A VHDL Synthesis Tutorial

First Edition

Valentina Salapura
Michael Gschwind

Technische Universität Wien
Vienna, AUSTRIA

Table of Contents

For bug reports, please contact *valentina@vlsivie.tuwien.ac.at*

Disclaimer: the authors make no warranty of any kind with regard to this material, including, but not limited to, the implied warranties or merchantability and fitness for a particular purpose.

VHDL Primer

Valentina Salapura
Michael Gschwind

About this VHDL primer

- this is not a complete description of VHDL
- concepts have been simplified
- only issues and constructs related to synthesis are explained in this overview
- refer to a VHDL textbook for a full and exact description of the language

VHDL

- VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
- developed by DARPA Very High Speed IC Initiative sponsored by US Department of Defence
- IEEE standard hardware description language
  - IEEE Std. No. 1076 since 1988
- developed for design specification and validation
- required for government defence projects by DoD
- similar to ADA programming language
  - also developed by DoD

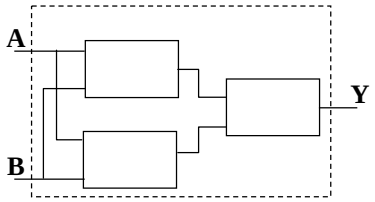
VHDL references

- The VHDL Cookbook, by Peter J. Aschenden
  - available via FTP
    ftp://ftp.vlsivie.tuwien.ac.at/pub/hdl/
    VHDL-Cookbook.tar.Z
- Peter J. Ashenden
  The Designer's Guide to VHDL
  Morgan Kaufmann Publishers, Inc.
- IEEE Standard 1076-1987
  IEEE Standard VHDL Language Reference Manual

VHDL descriptions

- a design can be described in VHDL at different abstraction levels: for the description at the behavioral level, register-transfer level and at the gate level
- appropriate for describing both hardware structure and behavior

structure



behavior

$$Y = AB + \overline{AB}$$

VHDL as a programming language ...

- several constructs in VHDL like in programming languages (not hardware specific)
- comments : two hypens --
- identifiers : letter {[underline] letter number}

```
ThisName = thisname /= This_Name /= ThisName
```

- numbers : integer and real, in bases 2 - 16

```
1234    33E2     0.5       12.5E-9
2#11011          2#110.01  16#abc2
```

- character constants : using single-quote marks `'Z'  '1'`
- string constants : using double-quote marks `"1100"`
- bit strings: B⇒2, O ⇒8, X ⇒16 `B "101010" X "A6"`

Data types

- **data type has to be declared - VHDL strongly typed language**
- **data type specified using type definition**

```
type identifier is type_definition
```

- **basic data types**
  - **scalar**
    - » **integer types**
    - » **floating point types**
    - » **physical types**
    - » **enumeration types**
  - **composite**
    - » **arrays**
    - » **records**

Scalar data types

- **integer: integer values within a specified range**

```
type small is range 0 to 100;
type bit_index is range 31 downto 0;
```

- **floating point: real values within a specified range**

```
type probability is range 0.0 to 1.0;
```

- **physical: for physical quantities (mass, voltage, time...)**

```
25 ns,  10 mm
```

- **enumeration: possible values are listed**

```
type boolean is (true, false)
type weekend is (sa, so)
```

Composite data types - arrays

- **array: indexed set of elements of the same type**
  - **one-dimensional and multidimensional**
  - **constrained and unconstrained**
  - **example: one-dimensional, constrained**

```
type word is array (31 downto 0) of bit;
```

  - **example: multidimensional, constrained**

```
type register_bank is array (0 to 31) of
word;
```

  - **example: one-dimensional, unconstrained**

```
type word is array (positive range <>) of
bit;
```

Composite data types - records

- **record: set of elements of different types**

```
type instruction is record
   op_code: processor_op;
   operand1: integer range 0 to 15;
   operand2: integer range 0 to 15;
end record;
```

Subtypes

- constrained subset of some data type
- subtypes of scalar types and arrays
- examples:

```
type days is (mo, tu, we, th, fr, sa,
  so);
subtype weekend is days range sa to so;

type bin_vector is array (integer range
  <>) of bit;
subtype word is bin_vector(31 downto 0);
```

Objects

- object: named item having value of a specified type
- three classes of objects
  - constants
  - variables
  - signals

```
constant crc_pol: bit_vector(7 downto 0)
  := "00110011";
variable crc_acc: bit_vector(7 downto 0)
  := "11000011";
signal crc_reg: bit_vector(7 downto 0);
```

Operators

- **logic operators**

  `and, nand, or, nor, xor, not`
- **arithmetic operators**

  `+, -, *, /, **, abs, mod`
- **relational operators**

  `=, /=, <, <=, >, >=`
- **concatenation operator**

  `&`

Execution in VHDL

- **statements in VHDL can be executed sequentially and concurrently**
- **sequential statements only contained in processes**
  - **detailed explanation later, see PROCESS statement**
- **several statements can be executed both sequentially and concurrently**
- **statements which can be executed both sequentially and concurrently**
  - **signal assignment**
  - **procedure call statement**

Sequential and concurrent execution

- **statements executed sequentially:**
  - **variable assignment**
  - **IF statement**
  - **CASE statement**
  - **LOOP statement**
  - **NULL statement**
- **statements executed concurrently:**
  - **PROCESS statement**
  - **conditional signal assignment**
  - **selected signal assignment**
  - **component instantiation statement**

Assignments and NULL statement

- **variable assignment**
  - **sequential statement**

```
a := new_value;     -- a is variable
```

- **signal assignment**
  - **can be executed both sequentially and concurrently**

```
b <= new_value;     -- b is signal
```

- **NULL statement**

```
null;
```

IF statement

- **IF statement**

```
if condition_1 then
    sequence_of_statements
elsif condition_2 then
   sequence_of_statements
else
   sequence_of_statements
end if;
```

CASE statement

- **CASE statement**

```
case expression is
   when choice_1 =>
      sequence_of_statements
   when choice_2 =>
      sequence_of_statements
   when others =>
      sequence_of_statements
end case;
```

LOOP statement

- **infinite loop**

```
label: loop
   sequence_of_statements
end loop;
```

- **basic loop statement can be extended to while or for loop**

```
label: while condition | for parameter
  loop
   sequence_of_statements
end loop;
```

- **NEXT and EXIT expressions optional**
  - **NEXT: starts the next loop iteration**
  - **EXIT: exits the loop**

Examples of LOOP statement

- **while loop**

```
Loop_1: while indx < max loop
          indx := indx + 1;
        end loop;
```

- **for loop**

```
Loop_2: for indx in 1 to max loop
          vector_a(indx) <= '0';
        end loop;
```

Subprograms

- two forms
  - procedures
  - functions
- defined in two parts
  - declaration
    - » interface
  - body
    - » implementation
- statements in a subprogram are executed sequentially

Subprogram declaration

- declaration syntax
  - procedure

```
procedure name (formal_parameter_list;)];
```

  - function

```
function name [(formal_parameter_list)]
         return type_result;
```

- declaration example
  - procedure

```
procedure read (L: inout line;
                value: out std_logic);
```

  - function

```
function shl (arg: signed; count: unsigned)
return signed;
```

Subprogram body

- **syntax**

```
subprogram_specification is
    subprogram_declarative_part
begin
    subprogram_statements_part
end name;
```

Example: function body

```
function max (L, R: integer)
        return integer is
begin
   if L > R then
      return L;
   else
      return R;
   end if;
end max;
```

Calling of subprograms

- if calling subprograms, parameters may be passed through
  - parameter position

```
read (act_line; new_value);
```

  - parameter name

```
read (value => new_value;
      L => act_line);
```

- overloading - several subprograms with the same name
  - subprogram is selected using the number and type of parameters

Package

- encapsulated modules for structured programming
- collection of data types, constants, and subprograms
- defined in two parts
  - package declaration
    - » interface description
  - package body
    - » implementation
- package usage

```
use package_name.all;
use std_logic_1164.all;
```

Package declaration

- **syntax**

```
package identifier is
  package_declarative_part
end identifier;
```

- **example**

```
package std_logic_1164 is
type std_logic is ('U', 'X', '0', '1',
                   'Z', 'W', 'L', 'H', '-');
type std_logic_vector is array
          (natural range <>) of std_logic;
function "and" (L, R: std_logic_vector)
          return std_logic_vector;
...
end std_logic_1164;
```

Package body

- **contains subprograms implementations**
- **syntax**

```
package body name is
   package_body_declaration_part
end name;
```

- **example**

```
package body std_logic_1164 is
  function "and" (L, R: std_logic_vector)
          return std_logic_vector is
    begin
      ...
  end "and";
  ...
end std_logic_1164;
```

VHDL description of a function block

- consists of
  - entity
    - » description of the function block interface
    - » contains generics and ports
  - architecture body
    - » implementation of the block functionality
    - » multiple architecture bodies for same interface can be defined
      - different performance
      - different size
      - different abstraction levels
      - different detail
      - for simulation or for synthesis

Entity declaration

- ports
  - input, output, inout
- generics
  - parameterize function blocks
  - select templates parameters
    - » delay (gates, …)
    - » bit width (RAM, ALU, adder, …)
    - » implementation (multiplier, divider, …)
- syntax

```
entity identifier is
   generic (generics_constants_list);
   port (port_list);
end identifier;
```

Example: entity declaration

```
entity processor is
  generic (width: integer);
  port(clock:   in    std_logic;
       address: in    std_logic_vector(31 downto 0);
       data:    inout std_logic_vector(31 downto 0);
       control: in    std_logic_vector(5 downto 0);
       ready:   out   std_logic);
  end processor;
```

Architecture body

- contains the implementation of the block functionality
- several architecture bodies may be specified for each entity
  - different implementations
- architecture bodies implemented at different abstraction levels
  - structural
    - » block diagrams, net lists
  - RTL
    - » operation description using formulas
  - behavioral
    - » procedural description

Architecture body syntax

- **syntax**

```
architecture identifier of entity_name
 is
    architecture_declarative_part
begin
    architecture_statements_part
end identifier;
```

Structural description

- **block functionality described as interconnection of smaller and simpler components**
  - **net list, schematic**
- **hierarchical block structure**
- **contains**
  - **components**
    - » **function blocks, usage of existing design entities**
  - **signals**
    - » **for interconnection of components**
- **advantage: easily mapped on hardware**
- **disadvantage: bad readability and overview**

Components

- for structural descriptions, interconnected with signals
- existing entities used as components
- component described at different abstraction levels
  - different levels of detail
- component used in a structural description has to be
  - declared
    - » in the architecture body declaration part
  - instantiated
    - » in the architecture body statements part

Component declaration

- in architecture body declaration part
- syntax

```
component identifier is
   generic(generic_constants_list);
   port(port_list);
end component;
```

- example

```
component counter
   generic(N: integer);
   port(clock, reset: in std_logic;
    y: out std_logic_vector(0 to N-1));
end component;
```

Component instantiation

- **in architecture body statements part**
- **specify connections and parameters for generic parameterizable modules**
- **syntax**

```
label: name generic map(generic_list);
           port map(port_list);
```

- **example**

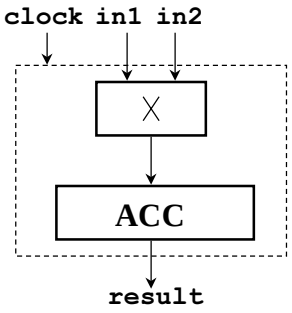
```
u3: counter generic map (4);
            port map(clk, n245, n254);
```

parameter for parameterizable blocks

connections

Structural description - example

- **multiply-and-accumulate function block**
- **block diagram**



MACC Entity

```
entity macc is
   port(
    in1:    in  std_logic_vector(7 downto 0);
    in2:    in  std_logic_vector(7 downto 0);
    clock:  in  std_logic;
    result: out std_logic_vector(15 downto 0));
end macc;
```

Architecture body - structural description

```
architecture structure of macc is
signal mul_res: std_logic_vector(15 downto 0);
component mult
     port(a,b: in  std_logic_vector(7 downto 0);
          y:   out std_logic_vector(15 downto 0));
end component;
component acc
     port(a:   in  std_logic_vector(15 downto 0);
          clk: in  std_logic;
          y:   out std_logic_vector(15 downto 0));
end component;
begin
multiply: mult port map(in1, in2, mul_res);
accum:    acc  port map(mul_res, clock, result);
end structure;
```

RTL description

- register transfer level
- description at higher abstraction level
- operations described with formulae
- usage of signals
  - for value exchange between formulae
- functionality described as calculation of new signal value from other signals

```
yn = f(yk, yl,...)
```

- advantage: good readability
- disadvantage: complicated mapping on hardware

Simplified VHDL simulation model

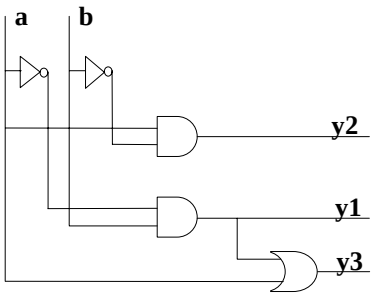
- execution of a VHDL description
  - statements in an endless loop
  - all statements executed repeatedly
  - signal change visible only in the next simulation loop iteration
    - » signals introduce one unit delay
    - » abstract model of hardware delay

Consequences of signal delay

- in architecture body all statements executed concurrently
  - corresponds to hardware where all components and wires concurrently active
- order of concurrent statements not important
- exact delay of signals possible using “after” clause

Execution of concurrent statements

- exercise:
  - write an RTL description for the following block diagram



Solution

```
entity simple is
    port(a,b:      in  std_logic;
         y1,y2,y3: out std_logic);
end simple;

architecture arch of simple is
begin
    y2 <= a and not(b);
    y1 <= not(a) and b;
    y3 <= a or (not(a) and b);
end arch;
```

AFTER clause

- **defines exact delay**
- **syntax**

```
target <= value_expression after
                  time_expression;
```

- **example**

```
out2 <= not(in2) after 5 ns;
in2  <= out1 after 1 ns;
```

Concurrent statements

- **signal assignment**
- **conditional signal assignment**
  - **corresponds to sequential IF**
- **selected signal assignment**
  - **corresponds to sequential CASE**

Conditional signal assignments

- **concurrent IF**
- **syntax**

```
target <= {waveform when condition else}
          waveform;
```

- **example**

```
reset <= '0', '1' after 10 ns when
                      short = '1' else
         '0', '1' after 25 ns;
```

Selected signal assignment

- **concurrent CASE**
- **syntax**

```
with expression select
     target <= {waveform when choice,}
                waveform when choice;
```

- **example**

```
with code select
     result <= a + b when addition,
               a - b when subtract;
```

MACC - RTL Description

```
architecture rtl of macc is
signal mul_res,reg: std_logic_vector(15
  downto 0);
    begin
    reg <= reg + in1 * in2 when
            clock = '1' and clock'event;
    result <= reg;
end rtl;
```

rising clock edge

- **VHDL defines attributes for various object types**
  - **for signals: `'event` - signal value has changed**

Behavioral description

- descriptions at high abstraction level
- behavioral VHDL similar to “normal” programming languages
- encapsulated in VHDL processes
- advantages
  - simple and quick description
  - high simulation speed
- disadvantages
  - complicated mapping on hardware
  - less detailed description

Process statement

- concurrent statement in architecture body
  - executed concurrently with other statements
- contains sequential statements
  - encapsulated statements executed sequentially
- variables used in a process
  - variables change value immediately, like in “normal” programming languages
- signals can still be used
  - semantics differ from variable $\Rightarrow$ changes only visible in next iteration of process
  - useful for communication with other concurrent elements in architecture body

Process statement: syntax

```
process [(sensitivity_list)]
   process_declarative_part
begin
   process_statements_part
end process;
```

Process statement: example

```
P22: process (clock, reset)
   begin
     if reset = '1' then
        reg <= "0000";
     elsif clock = '1' and clock'event then
        reg <= "1100";
     end if;
end process;
```

Building a VHDL description

- in a single VHDL design used many different packages, architecture bodies and entities
- active components (architecture bodies, packages) for a particular design selected using
  - libraries
  - configurations

Library

- encapsulates various related components
- identified with a logic name
- design descriptions compiled in libraries for simulation and synthesis
- order of compilation important
  - primary library units
    - » have to be compiled first
    - » package declarations, entities, configuration declaration
  - secondary library units
    - » package bodies, architecture bodies

Using libraries

- referenced with “use” clause
- example

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

- library “work” always selected
  - working library
  - need not to be referenced

Configuration

- configuration selects
  - architecture body for entities
  - entities for components
  - generics for components
- selects architecture body for each instance, e.g.
  - architecture body with optimized timing for timing critical parts
  - architecture body with minimal area for area critical parts
  - behavioral description for fast simulation

Configuration syntax

```
configuration identifier of entity_name
 is
   use_clause
   for architecture_name
      use_clause
      for block_name ...
      end for;
   end for;
end identifier;
```

- **first hierarchical "for" clause specifies architecture body**

Configuration: example

```
architecture simulate of adder is
   component hadd port(…); end component;
begin
  B1: hadd port map(…);
  B2: hadd port map(…);
end simulate;

configuration mixed of adder is
  for simulate
    for B1:
      hadd use entity work.hadd(rtl); end for;
    for B2:
      hadd use entity work.hadd(synth); end for;
  end for;
end mixed;
```

Standard logic

- standardized data types for portably describing signal values in VHDL
- IEEE standard multivalue logic system for VHDL model interoperability (Std_logic_1164)
  - IEEE Std. No. 1164-1993
  - defined in `IEEE.std_logic_1164` package
- should be supported by all simulators and synthesis tools
- simple mapping on hardware
- arithmetic operations on `std_logic` standardized in IEEE 1076.3 synthesis package Numeric_Std

Standard logic data types

- data type `std_logic`:
  - `'U'` - not initialized
  - `'X'` - unknown, strong
  - `'0'` - 0, strong
  - `'1'` - 1, strong
  - `'Z'` - high impedance (tri-state)
  - `'W'` - unknown, weak
  - `'L'` - 0, weak
  - `'H'` - 1, weak
  - `'-'` - don´t care

Other standard data types

- `std_logic_vector`
- `std_ulogic`, `std_ulogic_vector`
  - unresolved `std_logic`
    - » several drivers drive single bus / wire
- `signed`, `unsigned`
  - defined in IEEE 1076.3 Numeric_Std
  - array of `std_logic` with integer semantics

Data type conversion functions

- VHDL strongly typed - but not hardware
  - synthesis tools have to provide trivial mapping of functions
- in synthesis package `IEEE.std_logic_arith`
- most important conversion functions
  - `conv_integer`
  - `conv_unsigned`
  - `conv_signed`
  - `conv_std_logic_vector`

Some conversion functions

- **conv_integer**
  - input is **integer** | **signed** | **unsigned** | **std_logic**
  - return **integer**
- **conv_unsigned, conv_signed**
  - input is **integer** | **signed** | **unsigned** | **std_logic**
  - return **unsigned, signed**
- **conv_std_logic_vector**
  - input is **integer** | **unsigned** | **signed** | **std_logic**
  - parameter **integer** (for width)
  - return **std_logic_vector**

VHDL Simulation

Valentina Salapura
Michael Gschwind

Introductory note

- **this collection only gives short overview of simulation**
- **concepts have been simplified**
- **intended to give basic understanding of VHDL simulation**
- **for full information on VHDL simulation check VHDL simulation text book**

Simulation

- for validating VHDL Models
- simulation shows
  - whether errors are found for given test vectors
  - not that the design is correct
- quality of test vectors is critical for quality of simulation

Simulation development

- 1980 - CAE
  - simulation at the gate level
    - » long simulation time
- 1990 - High level approach
  - simulation at behavioral and RTL levels
    - » usage of test benches
    - » procedural description of test vectors
    - » simple and quick design at higher abstraction level
    - » faster simulation because less detail required

Usage of simulation

- VHDL system simulation
  - behavioral and RTL description
- post-synthesis simulation and timing analysis
  - at the gate level
- sign-off simulation
  - after layout and routing
    - » exact timing analysis

Simulation model

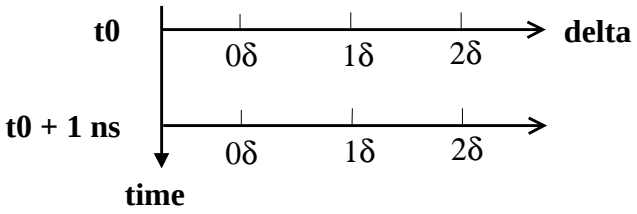
- discrete event simulation
- simulation controlled by events ⇒ event driven simulation
  - event – the value of a signal had changed
  - change of a signal is not immediately executed
  - assignment of new signal values scheduled along a time axis

Simulation algorithm

- defined by IEEE VHDL standard 1076
- simulation starts with the initialization phase
  - simulation time = 0
  - all objects initialized
- iterate 2 stage simulation cycle
  - propagation
    - » all transactions (signal changes) scheduled for this simulation time executed
  - evaluation
    - » execute all VHDL modules which react to events occurring in the first stage
    - » schedule new signal changes

Delta method

- two dimensional time

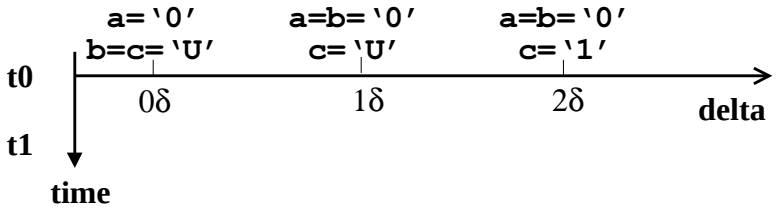


- guarantees
  - compatibility between different simulators
  - determinism

Signal assignment

- **without “after” clause**
- **input** `a` **is** `‘0’` **at t0**

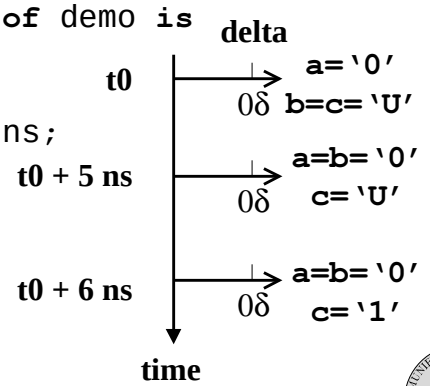
```
architecture simple of demo is
 begin
   b <= a;
   c <= not(b);
end simple;
```



Signal assignment (2)

- **with “after” clause**
- **input** `a` **is** `‘0’` **at t0**

```
architecture simple of demo is
begin
b <= a after 5 ns;
c <= not(b) after 1 ns;
end simple;
```



Controlling simulation

- evaluate
  - only the modules affected by changed signals
    - » activation list for concurrent statements
    - » sensitivity list for processes
- propagate
  - only the signals whose value should change

VHDL for simulation

- statements for describing time flow in the simulation
  - after
    - » for time flow of signal assignments

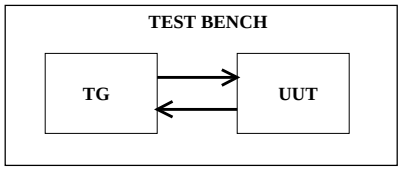
```
a <= "11110000" after 100 ns;
```

  - wait
    - » to start and stop a process

```
wait for 10 ns;
wait until reset = '1';
wait on clock'event;
```

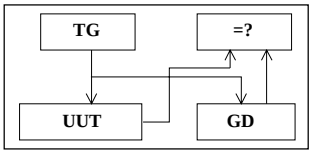
Test bench

- defines the stimuli of a design
- consists of two components
  - UUT (unit under test)
    - » tested design
  - TG (test generator)
    - » generates stimulus for the design
    - » collects the design's response, compares with reference values



Types of test bench

- system test bench
  - describes the design environment
  - designer know how
- comparison test bench
  - compares the implementation with a reference model
    - » reference model = expected simulation results at higher abstraction levels or golden device
    - » difficulties with timing differences between the implementation and the reference model



Building a test bench

```
entity TB is
end TB;

architecture TBA of TB is
  declarative_part_signals_components
begin
  instantiate_UUT_and_other_components
  and_optional
  behavioral_description_of_TG
end TBA;
```

Putting together a simulation model

- **arranging a configuration**
  - **connecting objects**
    - » **component <=> entity**
    - » **entity <=> architecture**
    - » **generics**
- **example**

```
configuration cfg_tb of TB is
  for TBA:
    for uut: u_entity
      use entity work.u_entity(u_arch);
    end for;
  end for;
end cfg_tb;
```

Simulation of VHDL models

- VHDL model has to be
  - compiled (analyzed)
  - simulated
- simulation using Synopsys
  - interpreted
    - » intermediate format (byte code)
    - » similar to VHDL source code
    - » easy debugging
    - » slow simulation
  - compiled
    - » translated to machine code
    - » fast simulation

Working environment for Synopsys

- setup file in the working directory
  `.synopsys_vss.setup`
  - contains data path for system libraries
- library `work`
  - contains analyzed elements
- typical working directory - listing

```
ls
 .synopsys_vss.setup   # Synopsys configuration
 vhdl_file.vhd         # VHDL source file
 work/                 # contains analyzed source
```

- Synopsys online documentation

  `iview`

Compilation with Synopsys

- **compilation command (for library `work`)**

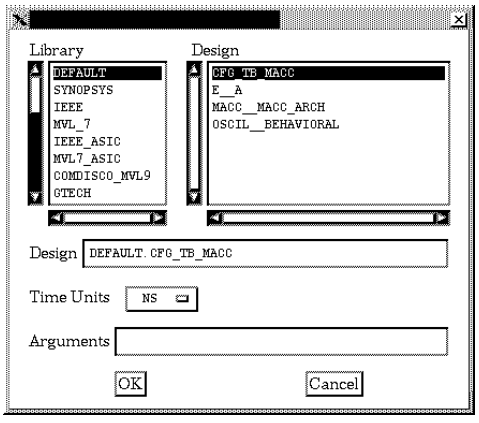
  `vhdlan` *`vhdl_file`*`.vhd`
- **compilation to other libraries**

  `vhdlan` *`vhdl_file`*`.vhd -w` *`library_name`*

  – **library name has to be specified in `.synopsys_vss.setup`**

Simulation with Synopsys

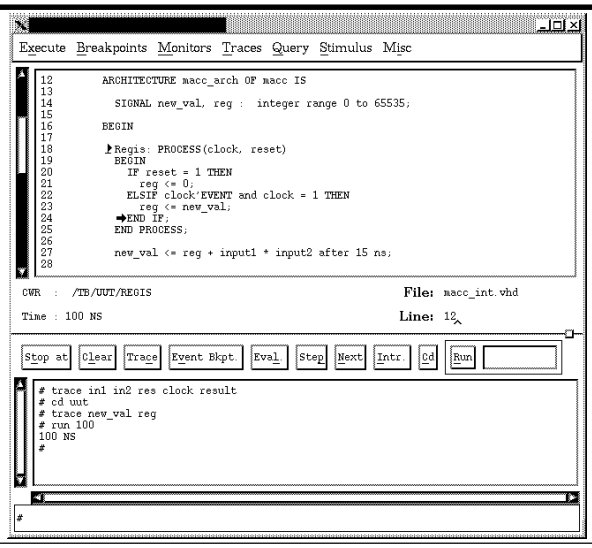
- **Synopsys VHDL debugger - `vhdldbx`**
- **select design**



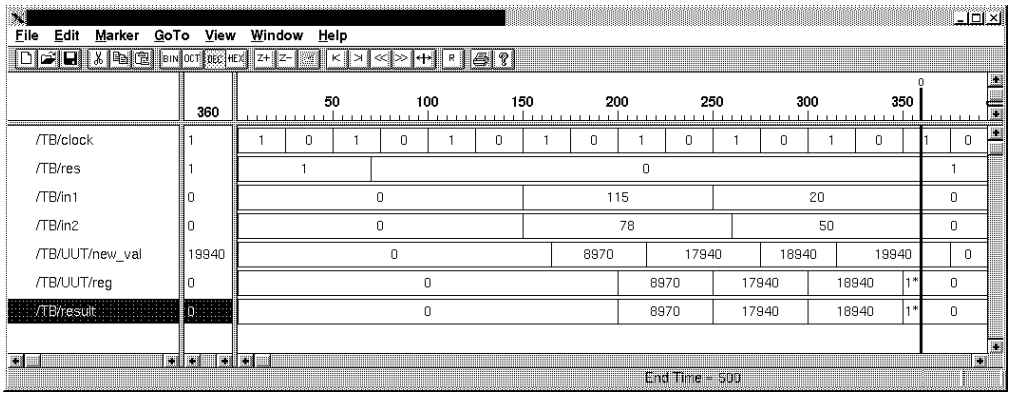
Useful commands

- **`trace`**
  - **signal selection for waveforms**
- **`cd`**
  - **moving through the program hierarchy**
- **`step, next`**
  - **execute simulation in steps**
- **`run` *`time_expression`***
  - **simulation start**
- **`include`**
  - **usage of scripts**

Vhdldbx - working window



Vhdldbx - waveforms



Simulation example: MACC entity

```
use work.all;
ENTITY macc IS
   PORT(
        input1:  IN   integer range 0 to 255;
        input2:  IN   integer range 0 to 255;
        reset:   IN   integer range 0 to 1;
        clock:   IN   integer range 0 to 1;
        out_reg: OUT  integer range 0 to 65535
       );
END macc;
```

Simulation example: MACC architecture

```
ARCHITECTURE macc_arch OF macc IS
  SIGNAL new_val, reg:  integer range 0 to 65535;
BEGIN
Regist: PROCESS(clock, reset)
    BEGIN
      IF reset = 1 THEN
         reg <= 0;
      ELSIF clock'EVENT and clock = 1 THEN
         reg <= new_val;
      END IF;
  END PROCESS;
  new_val <= reg + input1 * input2 after 15 ns;
  out_reg <= reg;
END macc_arch;
```

Simulation example: Oscillator for clock signal

```
ENTITY oscil IS
      PORT(txc: OUT  integer range 0 to 1:=0);
END oscil;
ARCHITECTURE behavioral OF oscil IS
BEGIN
  PROCESS
    BEGIN
      LOOP
         txc <= 1;
         wait for 25 ns;
         txc <= 0;
         wait for 25 ns;
      END LOOP;
  END PROCESS;
END behavioral;
```

Simulation example: MACC test bench

```
entity TB is
end TB;

Architecture Arch of TB is
  SIGNAL in1, in2: integer range 0 to 255;
  SIGNAL res, clock: integer range 0 to 1;
  SIGNAL result: integer range 0 to 65535;

  COMPONENT macc
     PORT (input1, input2: IN integer range 0 to 255;
           reset, clock:   IN integer range 0 to 1;
    =      out_reg:  OUT integer range 0 to 65535);
  END COMPONENT;

  COMPONENT oscil
     Port (txc: OUT integer range 0 to 1);
  END COMPONENT;
  %
```

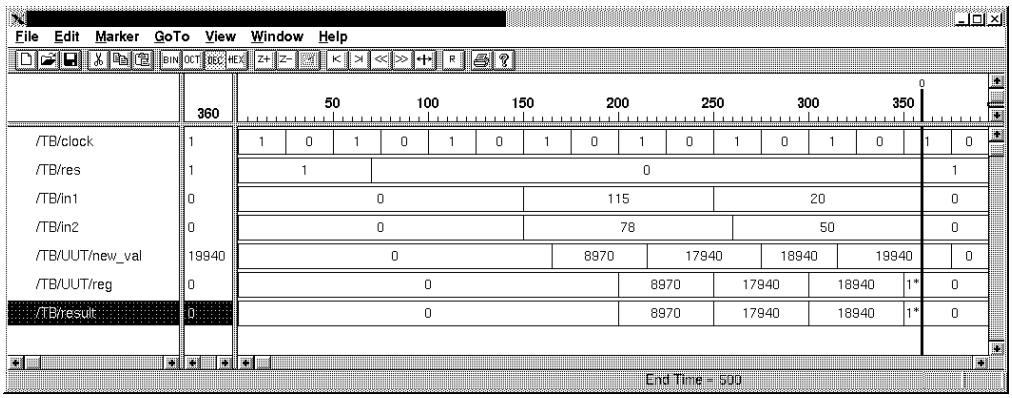
Simulation example: MACC test bench cont'd

```
BEGIN

UUT : macc PORT MAP (in1, in2, res, clock, result);
clk : oscil PORT MAP (clock);

TBP : PROCESS
  BEGIN
  in1 <= 0; in2 <= 0; res<=1;
  wait for 70 ns;              -- reset cycle
  res<=0; wait for 80 ns;
  in1 <= 115; in2 <= 78;       -- input data
  wait for 100 ns;
  in1 <= 20;=                  -- new input data
  wait for 10 ns;
  in2 <= 50;
  wait for 100 ns;
END PROCESS;
end Arch;
```

Simulation example: MACC configuration

```
configuration cfg_tb_macc of TB is
  for Arch:
    for uut: macc
      use entity work.macc(macc_arch);
    end for;
  end for;
end cfg_tb_macc;
```

Simulation example: Tracing MACC operation using waveforms



Exercise 1
VHDL design and simulation

Valentina Salapura
Michael Gschwind

ALU design and simulation

- describe an ALU with following properties in VHDL
  - 2 data inputs, 1 output
    - » data type integer range 0 to 256
  - control input for operation selection
    - » choose appropriate encoding for function selection: enumeration data type, integer, bitstring…
  - ALU operations — delay
    - » `pass A, pass B` — 2 ns
    - » `not A, A and B, A or B, A xor B` — 5 ns
    - » `A + B, A - B` — 16 ns
    - » `A * B` — 50 ns

Complete the following tasks

- describe the ALU in VHDL
- develop a test bench for simulation
- verify ALU functionality using functional simulation
- explain your design decisions and their impact

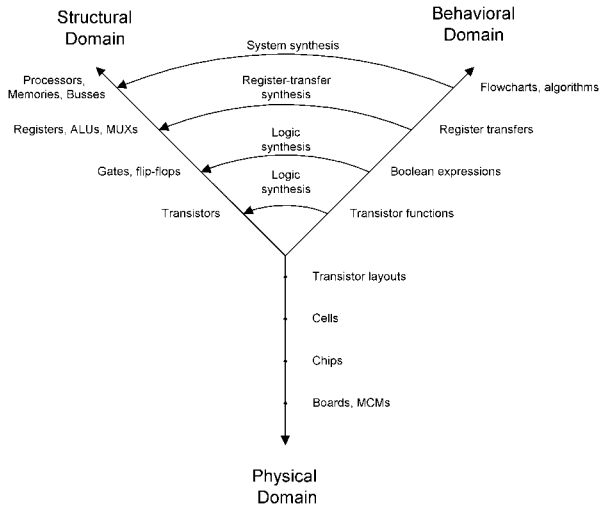
VHDL Synthesis Primer

Valentina Salapura
Michael Gschwind

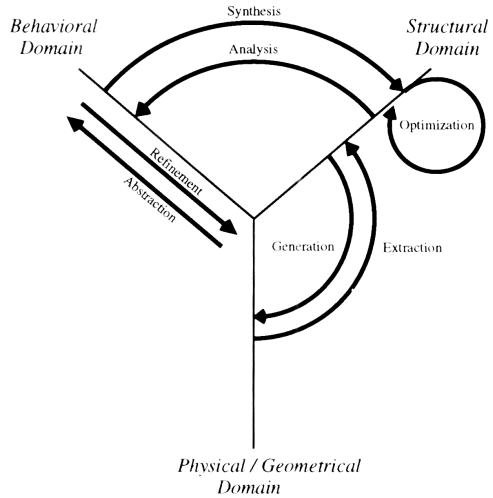
Synthesis

- **automatic translation from one abstraction level to a lower abstraction level**
  - **generates more detailed descriptions**
- **hardware structure directly inferred from an HDL description**
  - **hardware not specified explicitly**
  - **⇒ coding style important for efficiency**

Gajski's Y-Diagram



Design steps in Y-Diagram



Synthesis at different abstraction levels

- system level synthesis
- high-level synthesis
- RTL synthesis
- logic-level synthesis
- technology mapping

Theory …
synthesis at high abstraction levels

- system level synthesis
  - partitioning of a system into subsystems and definition of its algorithms
- behavioral synthesis (high-level synthesis)
  - translation from a behavioral into an RTL description
  - explore different architectures
  - design divided in data and control path (resource allocation, scheduling, resource assignment)

Theory …
RTL and logic synthesis

- RTL synthesis
  - performs control and data path synthesis
  - resource allocation and sharing
    - » maps RTL description to Boolean functions
- logic-level synthesis
  - design described with combinational logic (Boolean functions) and sequential elements
  - description mapped to the gate level and optimized

Theory …
Technology mapping

- technology mapping
  - abstract gates mapped to technology elements
  - function blocks sometimes mapped to macros

… and reality

- system synthesis does not exist
- behavioral synthesis - first tools available
  - for example, Behavioral Compiler from Synopsys
- RTL + logic synthesis available today
  - in good tools no difference between these steps
  - for example, Design Compiler from Synopsys
- pure logic synthesis
  - available for a long time, e.g. Espresso
  - optimization of Boolean equations

Advantages of synthesis

- shorter design time
- simplifies modularity and simple integration of design
- management of design complexity
  - 10M transistor designs already exist, and growing!
- facilitates optimization of the whole design
  - removal of redundant functions
- enables simple porting of design to other technologies by re-synthesizing it

Intellectual property (IP) theory

- how to make 100 Mgate designs
  - buy already existing design blocks from IP vendors
    - » x86 VHDL model
    - » chip set model
    - » SCSI controller model
    - » Ethernet controller model
    - ⇒equals PC on a chip
  - synthesize all blocks in a single chip
- today's integration of components on printed circuits board ⇒ tomorrow's integration of hardware descriptions on a chip

Disadvantages of synthesis

- synthesized hardware depends on coding style
- synthesis efficiency depends on
  - synthesis tools used
  - target technology
  - ⇒ less portable than ideally possible
- less control over the resulting hardware
- hand design of particular blocks may be better than the synthesis result
  - but hand design probably depends on technology

HDL - hardware description languages

- synthesis based on HDL
- most common HDLs: VHDL and Verilog
- developed for simulation, not for synthesis
  - synthesis semantics sometimes different from simulation semantics
  - what you synthesize is not always what you simulate
- synthesis basics similar for all HDLs

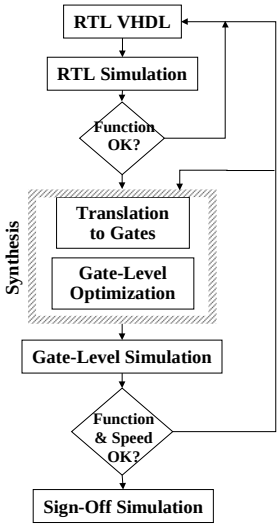
Verilog

- mostly used in the US
- developed by Gateway Design Automation software company 1983
- similar to “C” programming language
- data types defined by language, not by user

VHDL

- mostly used in Europe
- developed by DoD – required for defense contracts
- strong typing system
- similar to “ADA” – by design
  - also developed by DoD
  - usage of data types and libraries
  - similar syntax

Design flow



Synthesis tools

- RTL and logic synthesis widely used for approximately 5 years
- up to date hardware design method
- synthesis tools well developed, but quality of synthesis results varies
  - Synopsys, Compass, Powerview, Cadence, ...
- some synthesis tools ignore configuration specification and/or user-defined libraries
- some synthesis tools do not support resource-sharing

Mapping HDL to hardware

- in an HDL description, hardware is not explicitly defined
- structure of hardware inferred from the structure of a HDL description

  ⇒ the quality of synthesized hardware depends on coding style of an HDL description

  – possibly also depends on synthesis tools and target technology

HDL translation for synthesis

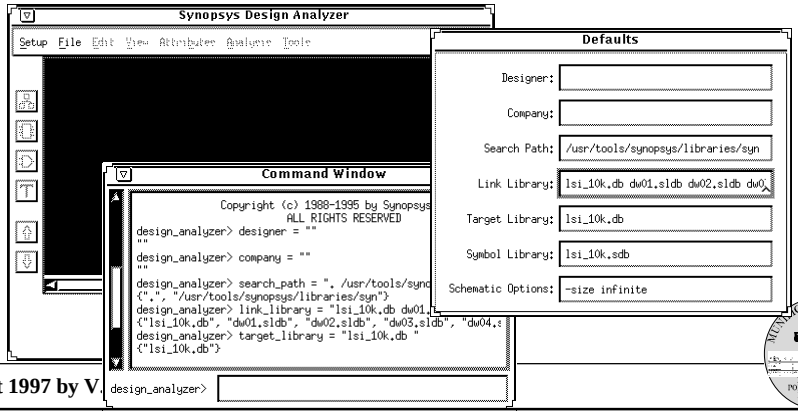
- translation in two steps (as performed by Synopsys Design Compiler)
  - module generator - modules inferred from an HDL description
    - » modules for simple operations
    - » technology independent
  - module mapping to gates
    - » technology specific
    - » makes gate-level design optimization possible

Synthesis environment

- VHDL based synthesis
- synthesis tools `design_analyzer` from Synopsys
  - probably the best synthesis tool currently available
- target technology (for our experiments) is the ASIC technology LSI 10K
  - available in the Synopsys standard distribution
  - 1.5µ, double metal, single poly
    - » approximately 10 years old
  - technology library specified in file `.synopsys_dc.setup`

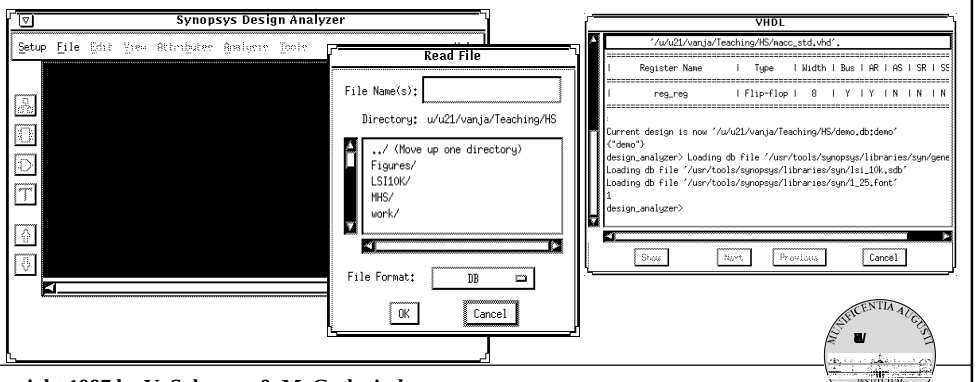
Synthesis using Synopsys

- invoke `design_analyzer`
- `setup` ⇒ `defaults` – selected technology libraries
  ⇒ `command window` – command line input



Synthesis using Synopsys

- synthesis executed in two steps
  1. step: `file` ⇒ `read` – read in VHDL files
     - » macro for commands `analyze` and `elaborate`



Mapping VHDL constructs

- synthesis tools map VHDL constructs to three hardware elements
  - combinational logic
  - multiplexors
  - memory elements
- what hardware is generated from
  - arithmetic, logic and comparison operators
  - IF and CASE statements
  - indexes and arrays
  - process statement

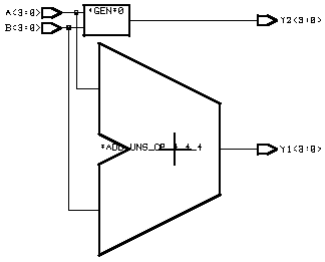
Arithmetic and logic operations

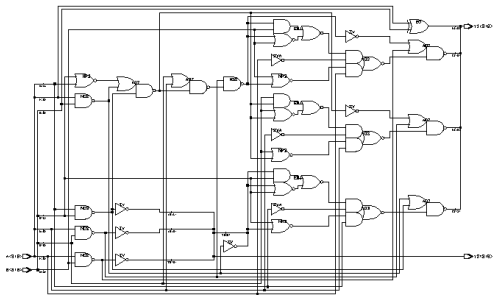
- arithmetic and logic operation are mapped to combinational logic

```
LIBRARY IEEE, work;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.all;
ENTITY adder IS
 PORT(A,B:   IN  std_logic_vector(3 downto 0);
      Y1,Y2: OUT std_logic_vector(3 downto 0));
END adder;
ARCHITECTURE arch OF adder IS
BEGIN
  Y1 <= A + B;
  Y2 <= A AND B;
END arch;
```

Result of synthesis

- **step 1: modules for simple operators**
- **step 2: `tools` ⇒ `design optimization` – modules mapped to gate level and optimized**

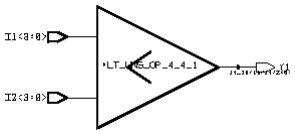


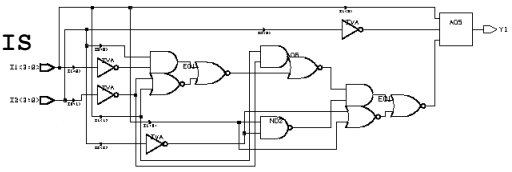


Comparison operators

- **comparison operators are mapped to combinational logic**

```
ENTITY compare IS
PORT(I1,I2: IN std_logic_vector
             (3 downto 0);
     Y1: OUT boolean);
END compare;

ARCHITECTURE arch OF compare IS
BEGIN
   Y1 <= ( I1 < I2 );
END arch;
```

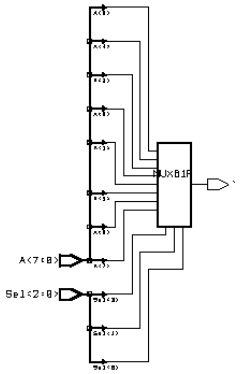




Array indexes

- **from a variable index - multiplexor**

```
ENTITY indexing IS
   PORT(A: IN std_logic_vector
              (7 downto 0);
        Sel: IN integer range 0 to 7;
        Y:   OUT std_logic);
END indexing;

ARCHITECTURE arch OF indexing IS
BEGIN
  Y <= A(Sel);
END arch;
```



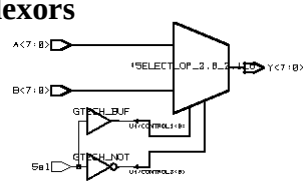
- **from a constant index - wire**

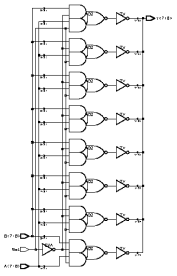
```
Y <= A(3);
```

IF statement

- **IF statements are mapped to multiplexors**

```
ENTITY muxing IS
   PORT(A,B: IN std_logic_vector
             (7 downto 0);
        Sel: IN std_logic;
        Y: OUT std_logic_vector
             (7 downto 0));
END muxing;
ARCHITECTURE arch OF muxing IS
BEGIN
  PROCESS (A, B, Sel) BEGIN
    IF Sel = '1' THEN
      Y <= A;
    ELSE
      Y <= B;
    END IF;
  END PROCESS; END arch;
```





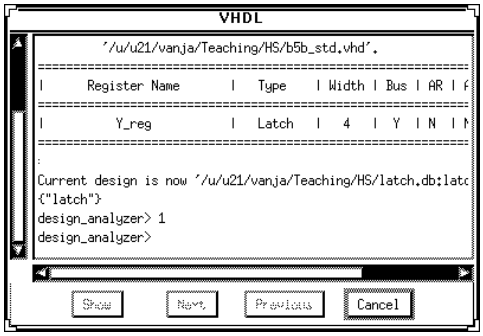
IF statement for memory elements

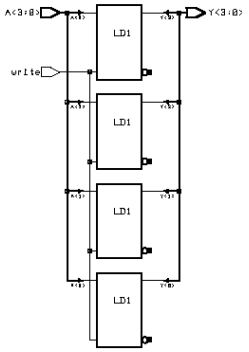
- **if not all states of an IF statement are defined, a memory element is inserted**

```
ENTITY latch IS
  PORT(A: IN  std_logic_vector(3 downto 0);
       write: IN std_logic;
       Y: OUT std_logic_vector(3 downto 0));
END latch;
ARCHITECTURE arch OF latch IS
BEGIN
  PROCESS (A, write) BEGIN
    IF write = '1' THEN
      Y <= A;
    END IF;
  END PROCESS; END arch;
```


Latch

- **important: all values have to be defined to avoid memory elements (latches) in combinational logic**

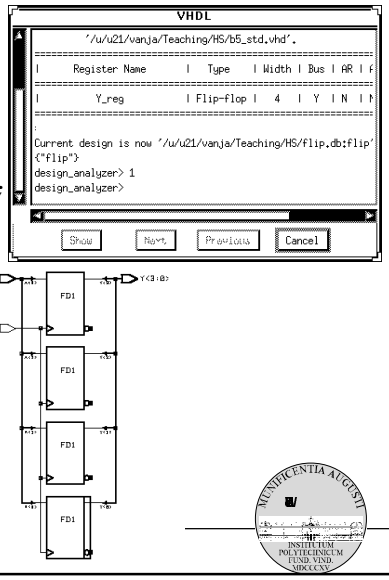




Flip-flop

- **controlled with clock edge**

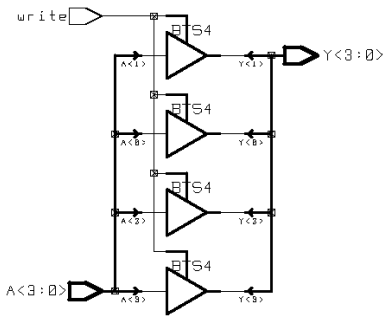
```
ENTITY flip IS
PORT(A: IN std_logic_vector(3 downto 0);
     clock: IN std_logic;
     Y: OUT std_logic_vector(3 downto 0));
END flip;

ARCHITECTURE arch OF flip IS
BEGIN
  PROCESS (A, clock) BEGIN
    IF clock = '1' AND clock'EVENT THEN
      Y <= A;
    END IF;
  END PROCESS; END arch;
```



IF statements for tri-state buffers

- **all values defined, "Z" used for "others/ELSE"**

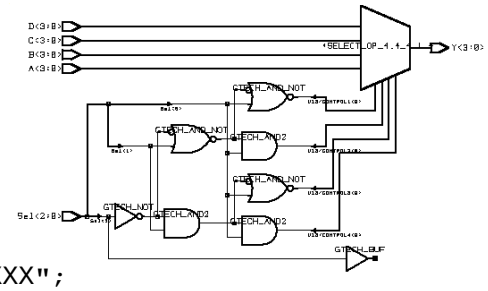
```
ARCHITECTURE arch OF tri IS
BEGIN
  PROCESS (A, write)
  BEGIN
    IF write = '1' THEN
      Y <= A;
    ELSE
      Y <= "ZZZZ";
    END IF;
  END PROCESS;
END arch;
```



CASE statement and don´t care

- **CASE statements are mapped to multiplexors**
- **don´t care for better optimization**
  - **"X" in std_logic for synthesis**

```
ARCHITECTURE arch OF casing IS
BEGIN
  PROCESS (A, B, C, D, Sel)
  BEGIN
    CASE Sel IS
      WHEN 0 => Y <= A;
      WHEN 1 => Y <= B;
      WHEN 2 => Y <= C;
      WHEN 3 => Y <= D;
      WHEN Others => Y <= "XXXX";
    END CASE;
  END PROCESS; END arch;
```



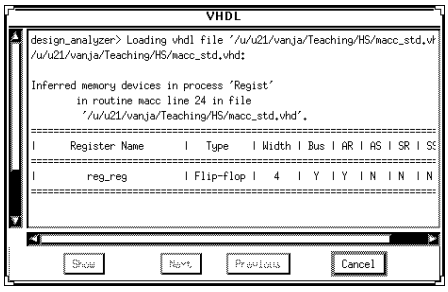
Example design - MACC

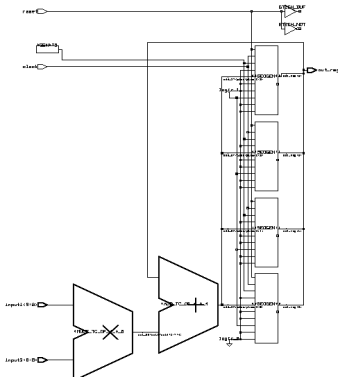
```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
ENTITY macc IS
  PORT(input1, input2: IN std_logic_vector(3 downto 0);
       reset, clock: IN std_logic;
       out_reg: OUT std_logic_vector(3 downto 0));
END macc;
ARCHITECTURE demo_arch OF macc IS
   SIGNAL new_val, reg: std_logic_vector(3 downto 0);
BEGIN
Regist: PROCESS(clock, reset) BEGIN
    IF reset = '1' THEN
      reg <= "0000");
    ELSIF clock'EVENT and clock = '1' THEN
      reg <= new_val;
    END IF; END PROCESS;
Combin: PROCESS(input1, input2, reg)
  VARIABLE new_val1: std_logic_vector(7 downto 0); BEGIN
    new_val1 := input1 * input2;
    new_val <= reg + new_val1(7 downto 4);
  END PROCESS;
  out_reg <= reg; END demo_arch;
```

Example design - MACC

- **read the VHDL description**

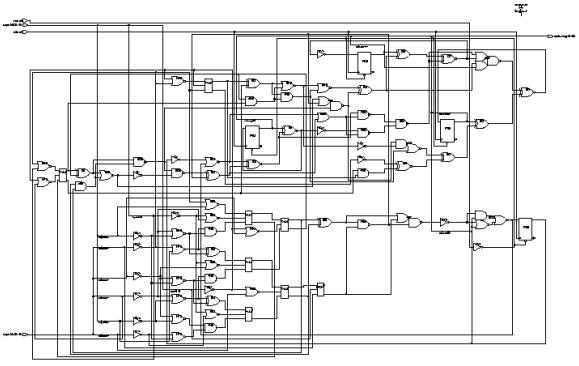
  `file` ⇒ `read` ⇒ `macc.vhd` ⇒ `OK`





Example design - MACC

- **design optimization**
  - **`tools` ⇒ `design optimization` ⇒ `OK`**
  - **`file` ⇒ `save as` ⇒ `DB, VHDL, SXNF`**



Synthesis and Gate Level Simulation with Synopsys

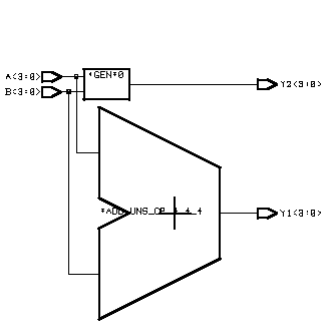
Valentina Salapura
Michael Gschwind

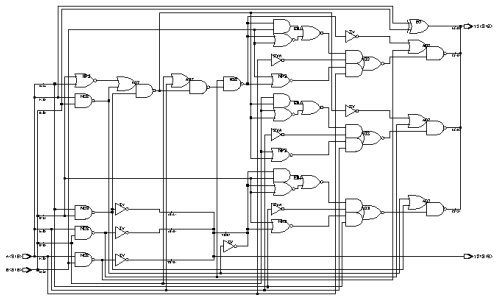
Synopsys design compiler

- dominant synthesis tool set in the market
- probably the best synthesis tool available today
- synthesis executed in two steps
  - module generator
    - » VHDL description mapped to modules
    - » modules for operators
    - » technology independent
  - technology mapping
    - » modules mapped to gates
    - » technology specific
    - » starting point for the optimization at the gate level

Two synthesis steps

- step 1: simple operators mapped to modules
- step 2: operators mapped to gates and optimization at the gate level





What Synopsys offers

- various options for exact specification of
  - technology and working environment
    - » target technology definition
    - » max fan-out, max transition, max capacitance
  - design optimization
    - » area and timing optimization
    - » clock specification
    - » various optimization efforts
    - » various optimization methods available (boundary optimization, flatten, Boolean)

What can Synopsys do

- read in and use user-defined libraries, packages, etc.
- generate design reports
  - » design complexity (number of gates/technology elements)
  - » design speed
  - » used resources (also as macros from libraries)

Starting design compiler

- with `"design_analyzer"` for graphical front-end
- with `"dc_shell"` for command line oriented interface
  - graphical front-end translates pull-down menu selections to script commands
    - » can be seen in command window
  - not all commands can be selected using the graphical interface

Selecting commands

- selecting choices in pull-down menus
- writing commands in command window
  - open with `setup` ⇒ `command window`
- directly in `dc_shell`

Working in batch mode

- using scripts
- scripts for synthesis control
  - advantageous for complex designs
    - » synthesis and optimization executed unattended over night or longer time
  - also useful for less complex designs
- each menu selection can be substituted with a script command
- for advanced usage
  - wider command selection
  - reproducible ⇒ script documents selected synthesis and optimization
  - reduced setup work for repeated synthesis

Script language

- design compiler controlled using script language
- various command classes for
  - script control
  - reading and saving designs
  - target technology definition
  - translation of VHDL programs
  - definition of module interfaces
  - optimization control
  - report generation

Script control

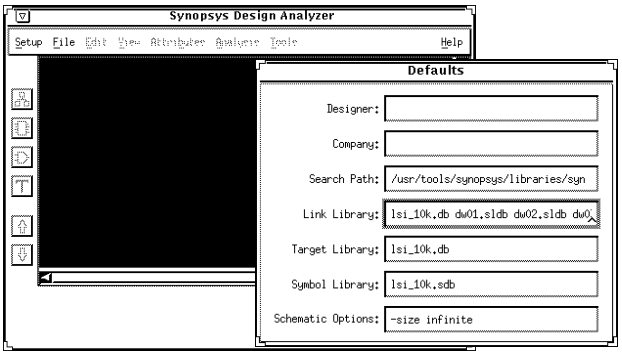
- menu selection
  `setup ⇒ execute script`
- include scripts in command window and in script files
  `include` ***`script_file_name`***
- in program `dc_shell`
  `dc_shell -f` ***`script_file_name`***

Reading and saving designs

- support for various design formats
  - VHDL, Verilog, DB, XNF, EDIF, PLA ...
- reading designs with `read`
  - example: `read -f xnf macc.xnf`
- saving designs with `write`
  - option `-hierarchy` saves all submodules
  - example:
    `write -f db -hierarchy -output macc.db`

Technology definition

- specified in file `.synopsys_dc.setup`
- menu selection `setup` ⇒ `defaults`



DesignWare library

- module generator
- contains different library elements
  - implementation of operators +, - *, /
  - different modules like RAMs, shift registers, etc.
- various implementations for same functionality
  - different trade-offs, area or speed optimized
- appropriate implementation selected automatically in Synopsys as a result of specified design constraints

File `.synopsys_dc.setup`

- an example of file `.synopsys_dc.setup`

```
search_path = {. /usr/tools/synopsys/libraries/syn }

target_library = lsi_10k.db
link_library = lsi_10k.db
symbol_library = lsi_10k.sdb

/* design ware */
synthetic_library = synthetic_library + { dw01.sldb
  dw02.sldb dw03.sldb dw04.sldb dw05.sldb }
link_library = link_library + { dw01.sldb dw02.sldb
  dw03.sldb dw04.sldb dw05.sldb }
```

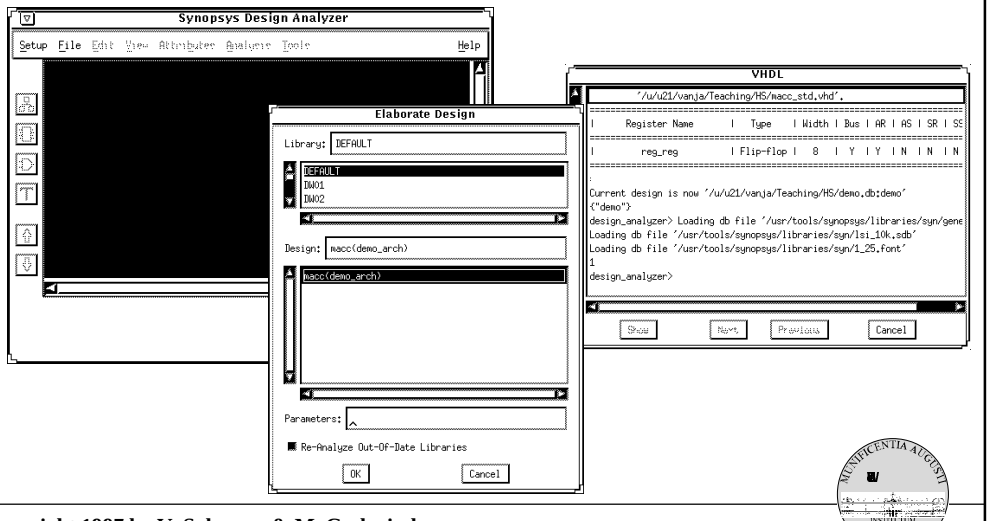
Translation of VHDL programs

- **in two steps**
  - **analyze**
    - » **translates the design into the working library**
  - **elaborate**
    - » **generates design structure from description in the working library**
- **menu selections**

  `file ⇒ analyze`

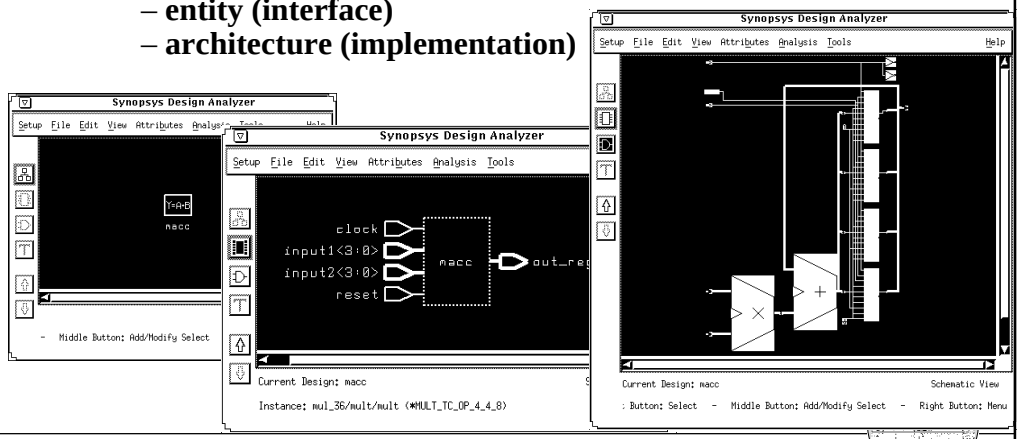
  `file ⇒ elaborate`

Translation of VHDL programs



Different design views

- design represented at different hierarchical levels
  - module (all modules read)
  - entity (interface)
  - architecture (implementation)



Definition of module interfaces

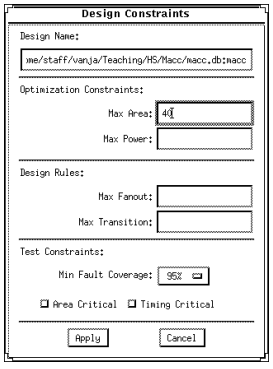
- ICs require special drivers for external signals ⇒ pads
- definition of external ports with `set_port_is_pad`
  - typical examples
    - » `set_port_is_pad all_inputs()`
    - » `set_port_is_pad "*"`
- insertion of pads with `insert_pads`

Commands for optimization control

- design optimization
  - area minimization
  - timing constraints
- controlling of different optimization methods
  - `set_boundary_optimization current_design()`
    - » constants on ports propagated to submodules

Area optimization

- menu selection `attributes` ⇒ `optimization constraints` ⇒ `design constraints`



- command `set_max_area 40`

Signal run time optimization

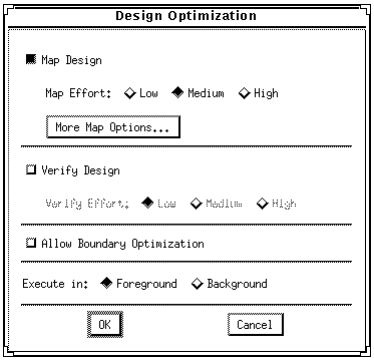
- **menu selection**
  `attributes ⇒ optimization constraints ⇒ timing constraints`
- **command `set_max_delay`**
  - **various options (rise, fall)**
  - **path specification with start and/or end point**
    `set_max_delay 15 -from port -to port`
- **example**
  `set_max_delay 2 -to out_port`

Synthesis start

- **optimization constraints have to be set before starting synthesis**
- **synthesis started with**
  `compile`
  - **example: `compile -map_effort high`**
  - **additional synthesis control options available**

Starting synthesis

- **starting with**
  `tools` ⇒ `design optimization`



Generating design reports

- **menu selection**
  `analyze` ⇒ `report`
- **report on resources used**
  `report_area`
- **signal delay time report**
  `report_timing`
- **report on FPGA resources**
  `report_fpga`
- **etc. etc.**

Area report

- **example: area consumption of design MACC**

```
Command Window

1
design_analyzer> report_area
Information: Updating design information... (UID-85)

****************************************
Report : area
Design : macc
Version: v3.4a
Date   : Thu Apr 10 15:20:17 1997
****************************************

Library(s) Used:

    lsi_10k (File: /usr/tools/synopsys/libraries/syn/lsi_10k.db)

Number of ports:              14
Number of nets:               31
Number of cells:              16
Number of references:         11

Combinational area:       179.000000
Noncombinational area:     36.000000
Net Interconnect area:     undefined  (No wire load specified)

Total cell area:          215.000000
Total area:                undefined
1
design_analyzer>
```

Timing report

```
Command Window

design_analyzer> report_timing -path full -delay max -max_paths 1 -nworst 1

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : macc
Version: v3.4a
Date   : Thu Apr 10 15:20:19 1997
****************************************

  Operating Conditions:
  Wire Loading Model Mode: top

  Startpoint: input1<3> (input port)
  Endpoint: reg_reg<3> (rising edge-triggered flip-flop clocked by clock)
  Path Group: clock
  Path Type: max

  Point                                                   Incr       Path
  ---------------------------------------------------------------------------
  clock (input port clock) (rise edge)                    0.00       0.00
  input external delay                                    0.00       0.00 r
  input1<3> (in)                                          0.00       0.00 r
  mul_36/mult/mult/BK3> (macc_DW02_mult_4_4_0)            0.00       0.00 r
  mul_36/mult/mult/U13/Z (IVA)                            0.60       0.60 f
  mul_36/mult/mult/U24/Z (NR2)                            1.07       1.67 r
  mul_36/mult/mult/U9/Z (AN2)                             0.77       2.44 r
  mul_36/mult/mult/S3_2_2/CO (FA1A)                       2.18       4.62 r
  mul_36/mult/mult/S5_2/S (FA1A)                          2.28       6.90 r
  mul_36/mult/mult/U8/Z (ED)                              1.19       8.09 f
  mul_36/mult/mult/FS/A3> (macc_DW01_add_6_0)             0.00       8.09 f
  mul_36/mult/mult/FS/U47/Z (OR2)                         0.97       9.06 f
  mul_36/mult/mult/FS/U15/Z (HD2)                         0.78       9.84 r
  mul_36/mult/mult/FS/SUM<3> (macc_DW01_add_6_0)          1.32      11.16 f
  mul_36/mult/mult/PRODUCT<3> (macc_DW02_mult_4_4_0)      0.00      11.16 f
  U25/Z (NR2)                                             0.81      11.97 r
  U26/Z (EON1)                                            1.12      13.09 f
  U27/Z (AO4)                                             1.44      14.53 r
  U28/Z (ED)                                              1.13      15.66 f
  reg_reg<3>/D (FD2)                                      0.00      15.66 f
  data arrival time                                                 15.66

  clock clock (rise edge)                                20.00      20.00
  clock network delay (ideal)                             0.00      20.00 r
  reg_reg<3>/CP (FD2)                                     0.00      20.00 r
  library setup time                                     -0.85      19.15
  data required time                                                19.15
  ---------------------------------------------------------------------------
  data required time                                                19.15
  data arrival time                                                -15.66
  ---------------------------------------------------------------------------
  slack (MET)                                                       3.49

design_analyzer>
```

Selecting current design

- all commands executed only on current design
- current design selection
  - in `design_analyzer`
    clicking on the module
  - in `dc_shell`
    `current_design` ***`module_name`***

Other useful commands

- `ungroup` and `group`
  - for creating and removing hierarchical levels
    - » e.g., to enable optimization over various subdesigns
  - example
    `ungroup -all`
- `dont_touch`
  - selected module is not further optimized
- `characterize`
  - transfers design constraints from higher hierarchical level to submodules

Example of script file

```
read -f vhdl  test_add.vhd
current_design  add8_impl1
include handle_design_script
report_area  >  add8_area.out
report_timing -path full -delay max
  -max_paths 1 nworst 1 > add8_time.out
write -f db -hierarchy -o add4_impl1.db
```

- **file** `handle_design_script:`

```
set_port_is_pad
ungroup current_design()
set_boundary_optimization current_design()
compile -ungroup_all -map_effort medium
```

Gate level simulation

- **to simulate at the gate level**
  - **generate a simulation library from the synthesis library**
  - **analyze the simulation library**
  - **reference the simulation library in gate-level description**
  - **generate an SDF file from a design**
  - **start the simulation using the SDF file**

Standard delay format (SDF)

- SDF is a standard delay format
  - portable description of design delays
  - supported by many EDA tools
- contains timing information of a synthesized design
- for back-annotating design timing with gate-level simulation

Generating simulation library

- simulation library is generated from the synthesis library file ***`lib_name.db`***
  - **`liban -arch`** ***`model lib_name.db`***
  - option -arch specifys one of five timing models
    - » **`ftbm`** – accurate timing model, slowest simulation
    - » **`udsm`** – all cells with 1ns delay, fastest simulation
    - » **`ftsm`** – each cell with accurate timing, less accurate than **`ftbm`**, slower simulation than **`udsm`**
    - » **`ftgs`** – as accurate as **`ftbm`**, fast simulation (optimized for Synopsys)
    - » **`vital`** – vital model
- results of compilation are files **`lib_name_components.vhd`** and **`lib_name_FTSM.vhd.E`**

Analyzing simulation library

- simulation library has to be analyzed

```
vhdlan -w lib_name
  lib_name_FTSM.vhd.E
  lib_name_components.vhd
```

- timing model library files are compiled and stored in the simulation library *lib_name*

Using the simulation library

- simulation libraries are similar to other libraries
- simulation library has to be
  - specified in the setup file
  - referenced in the VHDL description

Specifying simulation library

- simulation library has to be specified in the setup file `.synopsys_vss.setup`
- for simulation library generated from LSI 10k synthesis library example
  - insert in the file `.synopsys_vss.setup`

```
LSI_10K: $SYNOPSYS/libraries/sim/lsi_10k
```

Referencing simulation library

- simulation library has to be referenced in the VHDL file to be simulated
  - for the LSI 10k example
    - » insert in the VHDL gate level file

```
library lsi_10k;
use lsi_10k.components.all
```

- this can be automated by putting the following command in the setup file `.synopsys_dc.setup`

```
vhdlout_use_packages =
  {"LSI_10K.COMPONENTS"};
```

Generating an SDF file from a design

- SDF file contains timing information of a synthesized design
  - required for gate level simulation
- generation in command window
  - select file format for VHDL simulation

```
change_names -rules vhdl -hierarchy
```

  - generate SDF file

```
write_timing -format sdf -output
  sdf_file_name
```

Starting the gate level simulation

- gate level simulation uses the timing information from an SDF file
- gate level simulation has to be started in command mode

```
vhdlsim -sdf_top /tb/uut
  -sdf sdf_file_name cfg_tb_name
```

  - option `-sdf_top` specifies the design described by SDF file is (UUT in the above example)
  - the last argument (*`cfg_tb_name`*) specifies the test bench configuration

Simulation in command mode

- command mode simulator **vhdlsim** uses the same commands as the window mode **vhdldbx**
  - **trace**
    - » signal selection for waveforms
  - **cd**
    - » moving through the program hierarchy
  - **step, next**
    - » execute simulation in steps
  - **run *time_expression***
    - » simulation start
  - **include**
    - » usage of scripts

Simulation example: MACC gate-level description

```
library IEEE;
library LSI_10K;
use IEEE.std_logic_1164.all;
use LSI_10K.COMPONENTS.all;
entity macc is
   port(input1,input2: in std_logic_vector (3 downto 0);
        reset,clock: in std_logic;
        out_reg : out std_logic_vector (7 downto 0));
end macc;
architecture SYN_demo_arch of macc is
   component IVA
      port(A: in std_logic;  Z : out std_logic);
   end component;
   component FD2
      port(D,CP,CD: in std_logic; Q,QN: out std_logic);
   end component;                    ...
```

referencing the simulation library

Simulation example: MACC SDF file

```
DELAYFILE
(SDFVERSION "OVI 1.0")
(DESIGN "macc")
(DATE "Tue May 13 15:37:44
  1997")
(VENDOR "lsi_10k")
(PROGRAM "Synopsys Design
  Compiler cmos")
(VERSION "v3.4a")
(DIVIDER /)
(VOLTAGE 5.00:5.00:5.00)
(PROCESS)
(TEMPERATURE
  25.00:25.00:25.00)
(TIMESCALE 1ns)
```

```
(CELL
(CELLTYPE "FA1A")
(INSTANCE add_37_plus_plus/U1_5)
(DELAY
(ABSOLUTE
(IOPATH A CO (2.182:2.182:2.182)
  (2.274:2.274:2.274))
(IOPATH B CO (2.182:2.182:2.182)
  (2.274:2.274:2.274))
(IOPATH CI CO (1.082:1.082:1.082)
  (1.174:1.174:1.174))
(IOPATH A S (2.136:2.136:2.136)
  (1.925:1.925:1.925))
(IOPATH B S (2.136:2.136:2.136)
  (1.925:1.925:1.925))
(IOPATH CI S (1.036:1.036:1.036)
  (0.825:0.825:0.825)) ))
```

Simulation example: starting the simulation

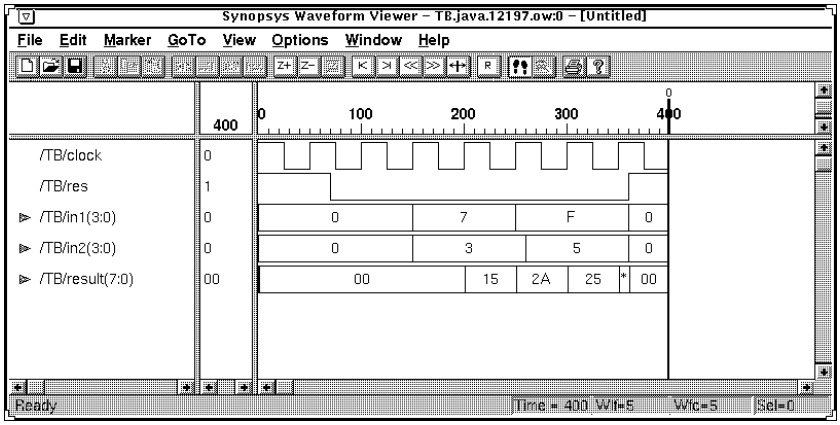
- simulation started with

```
vhdlsim -sdf_top /tb/uut
  -sdf macc.timing cfg_tb_macc
```

- simulation performed as usual

```
cd tb
trace clock res in1 in2 result
run 400
```

Simulation example: waveforms



Exercise 2
Synthesis and gate level simulation

Valentina Salapura
Michael Gschwind

Synthesis and gate level simulation

- use ALU description from exercise 1 as a starting point
- modify ALU to use `std_logic_vector` for inputs and output
- synthesize the ALU for an ASIC target technology
  - e.g., LSI 10k included in Synopsys standard distribution

Complete the following tasks

- synthesize and optimize the ALU
  - optimized for area
  - optimized for timing
- compare results generated by area and timing optimization
- explain why `after` clause is ignored
- simulate the synthesis results using test bench from exercise 1
- explain why only a subset of VHDL can be synthesized

Modeling Sequential Logic and Finite State Machines

Valentina Salapura
Michael Gschwind

Building complex VHDL models

- **VHDL supports modeling at different abstraction levels**
  - **language constructs for each abstraction level**
  - **behavioral level**
    - » **functions, procedure calls, processes**
  - **register transfer level (RTL)**
    - » **concurrent statements and formulae**
  - **structural level (VHDL netlist)**
    - » **components, component instantiation, component interconnections**

Hybrid descriptions

- language constructs for different abstraction levels can be mixed in a single description
  - using constructs of all abstraction levels in single architecture
    - » process for algorithm description
    - » RTL formulae
    - » component instantiation
  - useful for simulation and synthesis
    - » re-use of existing modules in description of new modules through component instantiation
    - » mixing of descriptions with different amount of detail

Component declaration and instantiation

- component declared in the declaration part of an architecture body
- syntax

```
component identifier is
    generic(generic_constants_list);
    port(port_list);
end component;
```

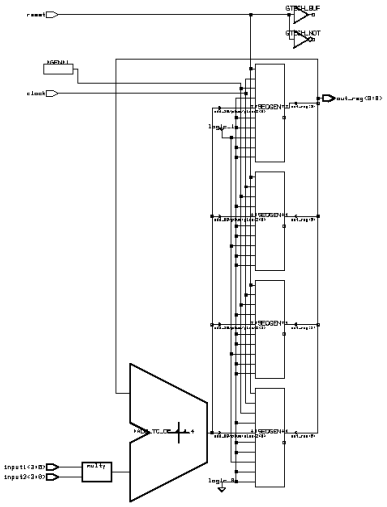
- component instantiated in the statements part of an architecture body
- syntax

```
label: name generic map(generic_list);
            port map(port_list);
```

Hybrid design example: MACC

```
ARCHITECTURE demo_arch OF macc IS
  COMPONENT multy
     PORT(in1, in2: IN  std_logic_vector(3 downto 0);
          res:      OUT std_logic_vector(3 downto 0));
  SIGNAL new_val, reg, res: std_logic_vector(3 downto 0);
BEGIN
  Reg: PROCESS(clock, reset)  BEGIN
     IF reset = '1' THEN
       reg <= "0000";
     ELSIF clock'EVENT AND clock = '1' THEN
       reg <= new_val;
     END IF;
  END PROCESS;
  Multiply: multy port map(input1, input2, res);
  new_val <= reg + res;
  out_reg <= reg;
END demo_arch;
```


Hybrid design example: MACC schematic

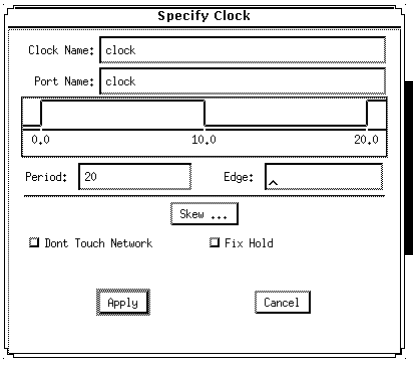


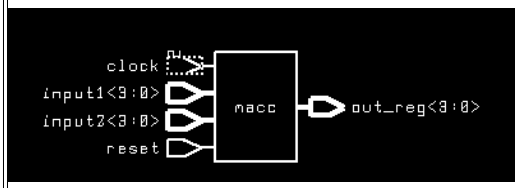
Synthesis of sequential logic

- to synthesize sequential logic need to define
  - clock(s) – requires special treatment
    - » stronger drivers
    - » uniform distribution
  - memory element type
    - » selection of flip flops and latch types to be generated by logic synthesis
    - » type of memory element (whether flip flops or latches are used) depends on VHDL description style

Setting the clock

- menu selection `attributes` ⇒ `clocks` ⇒ `specify`
  - for selected port of a module
  - set clock period and edges





Clock specification

- **in dc_shell and in command window**

```
create_clock -name "clock" -period 20
  -waveform {0 10} find(port,"clock")
```

- **explanation**
  - `-name "clock"`
    - » **specify clock name**
  - `-period 20`
    - » **period of the clock**
  - `-waveform {"0" "10"}`
    - » **rise and fall times for one period**
  - – `find(port,"clock")`
    - » **clock is specified for this port**

Selection of memory elements type

- **selection of particular flip flops or latches styles**
  - – **optional**
- **for exact specification of memory elements**

```
set_register_type -flip_flop "FD1S"
```

  - » **simple edge-triggered D flip flop**

```
set_register_type -latch -exact "LD1P"
```

  - » **positive edge active D latch**
- **menu selection**

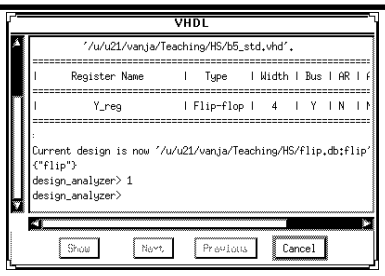
attributes ⇒ optimization directives ⇒ design attributes

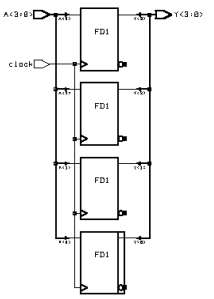
Description of memory elements

- **memory elements generated for signals which are not defined for all possible conditions in conditional assignments**
- **the type of memory element generated inferred implicitly from the VHDL description**
  - **flip-flop**
    - » **edge-triggered**
  - **latch**
    - » **level sensitive**

Flip-flop register

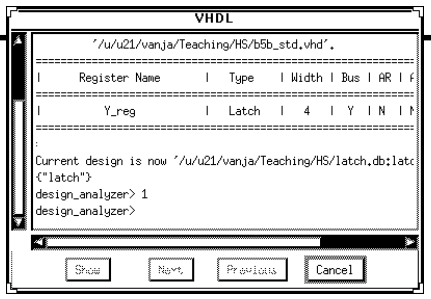
```
ENTITY ff_bank IS
PORT(A: IN  std_logic_vector
            (3 downto 0);
     clock: IN std_logic;
     Y: OUT std_logic_vector
            (3 downto 0));
END ff_bank;

ARCHITECTURE arch OF ff_bank IS
BEGIN
  PROCESS (A, clock) BEGIN
    IF clock = '1' AND clock'EVENT THEN
      Y <= A;
    END IF;
  END PROCESS;
END arch;
```

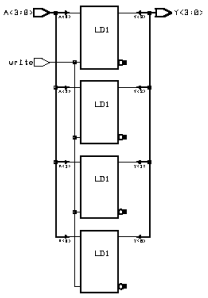




Latch register

```
ENTITY latch_bank IS
  PORT(A: IN  std_logic_vector
              (3 downto 0);
       write: IN std_logic;
       Y: OUT std_logic_vector
              (3 downto 0));
END latch_bank;

ARCHITECTURE arch OF latch_bank IS
BEGIN
  PROCESS (A, write) BEGIN
    IF write = '1' THEN
      Y <= A;
    END IF;
  END PROCESS;
END arch;
```





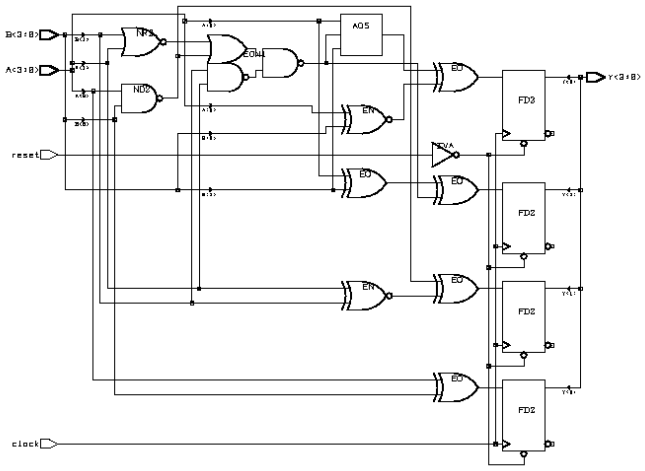
Combinational and sequential logic

- **most designs contain *both* combinational *and* sequential logic**

```
LIBRARY IEEE, work;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use work.all;

ENTITY abff IS
 PORT(A,B: IN std_logic_vector
             (3 downto 0);
      clock,reset: IN std_logic;
      Y:  OUT std_logic_vector
             (3 downto 0));
END abff;
```

```
ARCHITECTURE arch OF abff IS
BEGIN
  PROCESS (clock, reset)
  BEGIN
    IF reset = '1' THEN
      Y <= "0000";
    ELSIF clock'EVENT AND
          clock = '1' THEN
      Y <= A + B;
    END IF;
  END PROCESS;
END arch;
```

Synthesized design

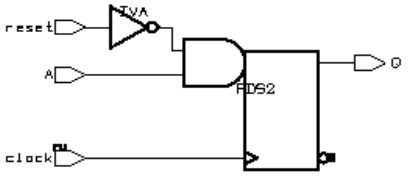


Design initialization

- all registers in a design are set to an initial value
- registers can be initialized using
  - synchronous reset
    - » executed on the next clock edge
    - » activated by positive or negative clock edge
  - asynchronous reset
    - » executed independently on the clock

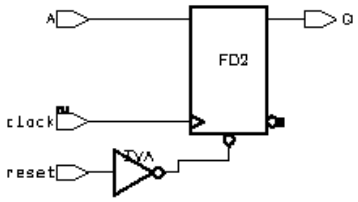
Synchronous reset

```
PROCESS(clock, reset, A)
  BEGIN
  IF clock'EVENT AND clock = '1' THEN
     IF reset = '1' THEN
        Q <= '0';
     ELSE
        Q <= A;
     END IF;
  END IF;
END PROCESS;
```



Asynchronous reset

```
PROCESS(clock, reset, A)
  BEGIN
  IF reset = '1' THEN
     Q <= '0';
  ELSIF clock'EVENT AND
        clock = '1' THEN
     Q <= A;
  END IF;
END PROCESS;
```



Example sequential logic: counter

- synchronous 8-bit counter
- asynchronous reset
- selection between counting up and down
  - selected with signal `dir`
  - increments if `dir = '1'`
  - decrements if `dir = '0'`

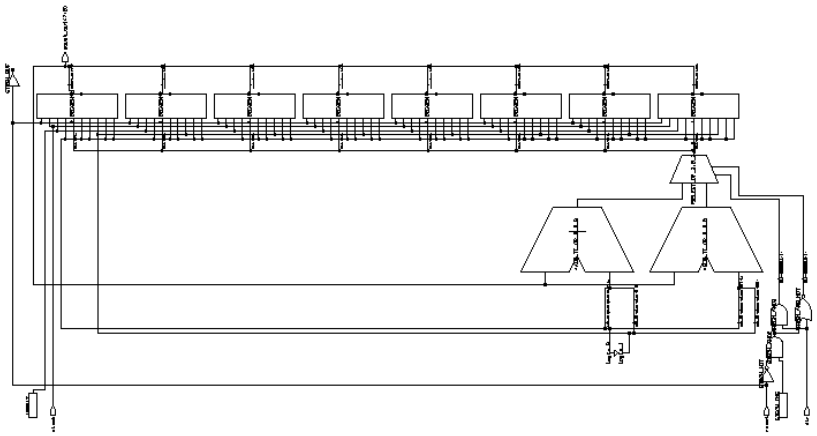
VHDL implementation of the counter

```
ENTITY counter IS
  PORT (
  reset : IN  std_logic;
  clock : IN  std_logic;
  dir   : IN  std_logic;
  count_out:  OUT
         std_logic_vector
         (7 downto 0));
END counter;

ARCHITECTURE arch OF counter
  IS
  SIGNAL count:
         std_logic_vector
         (7 downto 0);
BEGIN
```

```
Count_reg: PROCESS(clock,reset)
BEGIN
  IF reset = '1' THEN
    count <= "00000000";
  ELSIF clock'EVENT AND
    clock = '1' THEN
    IF dir = '1' THEN
      count <= count + 1;
    ELSE
      count <= count - 1;
    END IF; END IF;
END PROCESS;
count_out <= count;
END arch;
```

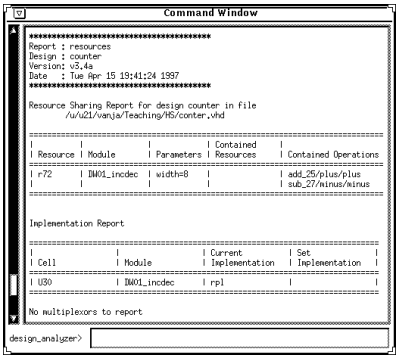
5-21

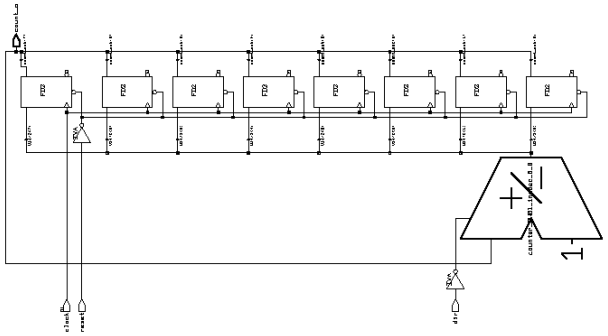
Elaboration generates structural representation



5-22

Optimized synthesis result





5-23

Example sequential logic: serial-to-parallel converter

- serial-to-parallel converter
- synchronous 8 bit conversion
- asynchronous reset
- accepts serial data stream
- packs 8 bits in a byte
- outputs byte

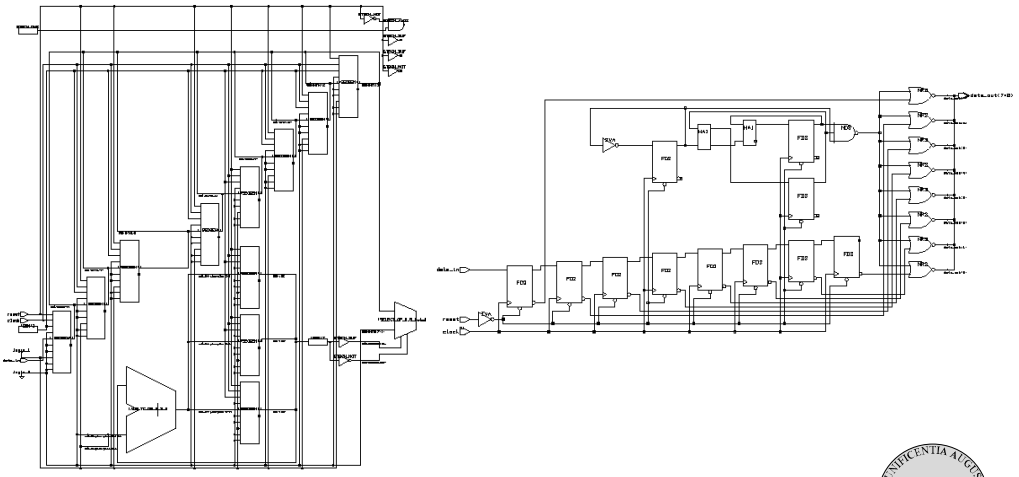
5-24

Serial/parallel source

```
ENTITY parallel IS
  PORT (
    reset : IN  std_logic;
    clock : IN  std_logic;
    data_i: IN  std_logic;
    data_o: OUT std_logic_vector
              (7 downto 0));
END parallel;

ARCHITECTURE arch OF parallel IS
  SIGNAL reg: std_logic_vector
            (7 downto 0);
  SIGNAL count: std_logic_vector
              (2 downto 0);
BEGIN
```

```
Shift: PROCESS(clock, reset)
BEGIN
IF reset = '1' THEN
    reg <= "00000000";
ELSIF clock'EVENT AND clock= '1' THEN
    reg <= data_i & reg(7 downto 1);
END IF; PROCESS;

Counter: PROCESS(clock, reset)
BEGIN
IF reset = '1' THEN
      count <= "000";
ELSIF clock'EVENT AND clock= '1' THEN
      count <= count + 1;
END IF;  END PROCESS;
data_o <= reg WHEN count = "111"
              ELSE "00000000";
END arch;
```

Implementation

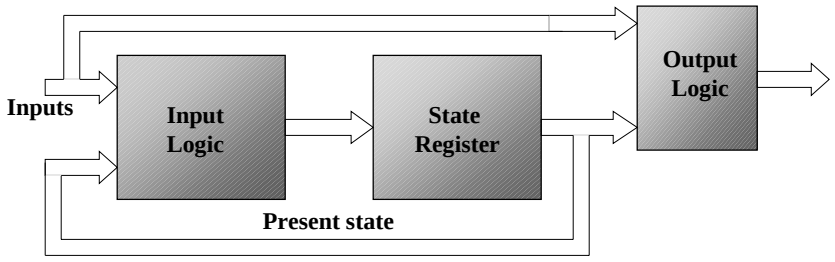


Description of finite state machines

- finite state machines important in hardware design
  - for design of control units
- finite state machines = FSM
- two types of FSM
  - Mealy FSM
  - Moore FSM

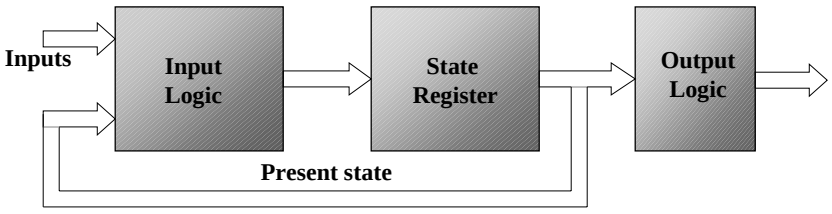
Mealy finite state machines

- outputs of the Mealy FSM depend on the current state of the machine and on the current inputs



Moore finite state machines

- outputs of the Moore FSM depend only on the current state of the machine



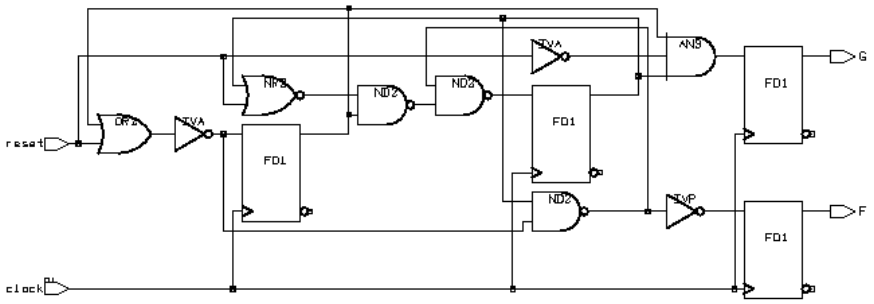
VHDL description styles of FSM

- **FSM description style depends on**
  - **number of state signals**
    - » **one signal for modeling both current and next state of the FSM**
    - » **two signals used for describing current and next state of the FSM**
  - **number of processes in the architecture**
  - **how combinational and sequential logic are allocated to processes**

Explicit description

```
ENTITY fsm IS
PORT(
 reset, clock: IN std_logic;
 F, G : OUT  std_logic);
END fsm;
ARCHITECTURE expl OF fsm IS
  TYPE state_type IS
  (idle, start, stop, clear);
  SIGNAL state: state_type;
BEGIN
PROCESS(clock, reset, state)
BEGIN
IF clock'EVENT AND clock= '1' THEN
  IF reset = '1' THEN
     state <= idle;
     F <= '0'; G <= '0';
  ELSE
CASE state IS
WHEN idle => state <= start;
     G <= '0';
WHEN start => state <= stop;
WHEN stop => state <= clear;
     F <= '1';
WHEN clear => state <= idle;
     F <= '0';  G <= '1';
END CASE;END IF;END IF;END PROCESS;
END expl;
```

Resulting design

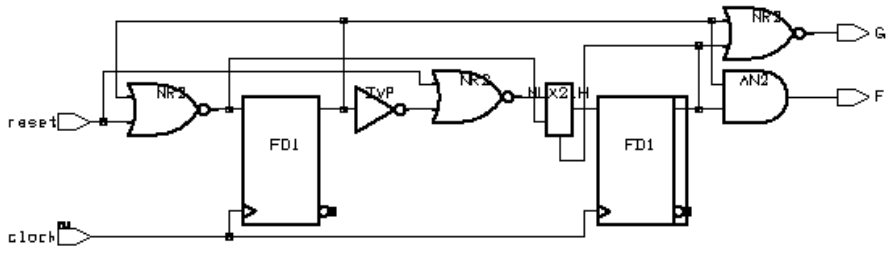


Separated generation of outputs

```
ARCHITECTURE separate OF fsm IS
  TYPE state_type IS
  (idle, start, stop, clear);
  SIGNAL state: state_type;
BEGIN
Output: PROCESS(state)
  BEGIN
    F <= '0';
    G <= '0';
    IF state = clear THEN
      F <= '1';
    END IF;
    IF state = idle THEN
      G <= '1';
    END IF;
  END PROCESS;
```

```
Transition: PROCESS(clock)
  BEGIN
IF clock'EVENT AND clock = '1'
  THEN
IF reset='1' OR state=clear THEN
  state <= idle;
ELSIF state = idle THEN
  state <= start;
ELSIF state = start THEN
  state <= stop;
ELSE
  state <= clear;
END IF; END IF;
END PROCESS;
END separate;
```

Resulting design



Separated sequential and combinational logic

```
ARCHITECTURE reg OF fsm IS
  TYPE state_type IS
  (idle, start, stop, clear);
  SIGNAL state, next_state:
  state_type;
BEGIN
State_reg: PROCESS(clock)
BEGIN
IF clock'EVENT AND clock ='1'
  THEN
  IF reset = '1' THEN
    state <= idle;
  ELSE
    state <= next_state;
END IF; END IF;
END PROCESS;
```

```
Combination: PROCESS(state)
  BEGIN
    F <= '0'; G <= '0';
    IF state = clear THEN
      next_state <= idle;
      F <= '1';
    ELSIF state = idle THEN
      next_state <= start;
      G <= '1';
    ELSIF state = start THEN
      next_state <= stop;
    ELSE
      next_state <= clear;
    END IF; END PROCESS;
END reg;
```

Separated in three processes

```
ARCHITECTURE three OF fsm IS
  TYPE state_type IS
  (idle, start, stop, clear);
  SIGNAL state, next_state:
  state_type;
BEGIN
State_reg: PROCESS(clock)
BEGIN
IF clock'EVENT AND
  clock = '1' THEN
  IF reset = '1' THEN
    state <= idle;
  ELSE
    state <= next_state;
END IF; END IF;
END PROCESS;
```

```
Comb_Output: PROCESS(state) BEGIN
CASE state IS
  WHEN clear => next_state <= idle;
  WHEN idle  => next_state <= start;
  WHEN start => next_state <= stop;
  WHEN others => next_state <= clear;
END CASE;
END PROCESS;
Output: PROCESS(state) BEGIN
    F <= '0'; G <= '0';
    IF state = clear THEN
      F <= '1';
    ELSIF state = idle THEN
      G <= '1';
    END IF;
END PROCESS;
END three;
```

Comparison of description styles

description style	# of state signals	# of processes	# FFs	# of gates	
explicit	1	1	4	39	*wrong functionality!!!*
separated outputs	1	2	2	24	
separated seq. und comb.	2	2	2	24	
3 processes	2	3	2	24	

Resource Sharing

Valentina Salapura
Michael Gschwind

Resource sharing

- **single functional unit used several times from different design parts**
  - **for example, for two "+" in source code sometimes can be used the same adder**
- **resource sharing (RS) tries to allocate different operators to same functional unit**
- **the quality of resource sharing depends on the synthesis tool**

How resource sharing works

- in most synthesis tools resource sharing depends on results of static analysis
  - resource sharing generated only for statically disjoint operations
    - » for example, in different branches of IF and CASE statements
- resource sharing introduces multiplexors to multiplex various inputs to the shared unit
  - multiplexor introduces additional delay

Resource sharing example: accumulator

- implement 8-bit accumulator
- synchronous reset
- adds or subtracts input
  - operations + or - selected with signal `ctl`
  - add if `ctl = '1'`
  - subtract if `ctl = '0'`
- both operations should use the same add/subtract unit

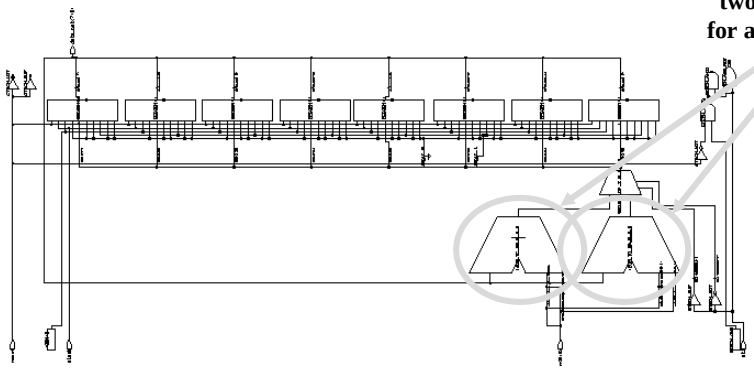
RS example: VHDL description

```
ENTITY accu IS
   PORT (
   reset   : IN   std_logic;
   clock   : IN   std_logic;
   ctl     : IN  std_logic;
   data_in : IN
             std_logic_vector
                (3 downto 0);
   data_out: OUT
             std_logic_vector
             (7 downto 0));
END accu;

ARCHITECTURE arch OF accu IS
  SIGNAL accu:
             std_logic_vector
             (7 downto 0);
BEGIN
Accumulator:PROCESS(clock,reset)
BEGIN
  IF reset = '1' THEN
    accu <= "00000000";
  ELSIF clock'EVENT AND
    clock = '1' THEN
    IF ctl = '1' THEN
       accu <= accu + data_in;
    ELSE
       accu <= accu - data_in;
    END IF; END IF;
END PROCESS;
data_out <= accu;
END arch;
```

RS example: elaboration result

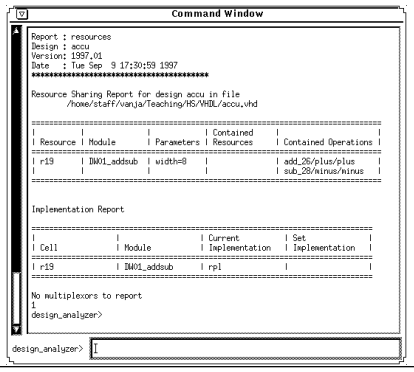
- **schematic generated from VHDL description during elaboration**
  - **no resource sharing**
    - » **addition and subtraction mapped to two units**

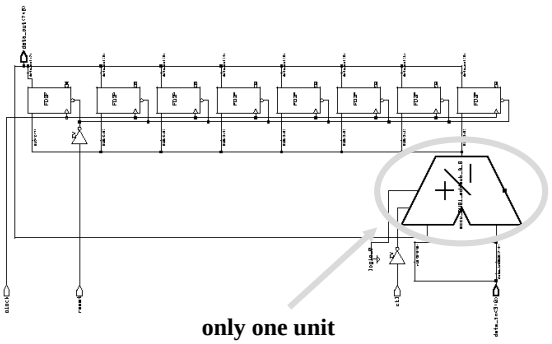
two separate units for add and subtract



RS example: optimized design

- **RS introduced in optimization**
  - **resource report lists shared units and operations mapped to them**





only one unit for add and subtract operations

Resource sharing in Synopsys

- **options**
  - **no RS**
  - **automatic RS**
  - **user controlled RS**
- **automatic RS used by default**
  - **only if design timing allows introduction of additional delay for multiplexor**
  - **RS problem for time constrained designs**

Time constrained designs

- automatic RS can be problem for timing critical designs
  - RS introduces multiplexor for different resource inputs ⇒ design slowed down
  - even a minimal timing violation eliminates RS
  - possible result ⇒ resource explosion
  - solution: user controlled RS

User controlled resource sharing

- synthesis tool sometimes generates no RS
  - does not recognize that a functional unit can be shared
- user can still control synthesis tool to introduce RS
  - introduced by describing it in VHDL programs
- user defines resources which are to be shared
  - using VHDL construct `attribute`
  - VHDL description has to be modified

Automatic RS example: VHDL source

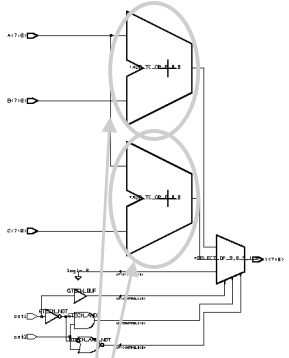
```
LIBRARY IEEE, work;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_signed.ALL;
use work.ALL;

ENTITY share IS
 PORT (
   A,B,C: IN  std_logic_vector
              (7 downto 0);
   cnt1:  IN  std_logic;
   cnt2:  IN  std_logic;
   Y:     OUT std_logic_vector
              (7 downto 0));
END share;
```

```
ARCHITECTURE arch OF share IS
BEGIN
  PROCESS(A, B, C, cnt1, cnt2)
  BEGIN
    IF cnt1 = '1' THEN
      Y <= A + B;
    ELSIF cnt2 = '1' THEN
      Y <= A + C;
    ELSE
      Y <= "00000000";
    END IF;
  END PROCESS;
END arch;
```

share one adder for these operations

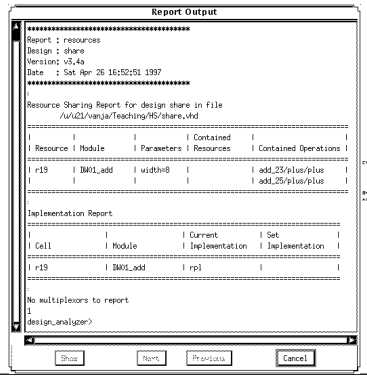
Automatic RS example: synthesis results

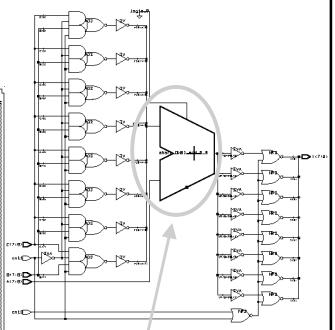
elaboration result



two adders

RS introduced during optimization





only one adder

No RS example: VHDL source

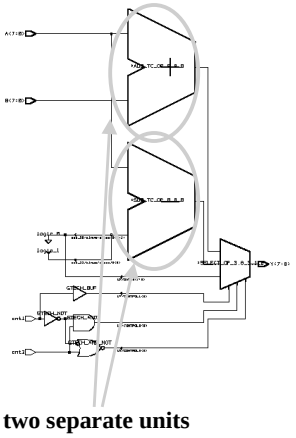
```
LIBRARY IEEE, work;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_signed.ALL;
use work.ALL;

ENTITY no_share IS
  PORT (
    A,B:  IN  std_logic_vector
              (7 downto 0);
    cnt1: IN  std_logic;
    cnt2: IN  std_logic;
    Y:    OUT std_logic_vector
              (7 downto 0));
END no_share;
```

```
ARCHITECTURE arch OF no_share IS
BEGIN
  PROCESS(A, B, cnt1, cnt2)
  BEGIN
    IF cnt1 = '1' THEN
      Y <= A + B;
    ELSIF cnt2 = '1' THEN
      Y <= A - 1;
    ELSE
      Y <= "00000000";
    END IF;
  END PROCESS;

END arch;
```

these two operations can share one add/sub unit

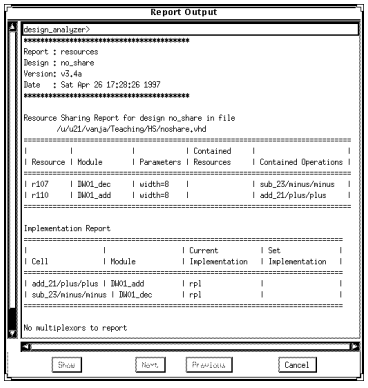
No RS example: synthesis results

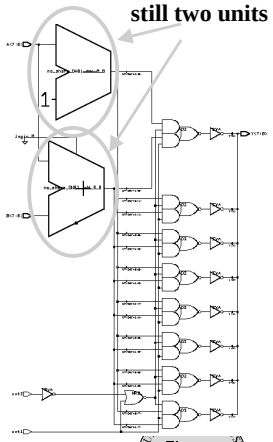
elaboration result



two separate units

no RS introduced in optimization





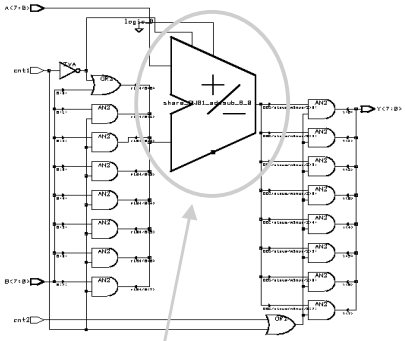
User controlled RS example: VHDL source

```
LIBRARY IEEE, work;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_signed.ALL;
use work.ALL;

ENTITY uc_share IS
  PORT (
    A,B:  IN  ...;
    cnt1,cnt2: IN  ...;
    Y:  OUT ...;
END uc_share;

ARCHITECTURE arch OF uc_shareIS
BEGIN
```

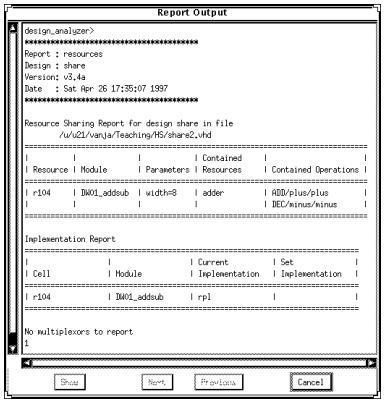
```
PROCESS(A, B, cnt1, cnt2)
 SUBTYPE resource IS integer;
 ATTRIBUTE ops: string;
 CONSTANT adder: resource := 0;
 ATTRIBUTE ops OF adder:
         constant IS "ADD DEC";
BEGIN
 IF cnt1 = '1' THEN
   Y <= A + B;
        -- pragma label ADD
 ELSIF cnt2 = '1' THEN
   Y <= A - 1;
        -- pragma label DEC
 ELSE
  Y <= "00000000";
 END IF; END PROCESS; END arch;
```

changes in VHDL code for user controlled RS

User controlled RS example: synthesis results



only one add/sub unit



What resources can be shared

- not all resources can be shared
- RS possible for units with similar functionality
- typically
  - '+' and '+'
  - '+' and '-'
  - '-' and '-'
  - '>' and '>'
  - '>' and '<'
  - '<' and '<='
  - '>' and '<='
- exact set depends on synthesis tool

Resource implementation

- Synopsys supports several implementations for the same functional unit
  - i.e., several different adder implementations available in DesignWare library
    - » `cla` – carry look-ahead synthesis model
    - » `rpl` – ripple carry synthesis model
    - » `clf` – fast carry look-ahead synthesis model
    - » `sim` – simulation model
  - additional implementations available
    - » from ASIC vendor optimized for target process
    - » from IP vendors

Selecting resource implementation

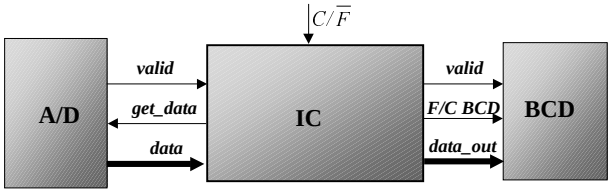
- **resource implementation can be selected**
  - **– indirectly – by setting constraints**
    - **» e.g. for fast design, `cla` model is chosen**
    - **» e.g. for small design, `rpl` model is included**
  - **– directly**
    - **» modifying the VHDL description to select the desired implementation**
    - **» forbidding the optimizer to use slow implementations, e.g. using command to forbid ripple carry model**
      `set_dont_use standard.sldb/*/rpl`

Exercise 3
Design reuse and integration

Valentina Salapura
Michael Gschwind

Design reuse and integration

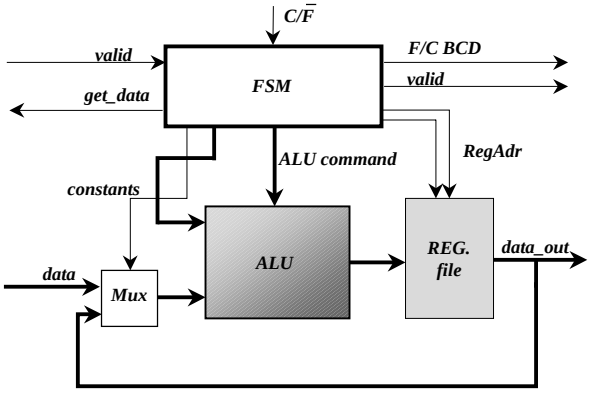
- **design a digital thermometer**
- **thermometer can display temperature in °C or °F**
- **advantageous over analog thermometer**
  - **selection °F/°C with switch**
    - » $C/\overline{F}$ **= 0, output in °F, convert input from °C to output in °F**
    - » $C/\overline{F}$ **= 1, output in °C, no conversion required**



Design overview

- **conversion formula**

  °F = °C * 2 + 32



Explanation of functional units

- **design an FSM for controlling design**
- **design reuse – use existing designs for**
  - **ALU**
  - **register file**
- **use ALU from exercise 2**
  - **functionality must not be changed**
- **use register file from DesignWare library**
  - **use component `DW03_ram2_s_d`**
  - **data sheet for component**
    - » **in Synopsys online documentation**

Control flow of the thermometer

- start A/D converters using signal `get_data`
- wait for results from the A/D converter
  - signal `valid` – data available
  - signal `data` – temperature in °C
- convert the temperature from °C into °F
  - if required, i.e., signal $C/\overline{F}$ = 0
  - use register file for storing intermediate results
- send data to BCD display
  - signal `data_out` – temperature in °F or °C
  - signal `F/C BCD` – display temperature scale
- repeat steps for new temperature input

VHDL description

- describe FSM as a separate entity
- integrate all components of the system using structural VHDL
- VHDL components reference existing designs
  - ALU
  - register file
  - FSM
- implement required glue logic in the VHDL net list

Complete the following tasks

- **design the control unit for this design as FSM in VHDL**
- **integrate all components in a single design**
- **simulate the design to validate functionality**
- **synthesize the design**
- **simulate synthesis results**