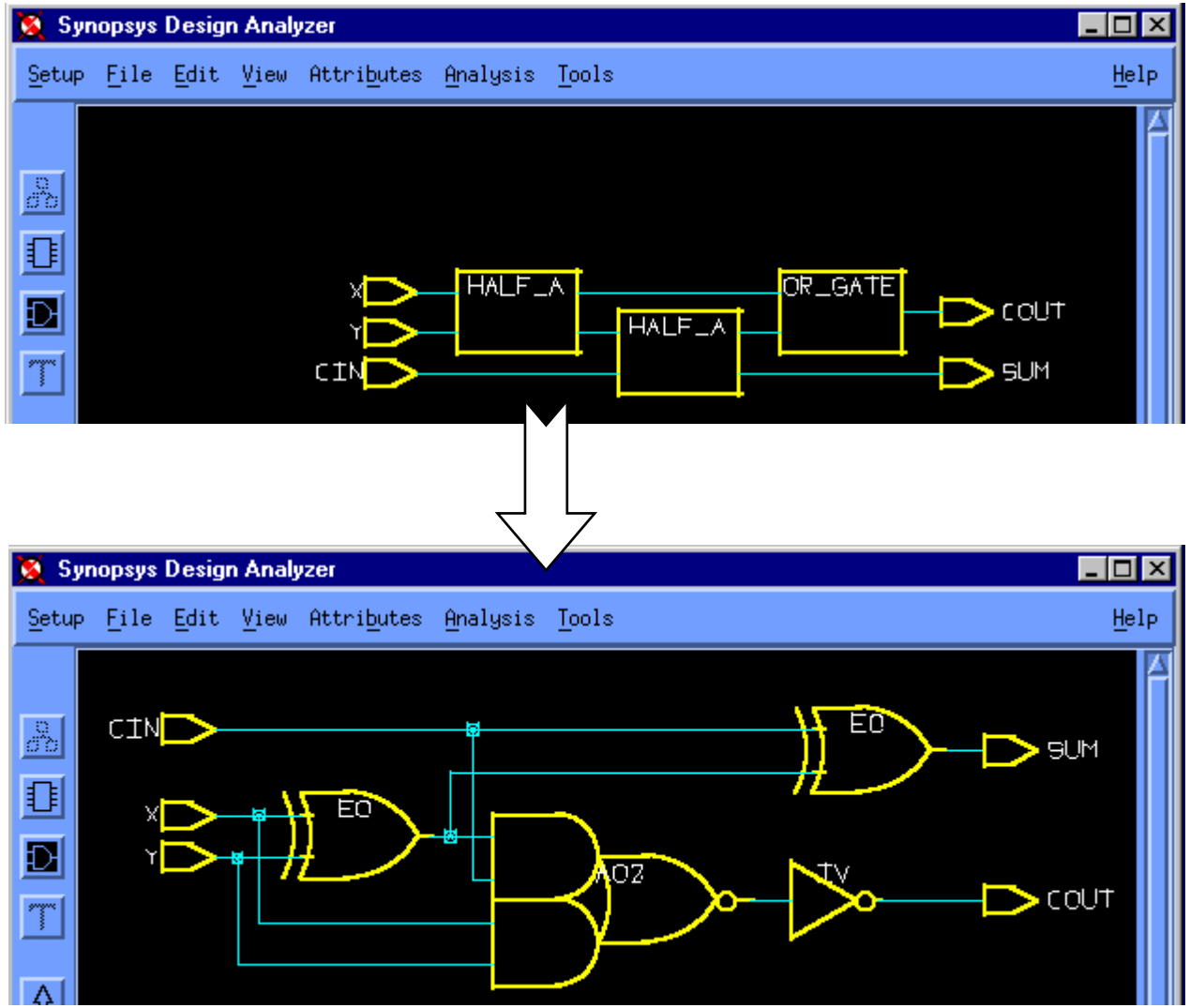
VHDL styles: data flow

```
architecture FA_BOOL of FULL_ADDER is
    signal S1, S2, S3: BIT ;
begin
     S1 <= X xor Y ;
     SUM <= S1 xor CIN after 1 ns;
     S2 <= X and Y ;
      S3 <= S1 and CIN after 1 ns;
     COUT <= S2 or S3 after 1 ns;
end FA_BOOL;
```



VHDL styles: structural

```
architecture FA_ST of FULL_ADDER is
     component HALF_A
        port(A, B: in BIT;
             S, C: out BIT);
     end component;
     component OR_GATE
        port(A, B: in BIT;
             O: out BIT);
     end component;
     signal C1, S1, C2: BIT;
begin
  HA1: HALF_A port map (A=>X, B=>Y, S=>S1, C=>C1);
  HA2: HALF_A port map (S1, CIN, SUM, C2);
  OR1: OR_GATE port map (C1, C2, COUT);
end FA_ST;
```



Synthesis aspects

- General aspects of some elements in relation with VHDL synthesis:
  - Vectors
  - Arithmetic, Logical & Relation operands
  - Subtypes, slices and functions

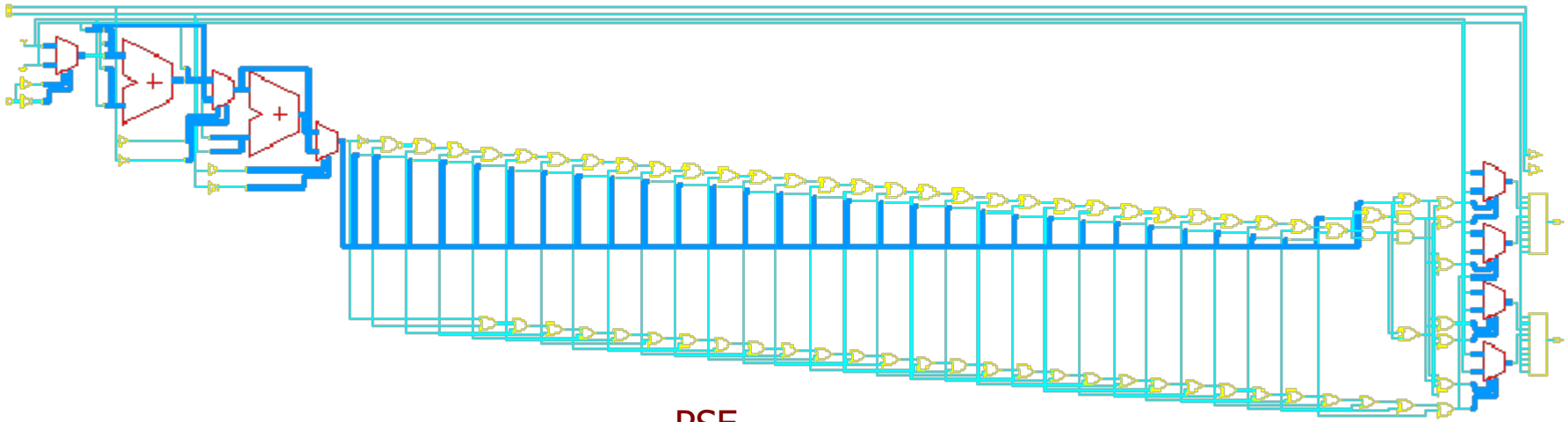
Synthesis aspects: vectors

```
entity ex1 is
    port(A, B, C: in bit_vector(1 to 5);
         Z: out bit_vector(1 to 5));
end ex1;
architecture arch of ex1 is
begin
    process(A,B,C)
     variable TEMP: bit_vector(1 to 5);
    begin
     TEMP := A and B;
     Z <= TEMP xor C;
    end process;
end;
```

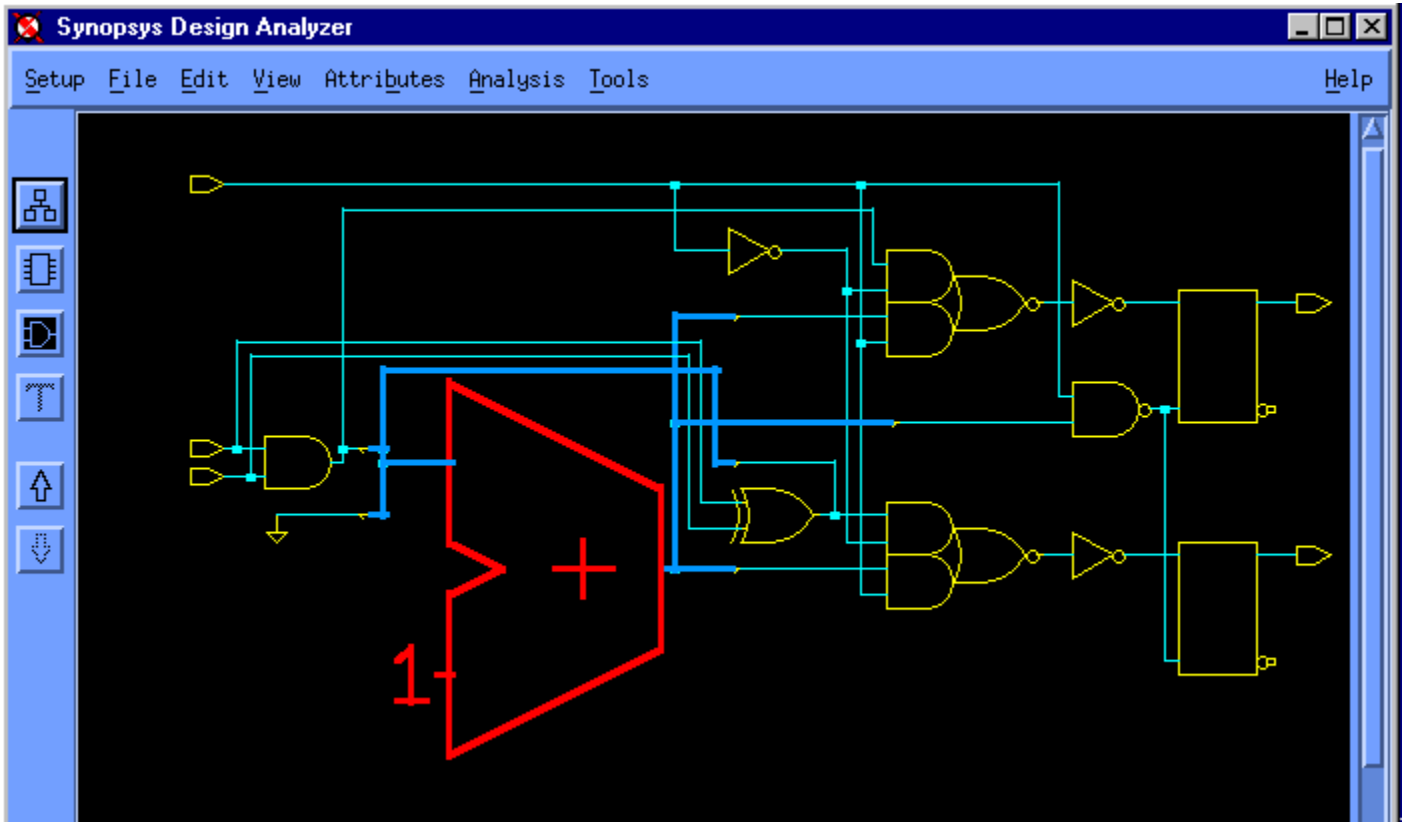
→ Bit-to-bit operations ⇒ I will have 5 "and" gates and 5 "xor" gates!!

Synthesis aspects: Turning Integer into Vectors

- Declare range for INTEGER objects for implementation of the minimum possible bits
- If a signal (or port) or variable is declared to be of type INTEGER without specifying a range, a tool may implement the object using a full 32 bits
  - The automatic tool warns the user of an unconstrained signal/variable declaration



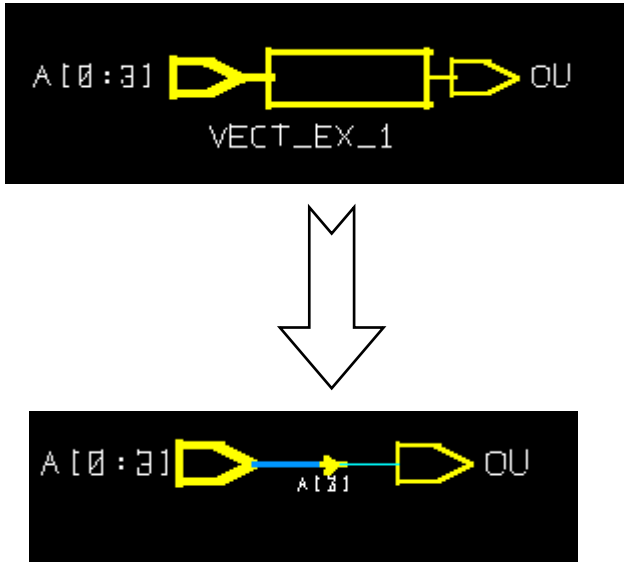
Synthesis aspects: Turning Integer into Vectors



Synthesis aspects: vector indexes

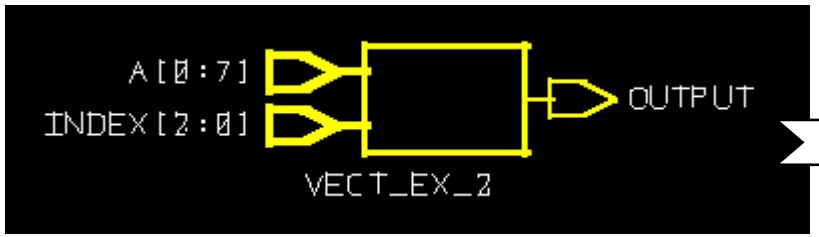
```
entity EX is
     port(A: in BIT_VECTOR(0 to 3);
          OU : out BIT) ;
end EX ;
```

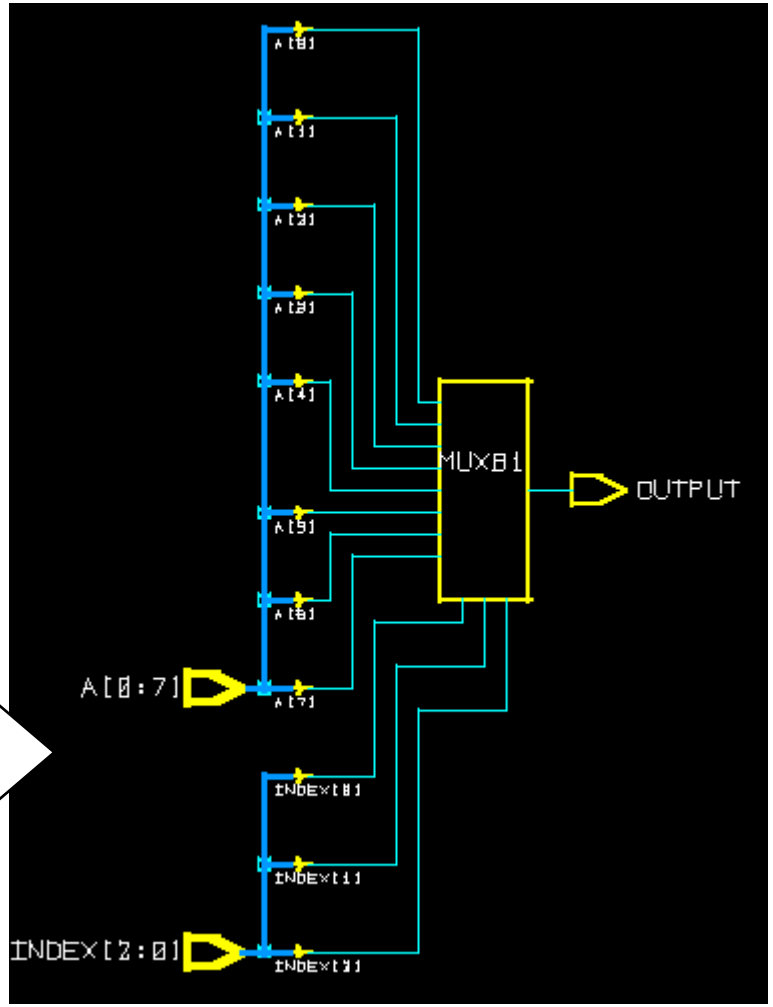
```
architecture EX_1 of EX is
begin
       OU <= A(2) ;
end EX_1 ;
```



Synthesis aspects: vector indexes

```
entity EX_VAR is
  port(A : in BIT_VECTOR(0 to 7) ;
       INDEX : in INTEGER range 0 to 7;
       OUTPUT : out BIT) ;
end EX_VAR ;
architecture EX_1 of EX_VAR is
begin
     OUTPUT <= A(INDEX) ;
end EX_1 ;
```





Synthesis aspects: vectors

- Types and subtypes
  - Supported types are:
    - Scalar
      - Enumerated
      - Integer
    - Composite
  - One dimensional arrays

VHDL Synthesis Subset

Synthesis aspects: Operators & Operands

- Arithmetic
  - abs, **, /, mod, rem are not supported
  - * supported only if:
    - both operands are constant
    - the second operand is a power of two
  - physical, real, string and null literals are not supported
- Relational

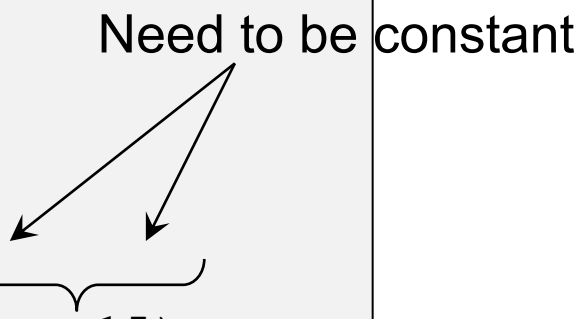
```
entity ex is
   port(A, B: in Integer range 0 to 15;
        Z: out boolean);
end ex;
architecture arch of ex is
begin
   Z <= (A < B);
end arch;
```

Usually supported
<, >, <=, >=, =, /=

! These operators may require a large amount of area

Synthesis aspects: Slices

```
package ops is
    subtype WORD is bit_vector(1 to 16);
    function asr (INP: WORD)
     return WORD;
end ops;
package body ops is
    function asr (INP: WORD)
     return WORD is
     variable RESULT: WORD;
    begin
     RESULT(1) := INP(1);
     RESULT(2 to 16) := INP(1 to 15);
     return RESULT;
    end;
end ops;
```



Synthesis aspects: functions

- The synthesizer creates hardware each time the function is used
- Recursion is bound to a constant and not all tools support it

```
entity EX is
       port(INPUT : in WORD ;
                    OUTPUT : out WORD) ;
end EX ;
architecture EX_1 of EX is
begin
                  OUTPUT <=   asr(asr(asr(INPUT))) ;
end EX_1 ;
```

calling 3 times the same function is not smart for the synthesis

Synthesis restrictions

- The synthesis subset suggests some restrictions on the following elements:
  1. WAIT statement
  2. LOOP statement
  3. BLOCK statement
  4. GENERATE statement

WAIT restrictions

- The UNTIL clause is the only supported clause:

```
WAIT UNTIL sig'EVENT AND sig= value;
```

- This statement has to be the first statement of the process.
- Not supported:
  - timeout clause (WAIT FOR 7ns)

LOOP restrictions

- The only allowed loop statements are FOR loops

```
FOR … IN … TO … LOOP
   sequence of statements
END LOOP;
```

- The discrete range in a FOR iteration scheme has to be static

Synthesizable loops are the ones that can be statically unrolled

GENERATE & BLOCK

- They are not supported
  - No GUARDED BLOCK
  - BLOCK statement is ignored
- The user must spend time to explicit what these constructs would immediately implement
- Instead of the block statement hierarchy should be used

Synthesized circuits

- How to obtain the “desired” kind of circuit:
  1. Combinational
  2. Sequential

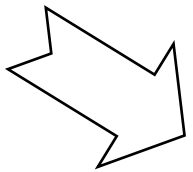
- Main problem:
  - Be sure not to write code that “introduces” memory elements

Combinational synthesis

- Combinational networks may be obtained by means of:
  - logical/arithmetical relations (data flow)
  - behavioral description
- Attention is focused on:
  - Assignment of all target signals in any possible situation, otherwise memory elements are necessary to store “old” values

Data flow

- Concurrent signal assignments
- It produces combinational networks UNLESS:
  - the waveform depends on the target signal, or
  - the signal assignment originate a combinational loop

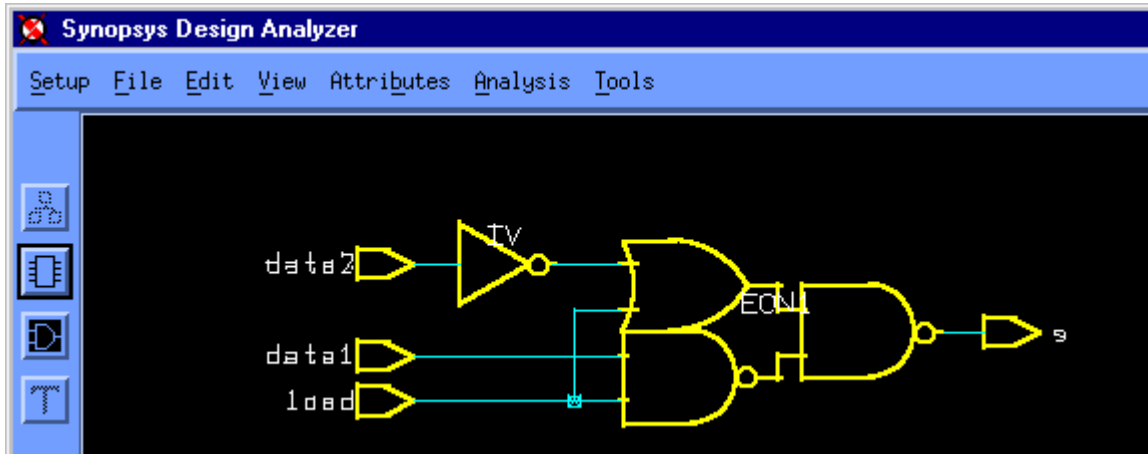


Asynchronous circuit

Data flow

Multiplexer 2x1

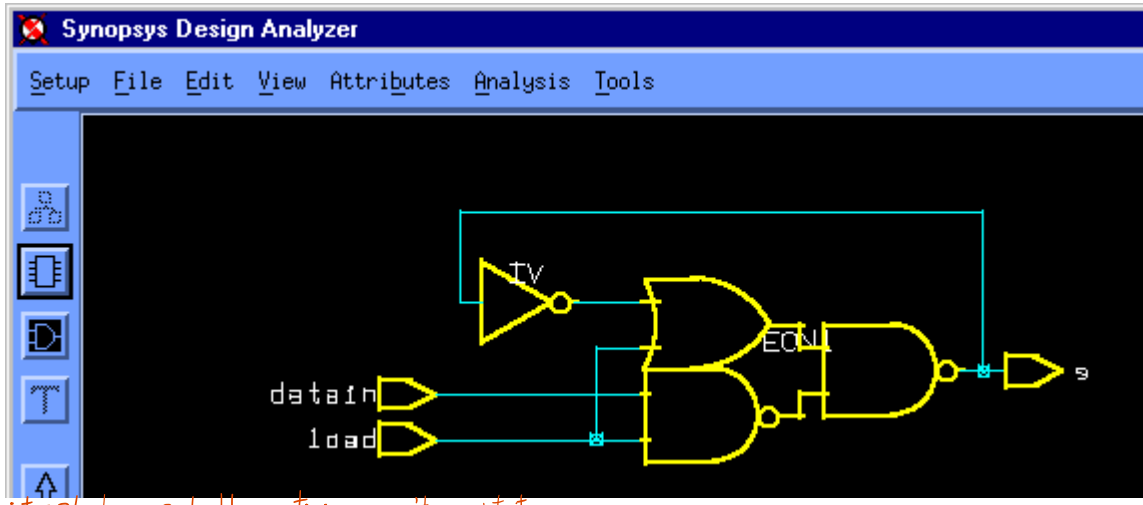
```
library IEEE;
use IEEE.Std_logic_1164.all;

entity MUX21 is
        port(load, data1, data2: in std_logic;
             s: out std_logic);
end MUX21;

architecture data_flow of MUX21 is
begin
        s <= data1 when (load='1') else data2;
end data_flow;
```



Multiplexer 2x1 with a feedback loop or a latch

```
library IEEE;
use IEEE.Std_logic_1164.all;

entity LATCH is
    port(load, datain: in std_logic;
         s: buffer std_logic);
end LATCH;

architecture data_flow of LATCH is
begin
    s <= datain when (load='1') else s;
end data_flow;
```

Not allowed because it's an internal loop, and the optimizer won't accept it

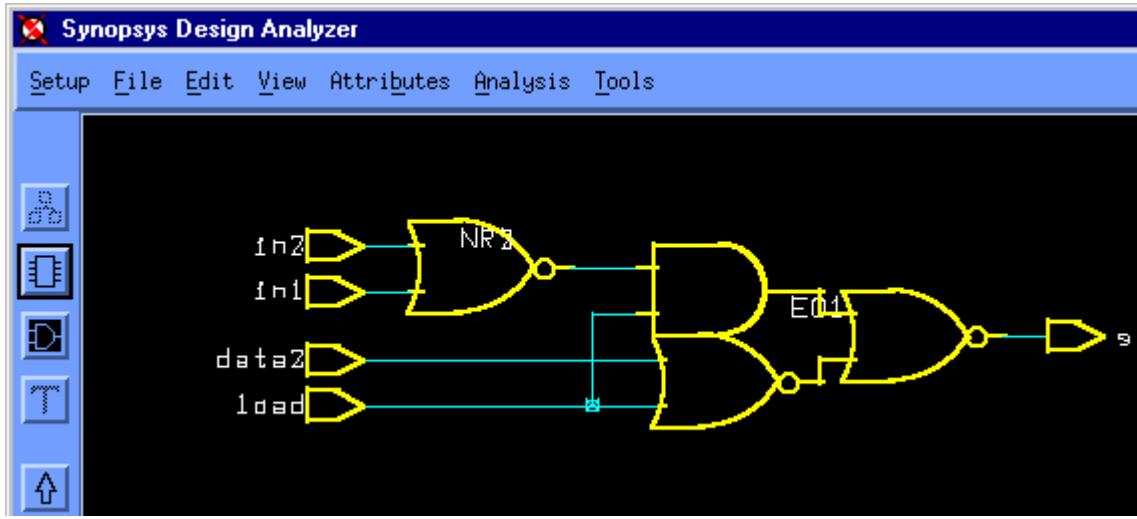


Waveform depends on target signal

Data flow

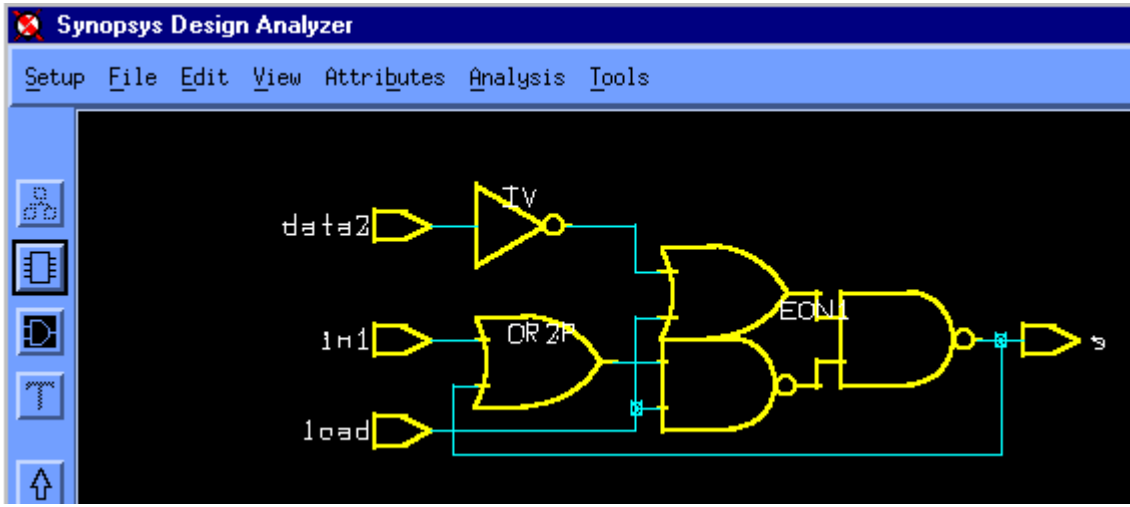
Multiplexer 2x1

```
library IEEE;
use IEEE.Std_logic_1164.all;
entity COMB is
        port(load, in1, in2, data2: in std_logic;
             s: out      std_logic);
end COMB;
architecture data_flow of COMB is
        signal data1: std_ulogic;
begin
        data1 <= in1 or in2;
        s <= data1 when (load='1') else data2;
end data_flow;
```



Multiplexer 2x1 with a feedback loop or a latch

```
library IEEE;
use IEEE.Std_logic_1164.all;
entity COMB is
      port(load, in1, in2, data2: in
std_logic;
           s: buffer    std_logic);
end COMB;
architecture data_flow of COMB is
      signal data1: std_ulogic;
begin
      data1 <= in1 or s;
      s <= data1 when (load='1') else data2;
end data_flow;
```



Waveform depends on target signal

Behavioral

- Statements contained in (one) process
- A process which model pure combinational logic has two main characteristics:
  - The sensitivity list contains all signals that are read inside the process
  - All signal and variables are assigned in all conditional branches of the process

Sequential synthesis

- Synchronous circuits
- One clock signal is identified and eventually a reset signal (synchronous or asynchronous) Only synchronous are synthesizable
- Behavioral description $\Longrightarrow$ process

Process styles

- Four styles of processes are envisioned:
  1. Processes with a sensitivity list including all read signals and assigning all signals and variables in all conditional branches
  2. Processes with a sensitivity list including all read signals and assigning all variables in all conditional branches
  3. Processes with a wait statement for detecting clock edges
  4. Processes with a sensitivity list including a clock signal and optionally an asynchronous reset signal

Process “style 1”

- Processes with a sensitivity list including all read signals and assigning all signals and variables in all conditional branches
  - The sensitivity list will contain all the intermediate signals
  - It models pure combinational logic

```
ENTITY ao IS
  port(i0, i1, i2: IN BIT;
         out0: OUT     BIT);
END ao;

ARCHITECTURE rtl OF ao IS
        SIGNAL result: BIT;
BEGIN
        PROCESS(i0, i1, i2, result)
        BEGIN
                result <= i0 AND i1;
                out0 <= result OR i2;
        END PROCESS;
END rtl;
```

Intermediate signal

If result is not inserted in the sensitivity list the synthesis would provide (when the tool provides a synthesis) a complex asynchronous sequential circuit with an event-triggered flip-flop

Process “style 1”

- Signal assignments are updated at the end of the process execution.
- Therefore `result` and `out0` are updated at the same time, `out0` accessing the old value of `result`
- To access the correct value of `result`, the process must be re-executed following an update of `result` to update `out0` to the correct value.
  This is done by placing the intermediate signal in the process sensitivity list.

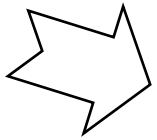
Process “style 1”

- Use of variables would resolve the problem of inserting in the sensitivity list intermediate signals
- Intermediate signals with no other specific functionality are inefficient

```
...
PROCESS(i0, i1, i2)
        VARIABLE result: BIT
BEGIN
        result := i0 AND i1;
        out0 <= result OR i2;
...
```

Process “style 2”

- Processes with a sensitivity list including all read signals and assigning all variables in all conditional branches



Model a mixture of pure combinational logic and asynchronous latches

- Latches are inferred when signals are not assigned in a conditional branch

Process “style 2”

```
entity ANDGATE is
        port(in1, in2: in std_logic;
             outp: out std_logic);
end ANDGATE;
architecture correct of ANDGATE is
begin
        process(in1, in2)
                variable x: std_logic;
        begin
                if (in1='1') then
                        x:=in2;
                else
                        x := '0';
                end if;
                outp <= x;
        end process;
end correct;
```

AND gate!

```
entity ANDGATE is
        port(in1, in2: in std_logic;
                       outp: out std_logic);
end ANDGATE;
architecture incorrect of ANDGATE is
begin
        process(in1, in2)
                variable x: std_logic;
        begin
                if (in1='1') then
                        x:=in2;
                end if;
                outp <= x;
        end process;
end correct;
```

A level-sensitive latch

Process “style 3”

- Processes with a WAIT statement as the first statement of a process
- It is a clocked circuit (synchronous sequential machine)
  - Finite State Machine (FSM)
    - Moore & Mealy

```
WAIT UNTIL clk’EVENT and clk = ‘1’
```

Process “style 4”

- Processes with a sensitivity list including a clock signal, and eventually an asynchronous reset signal
- An if statement constitutes the process, sensitive to the clock (and reset) events

```
Process (ck_name [, reset_name])
begin
       [if (reset_name = value) then
              … reset behavior
       els]if (ck_name = value and ck_name'event) then
              … clocked behavior
       end if;
end process;
```

Specifying FSMs

- Selection of:
  - Moore or Mealy style
  - Reset or not
  - Synchronous/asynchronous reset
  - Buffers position

  - *Data type definition for State elements*

FSMs in VHDL - Examples

- Two problems:
  1. Coin handler for can disposal
     - Accepts coins of 50, 100, Selection button
     - When at 150 allows selection and no change
     - When at 200 allows selection and change
  2. Bit sequence recognizer
     - Recognizes concatenated sequences
       - …0110…
       - …1110…
     - When recognized, output at 1.

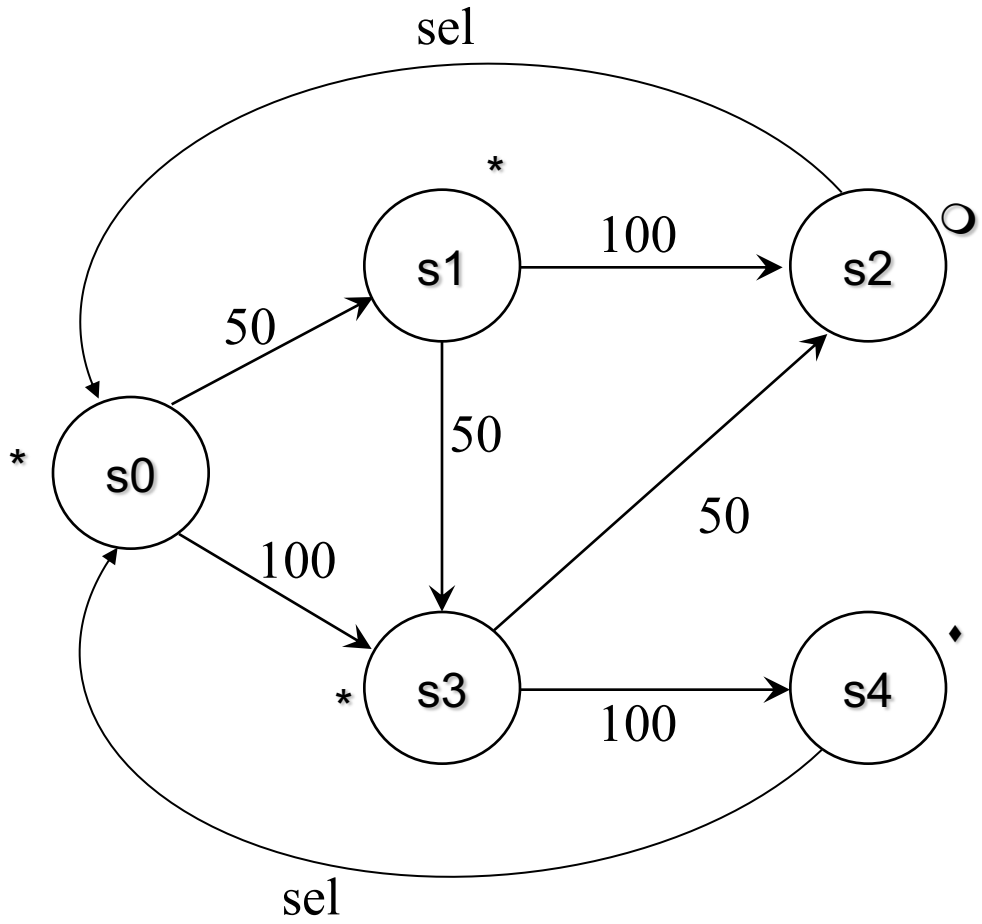
FSMs in VHDL - Coin Handler

- Coin handler

* No enable
No change

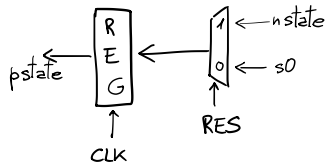
○ Enable
No change

♦ Enable
Change



FSMs in VHDL - Coin Handler

```
PACKAGE CH_pack IS
    type CHstate is (s0, s1, s2, s3, s4)
END CH_pack;

ENTITY CoinHandler is
    PORT(clk, res: in bit;
      Pin: in bit_vector(1 downto 0);
      --01: 50, 10: 100, 11:sel
      Enab, Change: out bit)
END CoinHandler;
```




```
ARCHITECTURE fsm OF CoinHandler IS
    signal pstate: CHstate := s0;
    signal nstate: CHstate := s0;
BEGIN
    StateEvol: process(clk, res)
    BEGIN
    IF (clk'EVENT and clk = '1') THEN
     IF (res = '0') THEN
        pstate <= s0;
     ELSE
        pstate <= nstate;
     END IF;
  END IF;
    END PROCESS StateEvol;
```

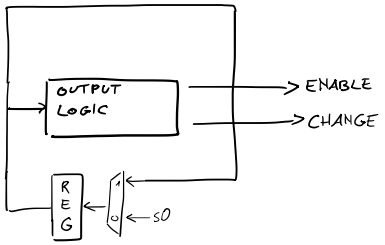
Synchronous reset signal

Reset active low

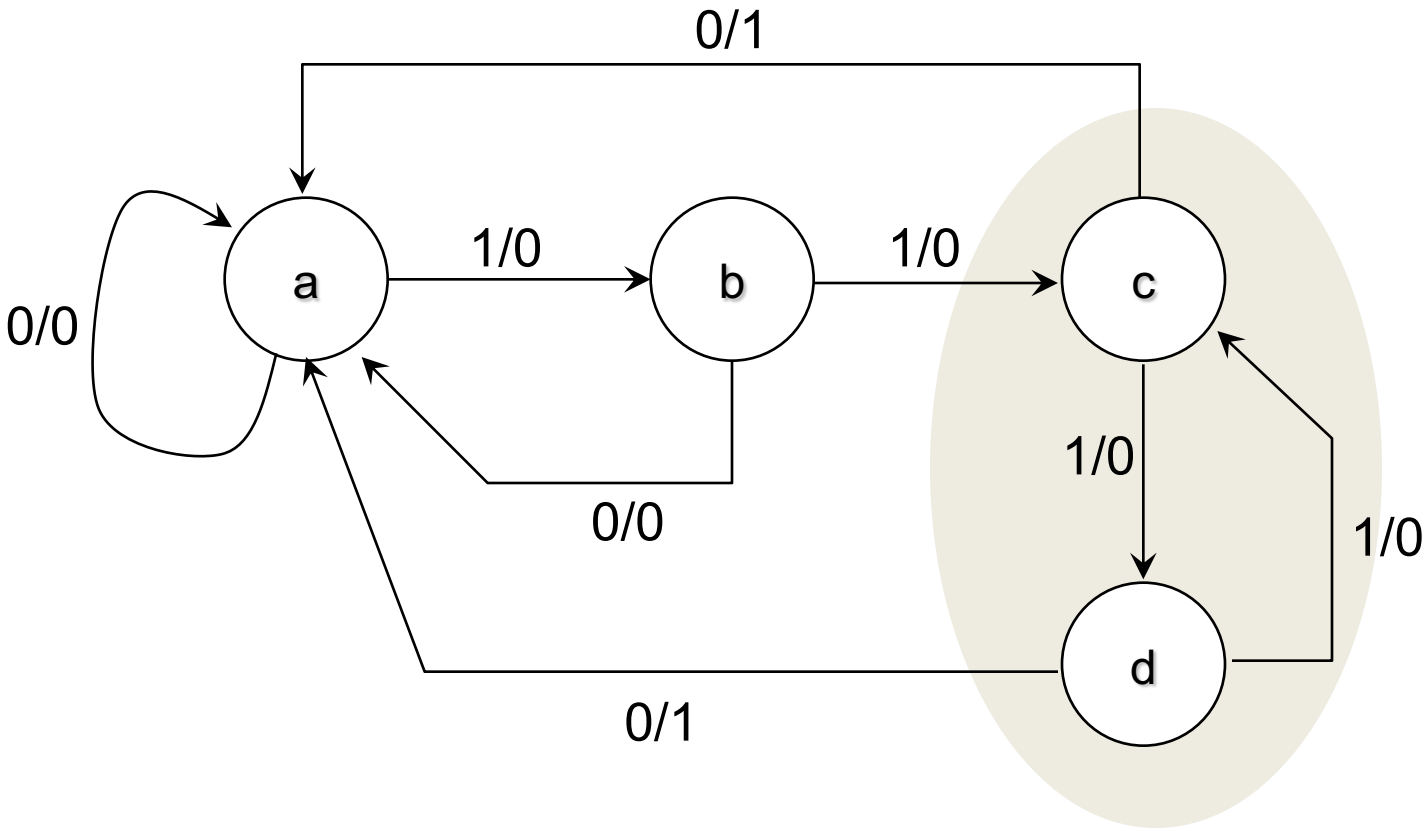
FSMs in VHDL - Coin Handler

```
NSL: process(pstate, Pin)
BEGIN
  CASE pstate IS
  WHEN 's0' => IF Pin = "01" THEN
                  nstate <= s1;
               ELSIF Pin = "10" THEN
                  nstate <= s3;
               ELSE
                  nstate <= pstate;
               END IF;
  WHEN 's1' => IF Pin = "01" THEN
                  nstate <= s2;
               …
  WHEN 's4' => IF Pin = "11" THEN
                  nstate <= s0;
               ELSE
                  nstate <= pstate;
               END IF;
  WHEN OTHERS => nstate <= s0;
  END CASE;
END PROCESS NSL;
```

```
        OL: Process(pstate)
        BEGIN
          CASE pstate IS
               WHEN s0|s1|s3 =>  Enab <= '0';
                                 Change <= '0';
               WHEN s2 =>  Enab <= '1';
                           Change <= '0';
               WHEN s4 =>  Enab <= '1';
                           Change <= '1';
               WHEN OTHERS => Enab <= '0';
                              Change <= '0';
          END CASE;
        END Process OL;
END fsm;
```

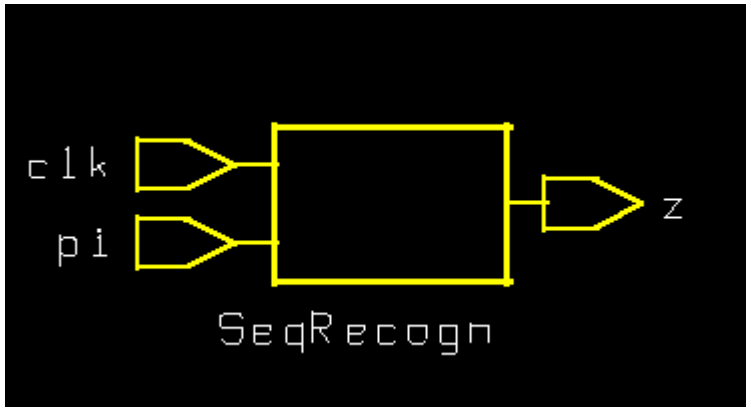


FSMs in VHDL - Recognizer



FSMs in VHDL - Recognizer

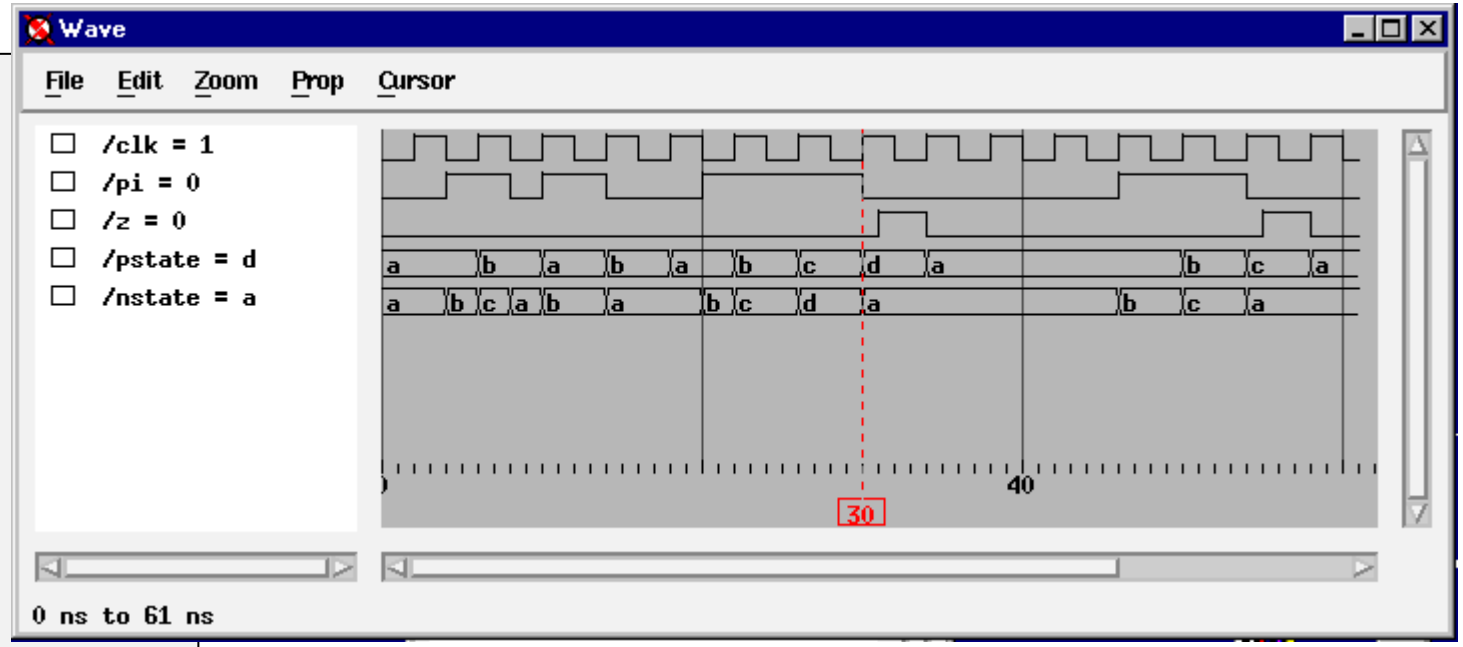
```
PACKAGE RecognPack IS
        type Rstate is (a, b, c, d);
END RecognPack;

ENTITY Recognizer is
        PORT(clk: in bit;
             PI: in bit;
             Z: out bit)
END Recognizer ;
```



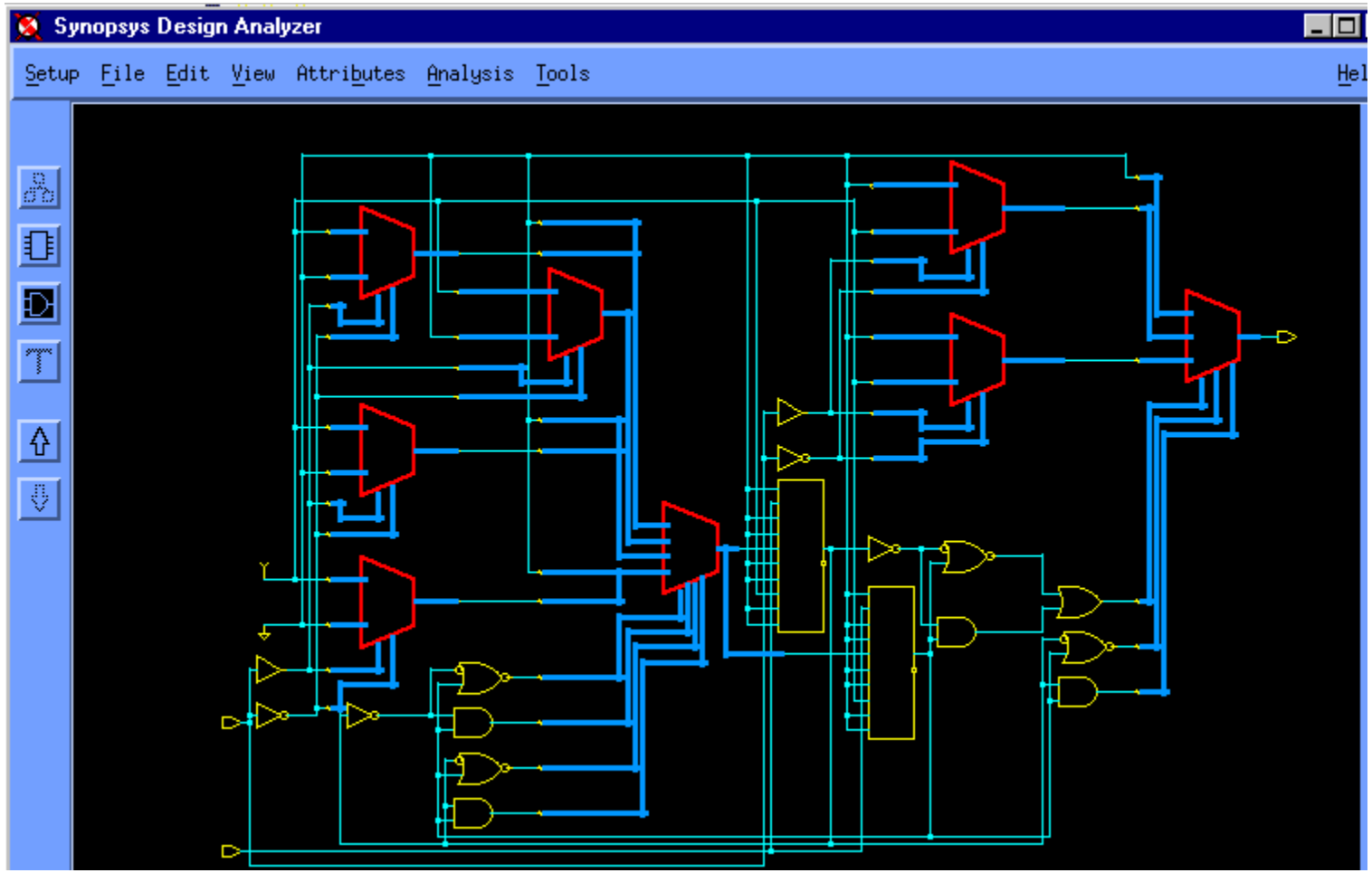

```
ARCHITECTURE ThreeProc OF Recognizer IS
        signal pstate, nstate: Rstate := a;
BEGIN
        NextState: PROCESS(pstate, PI)
        BEGIN
          CASE pstate IS
                WHEN a => IF PI = '1' THEN
                                nstate <= b;
                          ELSE
                                nstate <= a;
                          END IF;
                WHEN b => IF PI = '1' THEN
                                nstate <= c;
                          ELSE
                                nstate <= a;
                          END IF;
                WHEN c => IF PI = '1' THEN
                                nstate <= d;
                          ELSE
                                nstate <= a;
                          END IF;
                WHEN d => IF PI = '1' THEN
                                nstate <= c;
                          ELSE
                                nstate <= a;
                          END IF;
                 WHEN OTHERS => nstate <= a;
           END CASE;
        END PROCESS NextState;
```

FSMs in VHDL - Recognizer

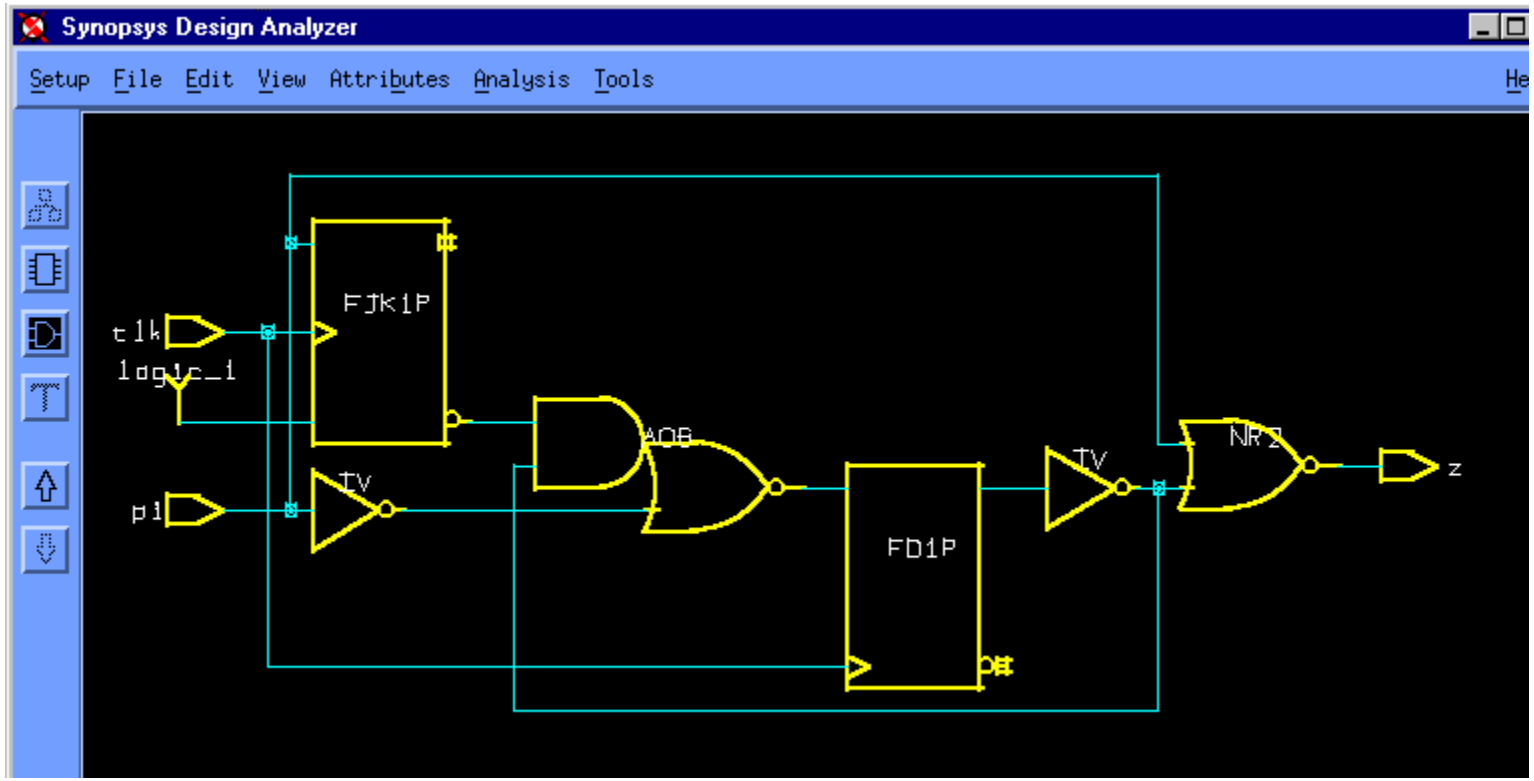
```
Output: PROCESS(pstate, PI)
BEGIN
  CASE pstate IS
 WHEN a => Z <= '0';
 WHEN b => Z <= '0';
 WHEN c => IF PI = '1' THEN
                Z <= '0';
           ELSE
                Z <= '1';
           END IF;
 WHEN d => IF PI = '1' THEN
                Z <= '0';
           ELSE
                Z <= '1';
           END IF;
 WHEN OTHERS => Z <= '0';
  END CASE;
END PROCESS;
 StateEvol: Process
 begin
      WAIT UNTIL clk'EVENT and clk = '1';
      pstate <= nstate;
 end process;
end ThreeProc;
```



FSMs in VHDL - Recognizer



FSMs in VHDL - Recognizer



Sequential elements - summary

- Latch inference
  - No specified assignment for a conditional branch
  - It is not possible to model latches using variables
- Clock inference

```
CLK = '1' AND clk'EVENT
```

  - type: Bit, Boolean, Std_ulogic, Std_logic
  - No ELSE clause in the IF statement checking the clock edge

Signals: wires or memory?

- Latches are inferred when signals are not assigned in all conditional branch.
- Latches are inferred each time an assignment is made before a wait until statement
- Memory (not latches) is inferred on variables anytime it is necessary to store data for subsequent simulation steps.

Synthesis guidelines

- Data Flow statements translate into combinational logic
- Sequential logic is specified in a PROCESS statement that includes a Clock signal
- Explicit State Machines contain one WAIT UNTIL CLOCK'EDGE in a process, with a CASE statement describing the FSM
- Implicit State Machines contain multiple WAIT UNTIL CLOCK'EDGE in a process without a sensitivity list