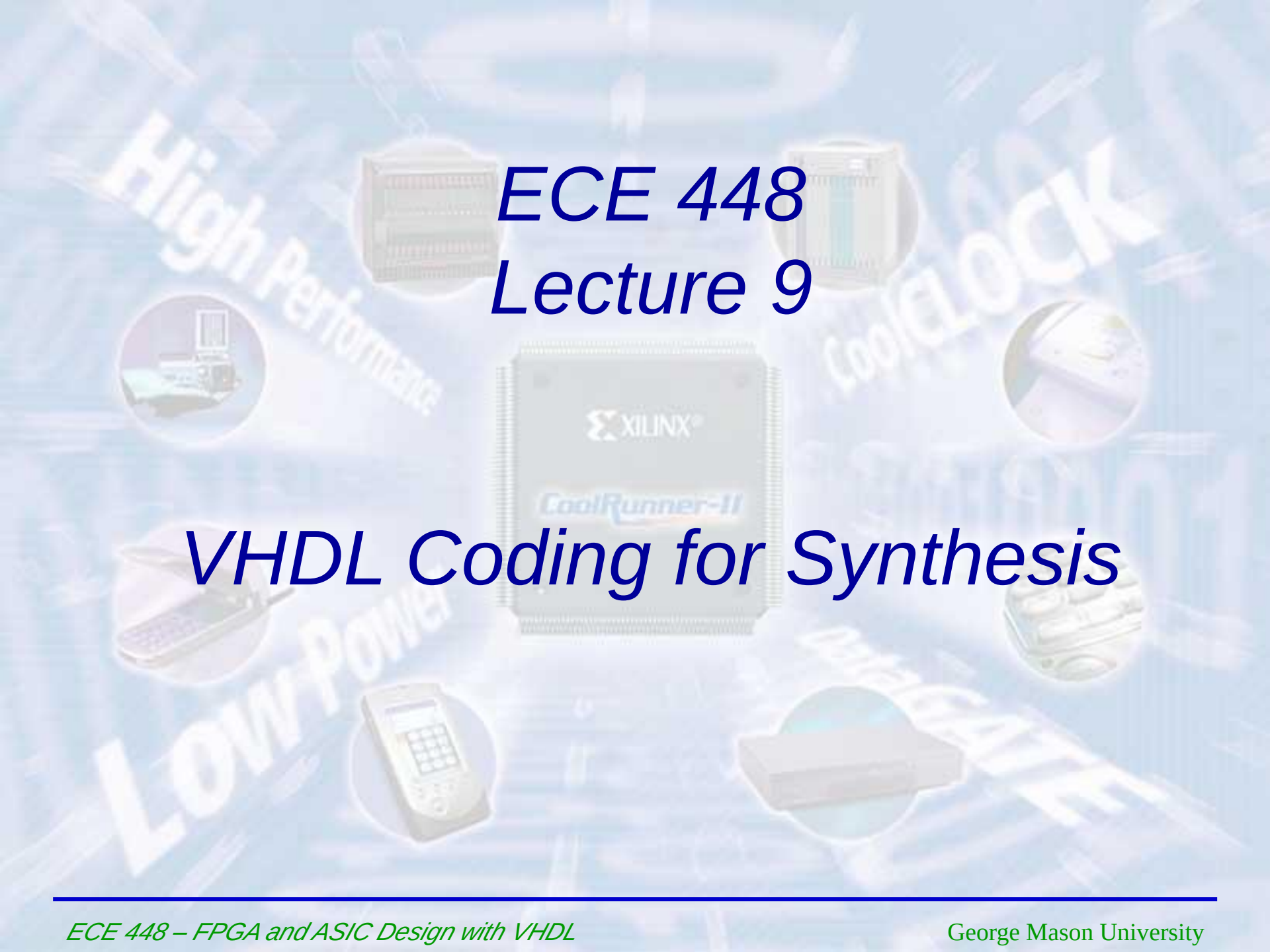



# ECE 448
# Lecture 9

# VHDL Coding for Synthesis

# Required reading

- S. Lee, *Advanced Digital Logic Design,*

***Chapter 4.4, Synthesis Heuristics*** (handout)


# Non-synthesizable VHDL

# Delays

*Delays are not synthesizable*

*Statements, such as*

***wait for** 5 ns*

*a <= b **after** 10 ns*

*will not produce the required delay, and should not be used in the code intended for synthesis.*

# *Initializations*

*Declarations of signals (and variables) with initialized values, such as*

*SIGNAL a : STD_LOGIC* **:= '*0*';**

*cannot be synthesized, and thus should be avoided.*

*If present, they will be ignored by the synthesis tools.*

***Use set and reset signals instead.***

# Reports and asserts

*Reports and asserts, such as*

```
report "Initialization complete";
assert initial_value <= max_value
    report "initial value too large"
    severity error;
```

***cannot be synthesized**, but they*
***can be freely used** in the code intended for synthesis.*

*They will be used during simulation and ignored during synthesis.*

# *Floating-point operations*

*Operations on signals (and variables) of the type*

***real***

*are not synthesizable by the current generation of synthesis tools.*

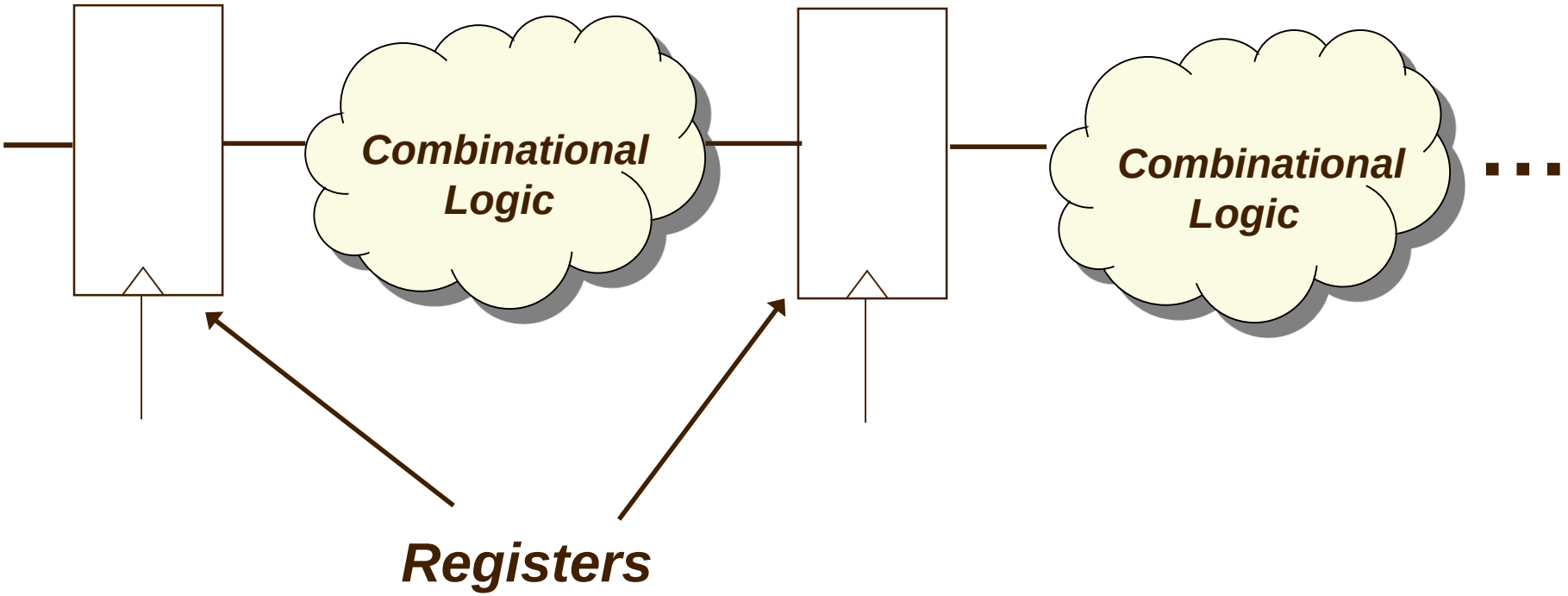
# Dual-edge flip-flops

```
PROCESS ( Clk )
BEGIN
    IF rising_edge(Clk) or falling_edge(CLk) THEN
        Q <= D ;
    END IF ;
END PROCESS ;
```

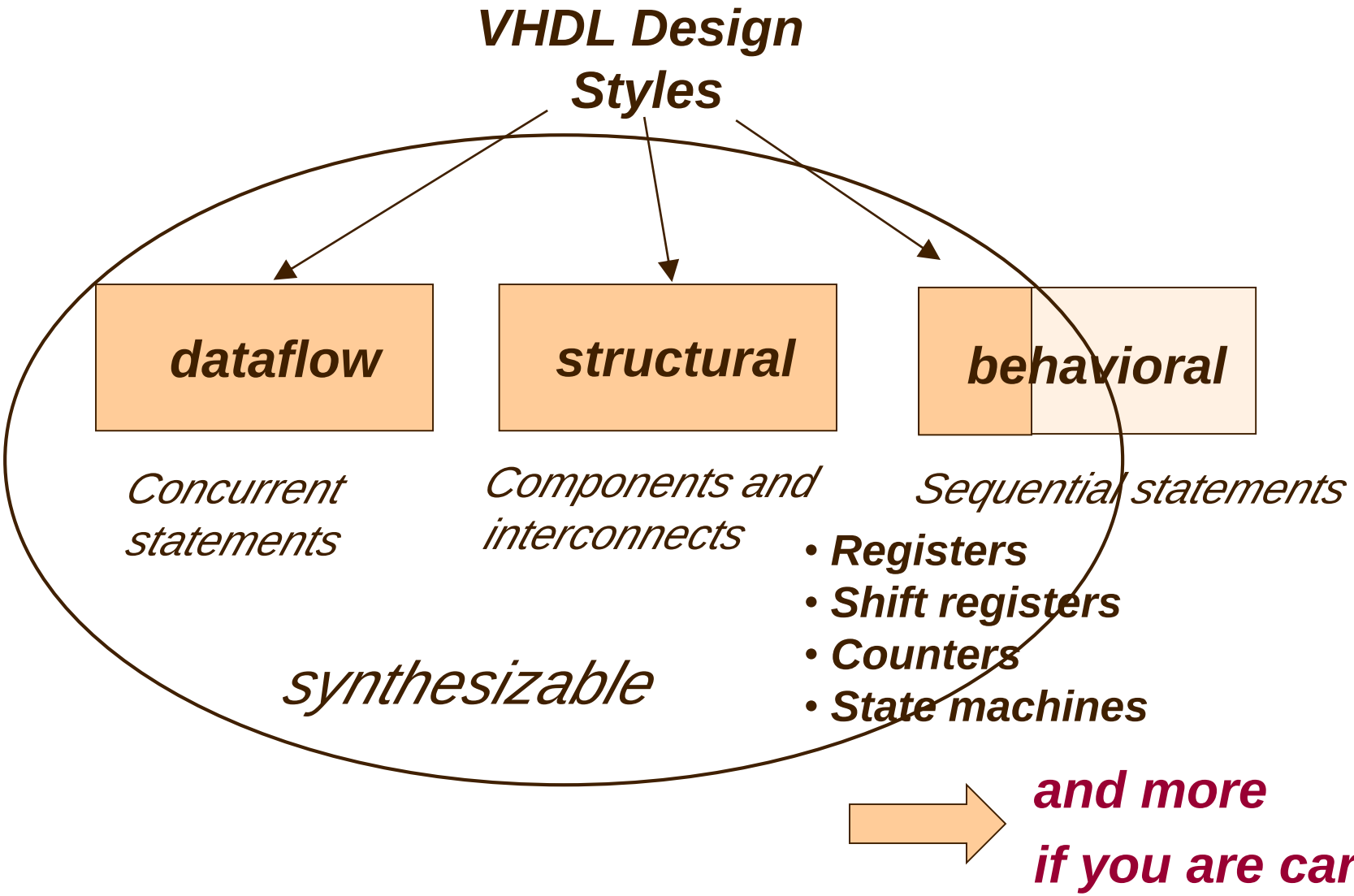
**Dual-edge flip-flops and registers not synthesizable using FPGA tools**


# Synthesizable VHDL

# Register Transfer Level (RTL) Design Description



# VHDL Design Styles




# *Combinational Logic Synthesis for Beginners*

# *Simple rules for beginners*

*For combinational logic,*

*use only concurrent statements*

- *concurrent signal assignment* ($\Leftarrow$)
- *conditional concurrent signal assignment (when-else)*
- *selected concurrent signal assignment (with-select-when)*
- *generate scheme for equations (for-generate)*

# *Simple rules for beginners*

*For circuits composed of*

*- simple logic operations (logic gates)*

*- simple arithmetic operations (addition, subtraction, multiplication)*

*- shifts/rotations by a constant*

*use*

- *concurrent signal assignment* (⇐)

# *Simple rules for beginners*

*For circuits composed of*

*- multiplexers*

*- decoders, encoders*

*- tri-state buffers*

*use*

- *conditional concurrent signal assignment (when-else)*
- *selected concurrent signal assignment (with-select-when)*

# *Left vs. right side of the assignment*

***Left side*** <= ***Right side***

<= *when-else*

*with-select* <=

- *Internal signals (defined in a given architecture)*
- *Ports of the mode*
  - ***out***
  - *inout*

*Expressions including:*

- *Internal signals (defined in a given architecture)*
- *Ports of the mode*
  - ***in***
  - *inout*

# *Arithmetic operations*

*Synthesizable arithmetic operations:*

- *Addition, +*
- *Subtraction, -*
- *Comparisons, >, >=, <, <=*
- *Multiplication, **
- *Division by a power of 2, /2**6 (equivalent to right shift)*
- *Shifts by a constant, SHL, SHR*

# *Arithmetic operations*

*The result of synthesis of an arithmetic operation is a*

*- combinational circuit*

*- without pipelining.*

*The exact internal* ***architecture*** *used (and thus delay and area of the circuit)* ***may depend*** *on the* ***timing constraints*** *specified during synthesis (e.g., the requested maximum clock frequency).*

# Operations on Unsigned Numbers

*For operations on unsigned numbers*

*USE ieee.std_logic_unsigned.all*
*and*
*signals (inputs/outputs) of the type*
*STD_LOGIC_VECTOR*

*OR*

*USE ieee.std_logic_arith.all*
*and*
*signals (inputs/outputs) of the type*
*UNSIGNED*

# Operations on Signed Numbers

*For operations on signed numbers*

*USE ieee.std_logic_signed.all*
*and*
*signals (inputs/outputs) of the type*
*STD_LOGIC_VECTOR*

*OR*

*USE ieee.std_logic_arith.all*
*and*
*signals (inputs/outputs) of the type*
*SIGNED*

# Signed and Unsigned Types

***Behave*** *exactly* ***like***

***STD_LOGIC_VECTOR***

*plus, they determine whether a given vector should be treated as a signed or unsigned number.*

*Require*

***USE ieee.std_logic_arith.all;***

# *Integer Types*

*Operations on signals (variables)*

*of the integer types:*

***INTEGER, NATURAL,***

*and their sybtypes, such as*

*TYPE day_of_month IS **RANGE 0 TO 31;***

*are synthesizable in the range*

$-(2^{31}-1)$ .. $2^{31}$ -1 *for INTEGERs and their subtypes*

0 .. $2^{31}$ -1 *for NATURALs and their subtypes*

# *Integer Types*

*Operations on signals (variables)*

*of the integer types:*

***INTEGER, NATURAL,***

*are less flexible and more difficult to control*

*than operations on signals (variables) of the type*

*STD_LOGIC_VECTOR*

*UNSIGNED*

*SIGNED, and thus*

*are recommened to be avoided by beginners.*

# Addition of Signed Numbers (1)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
      PORT (   Cin              : IN     STD_LOGIC ;
               X, Y             : IN     STD_LOGIC_VECTOR(15 DOWNTO 0) ;
               S                : OUT    STD_LOGIC_VECTOR(15 DOWNTO 0) ;
               Cout, Overflow   : OUT    STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
      SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNTO 0) ;
BEGIN
      Sum <= ('0' & X) + Y + Cin ;
      S <= Sum(15 DOWNTO 0) ;
      Cout <= Sum(16) ;
      Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

# Addition of Signed Numbers (2)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY adder16 IS
        PORT (  Cin                 : IN        STD_LOGIC ;
                X, Y                : IN        SIGNED(15 DOWNTO 0) ;
                S                   : OUT       SIGNED(15 DOWNTO 0) ;
                Cout, Overflow      : OUT       STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
        SIGNAL Sum : SIGNED(16 DOWNTO 0) ;
BEGIN
        Sum <= ('0' & X) + Y + Cin ;
        S <= Sum(15 DOWNTO 0) ;
        Cout <= Sum(16) ;
        Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

# Addition of Signed Numbers (3)

```
ENTITY adder16 IS
      PORT (    X, Y      : IN      INTEGER RANGE -32768 TO 32767 ;
                S         : OUT     INTEGER RANGE -32768 TO 32767 ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
      S <= X + Y ;
END Behavior ;
```

# Addition of Unsigned Numbers

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY adder16 IS
    PORT (  Cin          : IN   STD_LOGIC ;
            X, Y         : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
            S            : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
            Cout         : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(15 DOWNTO 0) ;
    Cout <= Sum(16) ;
END Behavior ;
```

# Multiplication of signed and unsigned numbers (1)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all ;

entity multiply is
        port(
                a : in STD_LOGIC_VECTOR(15 downto 0);
                b : in STD_LOGIC_VECTOR(7 downto 0);
                cu : out STD_LOGIC_VECTOR(11 downto 0);
                cs : out STD_LOGIC_VECTOR(11 downto 0)
            );
end multiply;

architecture dataflow of multiply is

SIGNAL sa: SIGNED(15 downto 0);
SIGNAL sb: SIGNED(7 downto 0);
SIGNAL sres: SIGNED(23 downto 0);
SIGNAL sc: SIGNED(11 downto 0);

SIGNAL ua: UNSIGNED(15 downto 0);
SIGNAL ub: UNSIGNED(7 downto 0);
SIGNAL ures: UNSIGNED(23 downto 0);
SIGNAL uc: UNSIGNED(11 downto 0);
```

# Multiplication of signed and unsigned numbers (2)

```
begin

-- signed multiplication
     sa <= SIGNED(a);
     sb <= SIGNED(b);

     sres <= sa * sb;
     sc <= sres(11 downto 0);
     cs <= STD_LOGIC_VECTOR(sc);

-- unsigned multiplication
     ua <= UNSIGNED(a);
     ub <= UNSIGNED(b);

     ures <= ua * ub;
     uc <= ures(11 downto 0);
     cu <= STD_LOGIC_VECTOR(uc);

end dataflow;
```


# *Combinational Logic Synthesis for Intermediates*

# Describing combinational logic using processes

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY dec2to4 IS
        PORT (    w      : IN        STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                  En     : IN        STD_LOGIC ;
                  y      : OUT       STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
        PROCESS ( w, En )
        BEGIN
                IF En = '1' THEN
                    CASE w IS
                            WHEN "00" =>            y <= "1000" ;
                            WHEN "01" =>            y <= "0100" ;
                            WHEN "10" =>            y <= "0010" ;
                            WHEN OTHERS =>          y <= "0001" ;
                    END CASE ;
                ELSE
                    y <= "0000" ;
                END IF ;
        END PROCESS ;
END Behavior ;
```

# Describing combinational logic using processes

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY seg7 IS
        PORT (   bcd     : IN        STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 leds    : OUT       STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;
ARCHITECTURE Behavior OF seg7 IS
BEGIN
        PROCESS ( bcd )
        BEGIN
                CASE bcd IS                                 --        abcdefg
                    WHEN "0000"   => leds           <=     "1111110" ;
                    WHEN "0001"   => leds           <=     "0110000" ;
                    WHEN "0010"   => leds           <=     "1101101" ;
                    WHEN "0011"   => leds           <=     "1111001" ;
                    WHEN "0100"   => leds           <=     "0110011" ;
                    WHEN "0101"   => leds           <=     "1011011" ;
                    WHEN "0110"   => leds           <=     "1011111" ;
                    WHEN "0111"   => leds           <=     "1110000" ;
                    WHEN "1000"   => leds           <=      "1111111" ;
                    WHEN "1001"   => leds           <=      "1110011" ;
                    WHEN OTHERS         => leds     <=      "-------" ;
                END CASE ;
        END PROCESS ;
END Behavior ;
```

# Describing combinational logic using processes

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY compare1 IS
      PORT (   A, B     : IN      STD_LOGIC ;
               AeqB     : OUT     STD_LOGIC ) ;
END compare1 ;

ARCHITECTURE Behavior OF compare1 IS
BEGIN
      PROCESS ( A, B )
      BEGIN
            AeqB <= '0' ;
            IF A = B THEN
                  AeqB <= '1' ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

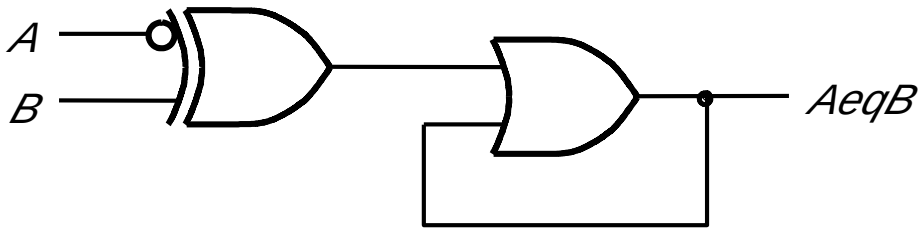
# Incorrect code for combinational logic - Implied latch (1)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
      PORT ( A, B     : IN      STD_LOGIC ;
                AeqB  : OUT     STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
      PROCESS ( A, B )
      BEGIN
            IF A = B THEN
                  AeqB <= '1' ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

# Incorrect code for combinational logic - Implied latch (2)



# Describing combinational logic using processes

## Rules that need to be followed:

1. *All inputs to the combinational circuit should be included in the sensitivity list*
2. *No other signals should be included in the sensitivity list*
3. *None of the statements within the process should be sensitive to rising or falling edges*
4. *All possible cases need to be covered in the internal* ***IF*** *and* ***CASE*** *statements in order to avoid implied latches*

# Covering all cases in the IF statement

***Using ELSE***

```
IF A = B THEN
        AeqB <= '1' ;
ELSE
        AeqB <= '0' ;
```

***Using default values***

```
AeqB <= '0' ;
IF A = B THEN
        AeqB <= '1' ;
```

# Covering all cases in the CASE statement

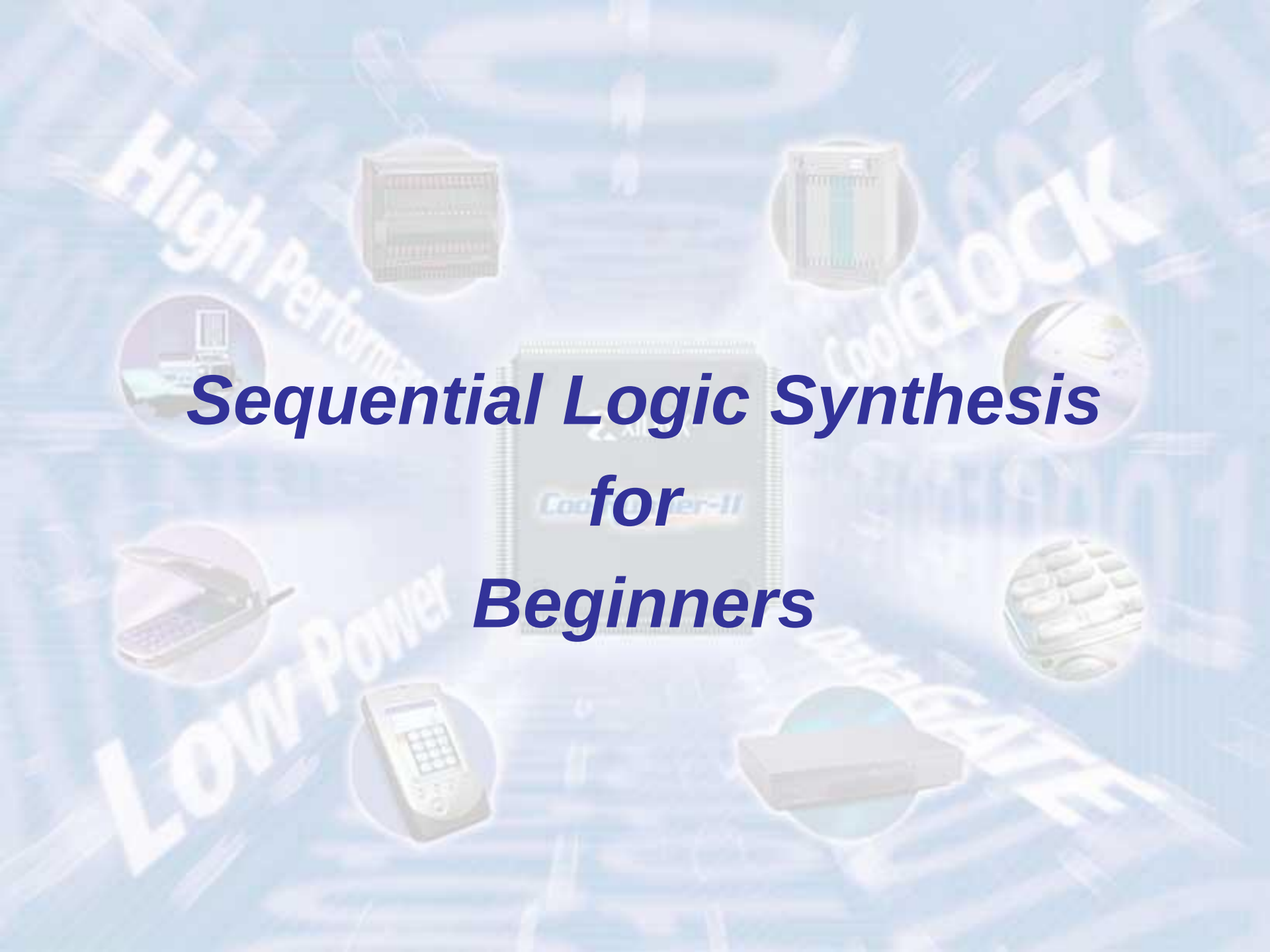
**Using WHEN OTHERS**

```
CASE y IS
    WHEN S1 => Z <= "10";
    WHEN S2 => Z <= "01";
    WHEN OTHERS => Z <= "00";
END CASE;
```

```
CASE y IS
    WHEN S1 => Z <= "10";
    WHEN S2 => Z <= "01";
    WHEN S3 => Z <= "00";
    WHEN OTHERS => Z <= „--";
END CASE;
```

**Using default values**

```
Z <= "00";
CASE y IS
    WHEN S1 => Z <= "10";
    WHEN S2 => Z <= "10";
END CASE;
```




# *Sequential Logic Synthesis for Beginners*

# *For Beginners*

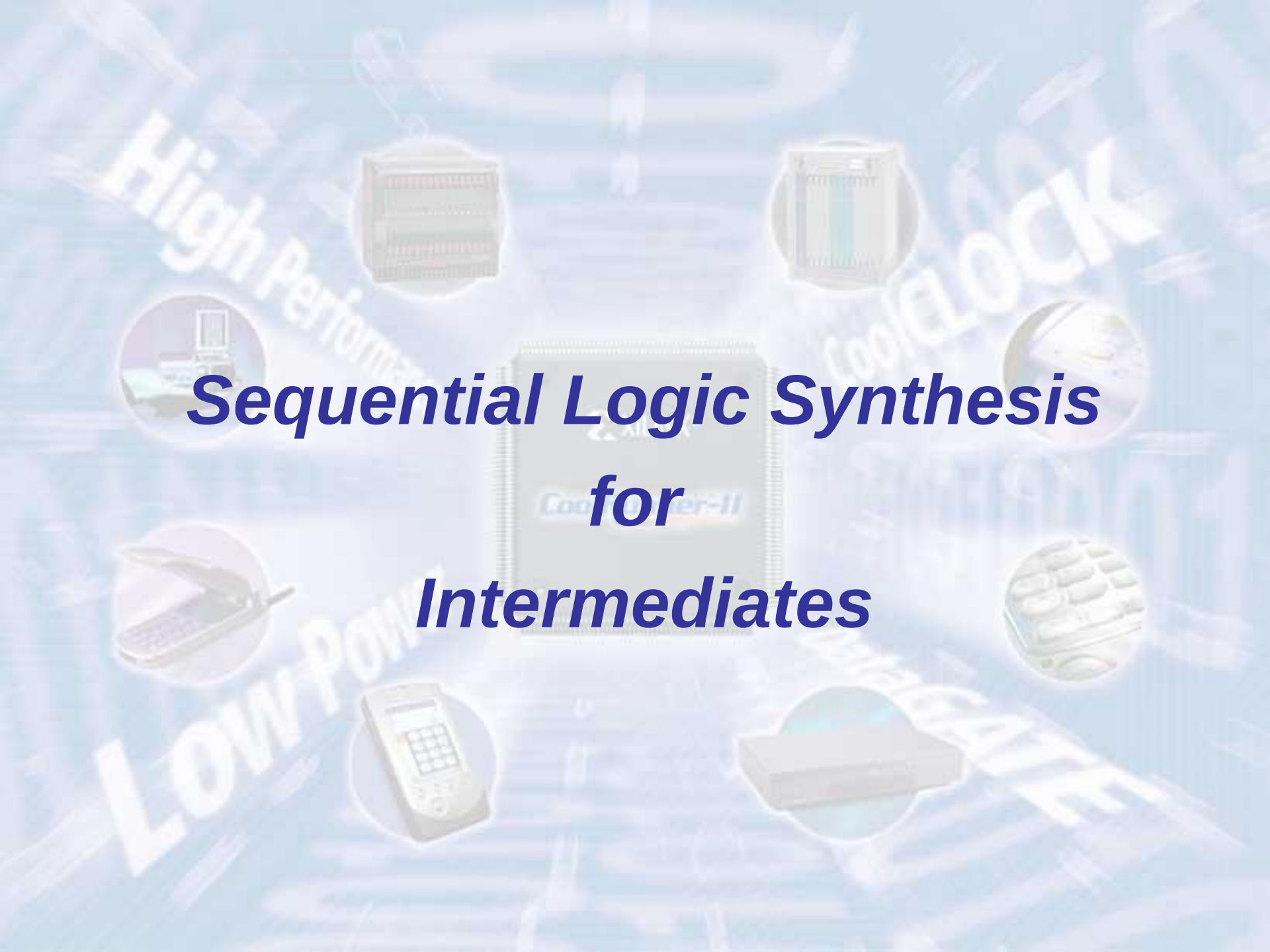
*Use processes with very simple structure only to describe*

- *registers*
- *shift registers*
- *counters*
- *state machines.*

*Use examples discussed in class as a template.*

*Create **generic** entities for registers, shift registers, and counters, and instantiate the corresponding components in a higher level circuit using GENERIC MAP PORT MAP.*

*Supplement sequential components with combinational logic described using concurrent statements.*




# *Sequential Logic Synthesis for Intermediates*

# *For Intermmediates*

1. *Use Processes with IF and CASE statements only. Do not use LOOPS or VARIABLES.*
2. *Sensitivity list of the PROCESS should include **only** signals that can by themsleves change the outputs of the sequential circuit (typically, clock and asynchronous set or reset)*
3. *Do not use PROCESSes without sensitivity list*

   *(they can be synthesizable, but make simulation inefficient)*

# For Intermmediates (2)

*Given a single signal, the assignments to this signal should only be made within a single process block in order to avoid possible conflicts in assigning values to this signal.*

```
Process 1: PROCESS (a, b)
BEGIN
   y <= a AND b;
END PROCESS;

Process 2: PROCESS (a, b)
BEGIN
   y <= a OR b;
END PROCESS;
```