Outline

• Behavioral Design Style
• Anatomy of a Process
• Combinational vs. Sequential Logic
• Registers
• Counters
• Shift Registers
• Register with parallel load
• Examples
Reading

Required

• P. Chu, *FPGA Prototyping by VHDL Examples*  
  *Chapter 4, Regular Sequential Circuit*

Recommended

• S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*  
  *Chapter 7, Flip-Flops, Registers, Counters, and a Simple Processor*
Review of Previous Classes - 1

Data-Flow VHDL

Concurrent Statements

- concurrent signal assignment

```vhdl
target_signal <= expression;
```

- conditional concurrent signal assignment (when-else)

- selected concurrent signal assignment (with-select-when)
"when-else" should be used when:
1) there is only one condition (and thus, only one else), such as in the 2-to-1 MUX
2) conditions are independent of each other (e.g., they test values of different signals)
3) conditions reflect priority (as in priority encoder); one with the highest priority need to be tested first.

"with-select-when" should be used when there is
1) more than one condition
2) conditions are closely related to each other (e.g., represent different ranges of values of the same signal)
3) all conditions have the same priority (as in the 4-to-1 MUX).
Combinational-Circuit Building Blocks

• Commonly used combinational building blocks in design of large circuits:
  – Fixed Shifters & Rotators
  – Adders
  – Multiplexers
  – Decoders and Encoders
  – Multipliers
  – Comparators
  – Buffers
  – ROM
VHDL Description Styles

- **dataflow**
  - Concurrent statements

- **structural**
  - Components and interconnects

- **behavioral**
  - Sequential statements
    - Registers
    - Shift registers
    - Counters
    - State machines

synthesizable
Behavioral Design Style: Registers & Counters

• Processes Describe Sequential Behavior
• Processes in VHDL Are Very Powerful Statements
  ▫ Allow to define an arbitrary behavior that may be difficult to represent by a real circuit
  ▫ Not every process can be synthesized
• Use Processes with Caution in the Code to Be Synthesized
• Use Processes Freely in Testbenches
Anatomy of a Process
PROCESS with a SENSITIVITY LIST

• List of signals to which the process is sensitive.
• Whenever there is an event on any of the signals in the sensitivity list, the process fires.
• Every time the process fires, it will run in its entirety.
Example

priority: PROCESS (clk)
BEGIN
   IF w(3) = '1' THEN
      y <= "11" ;
   ELSIF w(2) = '1' THEN
      y <= "10" ;
   ELSIF w(1) = c THEN
      y <= a and b;
   ELSE
      z <= "00" ;
   END IF ;
END PROCESS ;
Combinational vs. Sequential Logic

Combinational Logic Circuit

Output = \( f(\text{In}) \)

Sequential Logic Circuit

Output = \( f(\text{In}, \text{Previous In}) \)
Sequencing Elements

- **Latch**: Level sensitive
- **Flip-flop**: edge triggered

- Timing Diagrams
  - Transparent
  - Opaque
  - Edge-trigger
D latch
D flip-flop
Example: Draw $Q_{\text{latch}}$ and $Q_{\text{ff}}$
D latch

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY latch IS
  PORT ( D, Clock : IN STD_LOGIC;
         Q      : OUT STD_LOGIC);
END latch;
D flip-flop

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
    PORT ( D, Clock : IN STD_LOGIC;
           Q       : OUT STD_LOGIC);
END flipflop;
D flip-flop

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
  PORT ( D, Clock : IN STD_LOGIC ;
         Q : OUT STD_LOGIC);
END flipflop ;
Asynchronous vs. Synchronous

• In the IF loop, asynchronous items are
  ▫ **Before** the rising_edge(Clock) statement
• In the IF loop, synchronous items are
  ▫ **After** the rising_edge(Clock) statement
Reset

- Force output low when reset asserted
- Synchronous vs. asynchronous

![Diagram of Synchronous Reset]

![Diagram of Asynchronous Reset]
D flip-flop with asynchronous reset

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop_ar IS
    PORT ( D, Reset, Clock : IN STD_LOGIC;
           Q          : OUT STD_LOGIC);
END flipflop_ar;
D flip-flop with synchronous reset

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY flipflop_sr IS
    PORT ( D, Reset, Clock : IN STD_LOGIC;
           Q : OUT STD_LOGIC);
END flipflop_sr;
8-bit register with asynchronous reset

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg8 IS
    PORT ( D : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        Reset, Clock : IN STD_LOGIC ;
        Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg8 ;
N-bit register with asynchronous reset

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
    GENERIC ( N : INTEGER := 16 ) ;
    PORT ( D : IN STD_LOGIC_VECTOR( N-1 DOWNTO 0 ) ;
           Reset, Clock : IN STD_LOGIC ;
           Q : OUT STD_LOGIC_VECTOR( N-1 DOWNTO 0 ) ) ;
END regn ;

ARCHITECTURE behavioral OF regn IS

END behavioral ;
A word on generics

• Generics are typically **integer** values

• Generics are given a default value

• Generics are very useful when instantiating an often-used component
  ▫ Need a 32-bit register in one place, and 16-bit register in another
  ▫ Can use the same generic code, just configure them differently
Use of OTHERS

OTHERS stand for any index value that has not been previously mentioned.
Enable

Symbol

Multiplexer Design

Clock Gating Design

D → Q

\(\phi\)

D → Q

\(\phi\)

D → Q

\(\phi\)

D → Q
N-bit register with enable

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
  GENERIC (N : INTEGER := 8);
  PORT (D : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
    Enable, Clock : IN STD_LOGIC;
    Q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0))
END regne;

ARCHITECTURE behavioral OF regne IS

END behavioral;
Example:
Implementing two registers in a single process
Implementing two registers in a single process
Counters

• Counters are a specific type of sequential circuit

• The state serves as the “output” (Moore)

• A counter that follows the binary number sequence is called a binary counter
  ▫ n-bit binary counter: n flip-flops, count in binary from 0 to $2^n-1$

• Synchronous Counters:
  ▫ A common clock signal is connected to the C input of each flip-flop
Synchronous Binary Up Counter

- The output value increases by one on each clock cycle
- After the largest value, the output “wraps around” back to 0
- Using two bits, we’d get something like this:
Synchronous Binary Up Counter
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY upcount IS
  PORT ( Reset, Clock : IN STD_LOGIC;
         Q : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) );
END upcount;

2-bit up-counter with synchronous reset
4-bit up-counter with asynchronous reset (1)

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

ENTITY upcount_ar IS
   PORT ( Clock, Reset, Enable : IN STD_LOGIC ;
          Q                   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END upcount_ar ;
ARCHITECTURE behavioral OF upcount_ar IS

END behavioral ;
Shift register - internal structure
Video
Adding a parallel load operation

<table>
<thead>
<tr>
<th>LD</th>
<th>$Q(t+1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$Q(t)$</td>
</tr>
<tr>
<td>1</td>
<td>$D_3-D_0$</td>
</tr>
</tbody>
</table>
Register with parallel load

- When $LD = 0$, the flip-flop inputs are $Q_3-Q_0$, so each flip-flop just keeps its current value.
- When $LD = 1$, the flip-flop inputs are $D_3-D_0$, and this new value is “loaded” into the register.
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY shift4 IS
    PORT ( D : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Enable : IN STD_LOGIC ;
           Load : IN STD_LOGIC ;
           Sin : IN STD_LOGIC ;
           Clock : IN STD_LOGIC ;
           Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END shift4 ;
4-bit shift register with parallel load (2)

ARCHITECTURE behavioral OF shift4 IS

END behavioral ;
N-bit shift register with parallel load (1)

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY shiftn IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( D : IN STD_LOGIC_VECTOR(\(N-1\) DOWNTO 0) ;
        Enable : IN    STD_LOGIC ;
        Load   : IN    STD_LOGIC ;
        Sin    : IN    STD_LOGIC ;
        Clock  : IN    STD_LOGIC ;
        Q      : OUT   STD_LOGIC_VECTOR(\(N-1\) DOWNTO 0) ) ;
    END shiftn ;

    

\[ \text{N-bit shift register with parallel load (1)} \]
N-bit shift register with parallel load (2)

ARCHITECTURE behavioral OF shiftn IS

END behavior al;
N-bit register with enable

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( D : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          Enable, Clock : IN STD_LOGIC ;
          Q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regne ;
Constants

Syntax:

CONSTANT name : type := value;

Examples:
Constants - features
Example of package

library ieee;
use ieee.std_logic_1164.all;

end alu_pkg;
Using objects from a package

library ieee;
use ieee.std_logic_1164.all;

library work;
use work.alu_pkg.all;

entity alu_comb is
          ..........
Mixed Style Modeling
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.prng_pkg.all;

ENTITY PRNG IS
    PORT(
        Coeff                  : in  std_logic_vector(4 downto 0);
        Load_Coeff             : in  std_logic;
        Seed                   : in  std_logic_vector(4 downto 0);
        Init_Run               : in  std_logic;
        Clk                    : in  std_logic;
        Current_State          : out std_logic_vector(4 downto 0))
    END PRNG;

ARCHITECTURE mixed OF PRNG is
    signal Ands             : std_logic_vector(4 downto 0);
    signal Sin              : std_logic;
    signal Coeff_Q          : std_logic_vector(4 downto 0);
    signal Shift5_Q         : std_logic_vector(4 downto 0);
BEGIN

-- Data Flow
Sin <= Ands(0) XOR Ands(1) XOR Ands(2) XOR Ands(3) XOR Ands(4);
Current_State <= Shift5_Q;
Ands <= Coeff_Q AND Shift5_Q;

-- Behavioral
Coeff_Reg: PROCESS(Clk)
BEGIN
  IF rising_edge(Clk) THEN
    IF Load_Coeff = '1' THEN
      Coeff_Q <= Coeff;
    END IF;
  END IF;
END PROCESS;

-- Structural
Shift5_Reg : ENTITY work.Shift5(behavioral) PORT MAP ( D => Seed,
        Load  => Init_Run,
        Sin    => Sin,
        Clock => Clk,
        Q     => Shift5_Q);
END mixed;
Sequential Logic Synthesis 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

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 themselves 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 Intermediates (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;
Non-synthesizable VHDL

Delays
Non-synthesizable VHDL

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.**
Next Class

- Datapath vs. Controller
- Finite State Machines
- State Diagrams
- Algorithmic State Machine (ASM) Charts

Required reading
- P. Chu, *FPGA Prototyping by VHDL Examples*  
  Chapter 5, FSM

Recommended reading
- S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*  
  Chapter 8, Synchronous Sequential Circuits  
  Sections 8.1-8.5  
  Section 8.10, Algorithmic State Machine (ASM) Charts