Lecture Goals

Be able to write basic VHDL descriptions for simulation and synthesis. Understand the following:
- Entities and Architectures
- Dataflow (Concurrent) VHDL
- Sequential VHDL
- Structural VHDL
- Constants, Strings, and Types
- Subprograms (Functions and Procedures)
- Libraries and Packages
- Design Specifics

Assignment

- Exemplar HDL Synthesis Guide - Chapters 2 and 3.

VHDL Abstraction Levels

- **Functional:**
  Algorithms without timing.
- **Behavioral:**
  Algorithms (Behavior) with timing. Register level implementation not defined.
- **RTL (Register Transfer Level):**
  Register level implementation defined in terms of registers, datapaths, state machines. Where operations are performed, information stored and when register transfers occur occur in terms of clock cycles all determined.
- **Logic or Gate Level:**
  Implementation described in terms of logic components such as gates and flip-flops and Boolean equations.
**INTRODUCTION**

**VHDL Constructs**

- **Concurrent or Dataflow:**
  In hardware, operations occur in parallel or “concurrently.” Concurrent VHDL consists of a collection of statements and processes that execute concurrently.

- **Sequential:**
  It is difficult to describe complex hardware actions in a single concurrent statement. Thus, processes composed of sequences of statements are employed. Sequential VHDL consists of the sequences of statements within processes.

- **Structural:**
  Structural VHDL is analogous to logic diagrams or netlists. Structural VHDL describes interconnections of components (entities).

---

**ENTITIES AND ARCHITECTURES**

- **Entity:**
  The entity is the primary hardware abstraction in VHDL and represents a part of a hardware design that has well-defined inputs and outputs and a well-defined function.
  
  The entity declaration gives among other things, the entity name and the inputs and outputs. In this sense, it is analogous to a symbol in a block diagram.

- **Architecture Body:**
  An architecture specifies the relationships between the inputs and outputs of a design entity. An architecture may be a mixture of structural, concurrent and sequential VHDL. A given entity may have multiple different architectures.

- **Example 1 – An Entity and an Architecture:**
  (Note: VHDL is case-insensitive: AND aNd, and are equivalent.)
library ieee; -- These instantiations of a library and a package are tied
use ieee.std_logic_1164.all; -- to the following entity. If there is more than one entity
library ieee; -- libraries and packages used must be given with each.

entity decoder_3_to_8 is -- Declaration of entity named decoder_3_to_8
port (A : in std_logic_vector(2 downto 0); -- 3-bit input A of type std_logic.
    D : out std_logic_vector(7 downto 0));  -- 8-bit output D of type std_logic.
end decoder_3_to_8; -- End of entity declaration.

architecture functional of decoder_3_to_8 is -- Architecture named functional
begin -- for entity decoder_3_to_8
    D <= B"00000001" when A = B"000" -- Concurrent when else statement for
        else B"00000010" when A = B"001" -- evaluation of output D.
        else B"00000100" when A = B"010"
        else B"00001000" when A = B"011"
        else B"00010000" when A = B"100"
        else B"00100000" when A = B"101"
        else B"01000000" when A = B"110"
        else B"10000000" when A = B"111"
        else B"xxxxxxxx" when others; -- For 9 values of std_logic except 0 and 1.
end functional; -- End of entity declaration.

• Signals can be viewed as “wires.”
• Signals are concurrent and sequential objects; variables are sequential objects.
• Signal declarations have already appeared within the port declarations for
  entities.
  • Examples - Signal Declaration:

  signal a, b : std_logic;
signal widget : std_logic_vector (0 to 7)
signal x : in std_logic_vector (2 downto 0);
signal z : out std_logic;
**ECE 554**

**CONCURRENT VHDL**

**Signal Assignment**

- Uses signal assignment operator: `<=`

- A signal is assigned its value after a delay, whether real or a delta time.

- **Examples - Signal Assignment:**

  
  \[
  z := a \text{ or } b; \quad \text{-- } z \text{ assigned after infinitesimal delta time}
  \]

  \[
  z := a \text{ nand } b \text{ after } 10 \text{ ns}; \quad \text{-- } z \text{ assigned after inertial delay of } 10 \text{ ns}
  \]

  \[
  z := \text{reject } 5 \text{ ns } \text{inertial } (a \text{ nand } b) \text{ after } 10 \text{ ns}, (a \text{ or } b) \text{ after } 20 \text{ ns};
  \]

  \[
  \quad \text{-- } z \text{ assigned after inertial delay of } 10 \text{ ns and again after inertial delay}
  \]

  \[
  \quad \text{-- of } 20 \text{ ns with pulses on input of } 5 \text{ ns or less rejected.}
  \]

---

**ECE 554**

**CONCURRENT VHDL**

**When Else Statement**

- `widget <= transport ("000" & x & a & b) after 10 ns;` -- z assigned after transport delay of 10 ns; & is concatenation operator.
- Provides conditional assignment.
- See decoder_3_to_8 example.
- **Additional Example - When Else:**

  \[
  \text{library ieee;}
  \]

  \[
  \text{use ieee.std_logic_1164.all;}
  \]

  \[
  \text{entity function_eval is}
  \]

  \[
  \text{port} \(X, Y: \text{in std_logic_vector(15 downto 0)};\)
  \]

  \[
  \quad S : \text{in std_logic_vector(1 downto 0)}; \text{T: in std_logic;}
  \]

  \[
  \quad F : \text{out std_logic (15 downto 0);}
  \]

  \[
  \text{end function_eval;}
  \]

  \[
  \text{architecture when_else of function_eval is}
  \]

  \[
  \text{begin}
  \]

  \[
  \quad F := X \text{ or } Y \text{ when } S = "00"
  \]

  \[
  \quad \text{else } X \text{ xor } Y \text{ when } S = "01"
  \]

  \[
  \quad \text{else } X \text{ and } Y \text{ when } S = "10"
  \]

  \[
  \quad \text{else Y when } T = 1; \quad \text{-- Note that statements are evaluated in order.}
  \]
**CONCURRENT VHDL**

**With Select Statement**

```vhdl
else X;
end when_else;

• Provides conditional assignment.

• Example - With Select:

  library ieee;
  use ieee.std_logic_1164.all;
  entity function_eval is
  port (X, Y : in std_logic_vector(15 downto 0);
        S : in std_logic_vector (1 downto 0);
        F : out std_logic (15 downto 0);
  end function_eval;
  architecture with_select of function_eval is -- Note different architecture
  begin -- for same entity.
    with S select
      F <= X or Y when "00" -- so less flexible than when else.
      X xor Y when "01"
      X and Y when "10"
      not X when "11"
  end with_select;
```

**CONCURRENT VHDL**

**Block Statement**

```vhdl
“xxxxxxxxxxxxxxxxxxx” when others;
end with_select;

• The block statement is used to conveniently partition VHDL code for readability and for partitioning among designers.

• Allows use of a sort of hierarchy without the associated multiple entities that can reduce the synthesis efficiency achieved with 500 to 6,000 gate entities.

• The block statement also allows local signal and variable scoping for those signals or variables declared within a block.

• Example - Use of Block Statement:

  library ieee;
  use ieee.std_logic_1164.all;
  use ieee_unsigned_arith.all;
  entity function_eval is
  port (X, Y : in std_logic_vector(15 downto 0);
        S : in std_logic_vector (1 downto 0); T : in std_logic;
        F, G : out std_logic (15 downto 0);
  end function_eval;
```
A block statement may contain a **guard expression** which controls the execution of assignment statements in the block containing the keyword **guarded**.

This mechanism can be used to implement latches and registers within concurrent statements.

**Examples – Register Implementation:**
Assume **entity** function_eval as given previously.

**Example 1:**
```
register_simple: block (clk'event and clk = '1')
begin
    F <= guarded X;
end block;
```
Example 2:
register_logical: block (clk'event and clk = '1')
begin
    F <= guarded X or Y when S = "00"
    else X xor Y when S = "01"
    else X and Y when S = "10"
    else Y when T = '1';
    else X;
end block;

Example 3:
register_arithmetic: block (clk'event and clk = '1')
with S select
    F <= guarded X + Y when "00"
    X – Y when "01"
    X * Y when "10"
    – X when = "11"

“xxxxxxxxxxxxxxxxxxxx”when others;
end block;

• Used within processes, functions and procedures.
• Statements are executed sequentially as in the typical programming language.
• Processes execute concurrently with each other. In this sense, a complete process behaves like a concurrent statement.
• In a process, variables can be used as well as signals. Variables are evaluated without delay, i.e., immediately and the result of the evaluation is available to subsequent statements in the process.
• Variables are declared and assigned only within sequential VHDL.
• Variables may be equated to signals and signals to variables as long as they are of the same type.
A process is a sequence of sequentially-executed statements. In a process, the statements are ordered.

A sensitivity list governs the activation of the process. Process executes only when a signal on the sensitivity list changes.

If process represents a combinational circuit, all inputs must be on the sensitivity list.

If process represents the next state part of a synchronous state machine, the clock and asynchronous reset signal typically are on the sensitivity list.

Example – Process:

```
library ieee;
use ieee.std_logic_1164.all;
entity multiplexer_4_to_1 is
  port (S : in std_logic_vector(1 downto 0);
         D : in std_logic_vector(3 downto 0);
         Y : out std_logic);
end multiplexer_4_to_1;
architecture behavioral of multiplexer_4_to_1 is
begin
  pick: process (S, D) is
    begin
      case S is
        when B"00" => Y <= D(0);  -- Use of subarrays D(i) of array D.
        when B"01" => Y <= D(1);
        when B"10" => Y <= D(2);
        when B"11" => Y <= D(3);
        when others => Y <= 'X';
      end case;
    end process pick;
end behavioral;
```
Variable Declaration

- Ordinarily declared within processes, functions, or procedures to store values temporarily.

- In VHDL 93, the shared variable introduced; can be used globally within multiple processes. Apparently has determinism problems; suggest you avoid.

- Examples - Variable Declaration:

```vhdl
variable a, b : std_logic;
variable widget, fidget : std_logic_vector (0 to 7)
variable x : std_logic_vector (2 downto 0);
```

Variable Assignment

- Can only be done within processes;
- Uses the variable assignment operator :=
- Assignments are instantaneous so that subsequent statements use the new values generated by the prior statements.

- Example - Variable Assignments:

```vhdl
a := x and y;
b := (not x) and w;
z := a or b;
```

- If the above statements are executed in sequence, the Boolean function implemented is:

```
z = xy + \overline{x}w
```
• Signal assignments can be performed within processes.

• Example – Combined Variable and Signal Assignment:

  combinational:process (x, y, w) is
    a := x and y;
    b := (not x) and w;
    z <= a or b;
  end process;
Case Statement

\[ z = d; \quad \text{-- Occurs for } x = 0, y = 0 \text{ and } z = 1 \]
\[ \text{end if}; \quad \text{-- and for all of the 9 std_logic values} \]
\[ \text{-- except 0 and 1}. \]

- Not evaluated in order; synthesizes more like regular “flat decoder.”
- Example – Case:

```vhdl
architecture behavioral of multiplexer_4_to_1 is
    begin
        pick: process (S, D) is
            begin
                case S is
                    when B"00" => Y <= D(0);  -- Use of subarrays D(i) of
                    when B"01" => Y <= D(1); -- array D.
                    when B"10" => Y <= D(2);
                    when B"11" => Y <= D(3);
                    when others => Y <= 'X';
                end case;
            end process pick;
    end behavioral;
```

Loop Statements

```vhdl
end process pick;
end behavioral;

- Example – For loop:

```vhdl
process (x, y, z) is
    variable c : std_logic_vector (4 downto 0);
    variable next_c: std_logic;
        next_c = '0';
    begin
        for i in 0 to 3 loop
            c(i) := next_c;
            next_c := ((x(i) xor y(i)) and c(i)) or (x(i) and y(i));
            z(i) <= x xor y xor c(i);
        end loop;
    end process;
```
• Example – While loop:
  
  ```vhdl
  process (x, y, z) is
  variable c : std_logic_vector (3 downto 0);
  variable i : integer range 0 to 4;
  begin
  next_c := '0';
  i := '0';
  while i < 4 loop
    c(i) := next_c;
    next_c := ((x(i) xor y(i)) and c(i)) or (x(i) and y(i));
    z(i) <= x xor y xor c(i);
    i := i + 1;
  end loop;
  end process;
  ```

Wait and Null Statements

Wait Statements

• Example – Wait On:
  ```vhdl
  wait on a, b; -- equivalent to process (a, b) is
  ```

• Example – Wait Until:
  ```vhdl
  wait until a = '1'; -- For execution to continue, a must change value and
  -- its new value must be 1.
  ```

• Example – Wait for:
  ```vhdl
  wait for 10 ns; -- For execution to continue, 10 ns of time must elapse.
  ```

Null Statement

• Permits signal values to remain unchanged for particular conditions:
Components

- Example - Null Statement:
  
  ```vhdl
  when others => null;
  ```

- Structural VHDL is based on an interconnection of components (entities).
- A key element of structural VHDL is the component.
- Example – Structural VHDL – Components:
  
  ```vhdl
  library ieee;
  use ieee.std_logic_1164.all;
  entity adder_4_bit_co is
    port (A, B : in std_logic_vector(3 downto 0);
          S : out std_logic_vector(3 downto 0);
          C0 : in std_logic; C4 : out std_logic);
  end entity;
  architecture full_adder_imp of adder_4_bit_co is
    component full_adder
      port (X, Y, Z : in std_logic;
            S, C : out std_logic);
    end component;
    signal C : std_logic_vector(3 downto 1);
  begin
    fa0: full_adder
      port map(X => A(0), Y => B(0), Z => C0, S => S(0), C => C(1));
    fa1: full_adder
      port map(X => A(1), Y => B(1), Z => C(1), S => S(1), C => C(2));
    fa2: full_adder
      port map(X => A(2), Y => B(2), Z => C(2), S => S(2), C => C(3));
    fa3: full_adder
      port map(X => A(3), Y => B(3), Z => C(3), S => S(3), C => C4);
  end full_adder_imp;
  ```

Components (Continued); Direct Instantiation

- Example – Structural VHDL – Direct Instantiation:
  
  ```vhdl
  library ieee;
  use ieee.std_logic_1164.all;
  entity adder_4_bit_st is
    port (A, B : in std_logic_vector(3 downto 1);
          S, C : out std_logic);
  end entity;
  ```
ECE 554

STRUCTURAL VHDL
Direct Instantiation (Continued)

```vhdl
port (A, B : in std_logic_vector(3 downto 0);
     S : out std_logic_vector(3 downto 0);
     C0 : in std_logic; C4 : out std_logic);
end adder_4_bit_st;
architecture full_adder_implementation of adder_4_bit_st is
  signal C : std_logic_vector(3 downto 1);
begin
  FA0: entity work.full_adder
    port map (X => A(0), Y => B(0), Z => C0, S => S(0), C => C(1));
  FA1: entity work.full_adder
    port map (X => A(1), Y => B(1), Z => C(1), S => S(1), C => C(2));
  FA2: entity work.full_adder
    port map (X => A(2), Y => B(2), Z => C(2), S => S(2), C => C(3));
  FA3: entity work.full_adder
    port map (X => A(3), Y => B(3), Z => C(3), S => S(3), C => C4);
end full_adder_implementation;
```

ECE 554

VHDL CONSTANTS AND STRINGS

Constants
- A **constant** is a class object just as signal and variable; does not change value.
- Constant declaration can be anywhere and of any type.
- **Example – Constant Declaration:**
  ```vhdl```
  ```constant loop_count: integer := 63;```
  ```end```
- **Bit String Assignment**
  - **Example – Single bit:** `a <= '1';`
  - **Example – Multiple bits:** `z <= "00010101" "0001_0101"` **not valid!**

Bit String Literals
- **Binary** `B"11000"` `B"1_1000"` Underlines to ease readability.
- **Octal** `O"30"` Can be used as well with O and X.
- **Hex** `X"18"`
- **Decimal** 24 Allowed only for constants
- **Real** `2.4E+1` Not supported for synthesis.
- **All signals and variables in VHDL are typed.**
Declaration:

```
    type integer is range -2147483647 to 2147483647;
```

- Example integer:

```
    signal a, b : integer;
```

**Physical**

- Declaration:

```
    type time is range -2147483647 to 2147483647
        units

        fs
        ps = 1000 fs;
        ns = 1000 ps;
        ...
        hr = 60 min;
    end units;
```

- Example physical:

```
    z <= a after 10 ns;
```

---

**IEEE 1076, IEEE 1164**

- IEEE 1076 Predefined Types:

```
    type bit is ('0','1');
    type bit_vector is array (integer range <>) of bit;
    type integer is range MININT to MAXINT;
    subtype positive is integer range 1 to MAXINT;
    subtype natural is integer range 0 to MAXINT;
    type boolean is (TRUE, FALSE);
```

- IEEE 1164 Predefined Types:

```
    type std_ulogic is ('U','X','0','1','Z','W','L','H','-');
    type std_ulogic_vector is array (natural range <>) of std_logic;
    subtype std_logic is resolution_func std_logic;
    type std_logic_vector is (natural range <>) of std_logic;
    subtype X01Z is resolution_func std_logic range 'X' to 'Z';
        -- includes X, 0, 1, Z
```
• Subprogram bodies consist of sequential VHDL. Thus only sequential constructs can be used and only variables can be declared. Wait statements are not permitted.
• Local variables may be declared within subprograms, but not local signals.
• Subprograms can be defined in processes, architectures or packages.
• Subprograms can be called concurrently or sequentially as long as types are carefully handled.

Functions
• A function has multiple arguments, all of which are constants or signals of with mode in, and returns a single value.
• Example – Function:
  
  ```vhdl
  function max (a, b : in std_logic_vector) return std_logic_vector is
  begin
    if a > b then
      return a;
    else
      return b;
    end if;
  end;
  ```

• Example – Function Call:
  
  ```vhdl
  comp_res <= max (data_1, data_2)
  ```

Procedures
• A procedure has multiple arguments, which can be constants, variables and signals with any of the three modes, and can return multiple values. If a mode is not specified, it is assumed to be in. If no mode is specified, then in is the default.
### SUBPROGRAMS

Functions and Procedures (Continued)

- **Example – Procedure:**

  ```vhdl
  procedure a_m (signal a, b : in integer;
                 signal avg, max : out integer) is
  begin
    avg <= (a + b)/2;
    if a > b then
      max <= a;
    else
      max <= b;
    end if;
  end a_m;
  ```

- **Example – Procedure Call:**

  ```vhdl
  a_m (data_1, data_2, result_avg, result_max);
  ```

  Since all arguments are signals, can be called concurrently or sequentially.

### LIBRARIES

Components and Packages

- Compiled VHDL entities are stored in a **library** with a logical name.
- These entities can be viewed as components if desired.
- Packages can also be stored in libraries.
- The library or libraries containing components or packages to be used must be specified for each of the entities in a description.

- **Example – Library Specification:**

  ```vhdl
  library widgets;
  ```

  By default, the following libraries are specified for each entity declared:

  ```vhdl
  library work;
  library std;
  ```

  Typically, we will use the std_logic type, hence, for each declared entity:

  ```vhdl
  library ieee;
  ```
• A package contains functions, procedures, types, constants, attributes or components.
• A package consists of a declaration and a body.
• The declaration is similar to multiple entities, but describes instead exportable objects.
• The body is similar to multiple architectures, providing the actual subprograms and component bodies.
• Example – Package Use:
  use widgets.gizmos.all;
This indicates that all of the objects in the package gizmos in library widgets are to be available for use.
• By default, the following package use is specified:
  use std.standard.all;
• Typically, we will use the std_logic type, hence, for each declared entity:
  use ieee.std_logic_1164.all;

• Other packages may be essential to the designs you do to avoid difficult VHDL coding. These include Mentor Graphics packages:
STD_LOGIC_ARITH
QSIM_LOGIC
and Exemplar package:
ex_1164
See Chapter 4 of the HDL Synthesis Guide.
• Issues to be resolved:
Use of ieee.ieee_1164.all versus use of exemplar.ex_1164.all in VHDL for simulation and synthesis; the latter appears to contain more useful objects such as latches and flip-flops.
Packages required for use of Leonardo Module Generation for arithmetic, comparison, SRAM and counter implementations.
Latches and Registers

• Example – Latch – Dataflow:

```vhd
signal D, en, Q : std_logic;
latch: block (en = '1')
begin
    Q <= guarded D;
end block;
```

• Example – Latch – Sequential:

```vhd
signal D, en, Q : std_logic;
process (en, D) is
begin
    if (en = '1') then
        Q <= D;
    end if;
end process;
```

Latches and Registers (Continued)

• Example – Positive Edge-Triggered Flip-Flop – Dataflow:

This example uses an event attribute. An attribute is a derived property of an object. The event attribute is TRUE if a signal changes and FALSE otherwise.

```vhd
signal D, clk, Q: std_logic;
dff: block (clk'event and clk = '1')
begin
    Q <= guarded D; -- What about glbreset and clock enable?
end block;
```

• Example – Positive Edge-Triggered Flip-Flop - Sequential:

```vhd
signal D, clk, Q: std_logic;
process (clk, glbreset) is
begin
    if (glbreset = '1')
        Q <= '0';
    elseif (clk'event and clk = '1') then -- What about clock enable?
        Q <= D;
    end if;
end process;
```
Latches – Unintentional!

- Example – Incomplete Combinational Process:
  ```vhdl
  signal a, b, z : std_logic;
  process (a,b) is
  begin
    if (a = '1')
      z <= '0';
    elseif (b = '1') then
      z <= '1';
    end if;
  end process;
  ```

  For the above, z is defined for a,b = 1x and 01, but is not defined for 00. This causes a latch to be synthesized for z that is enabled for 1x and 01 and disabled for 00!

  - Can also occur for any incomplete sequential process; possibly for when-else and with-select as well.

State Machines

- Example - Typical State Machine: (adapted from Exemplar HDL Synthesis Guide, pp. 3-19 and 3-20.)
  ```vhdl
  entity ras_cas is
    port ( clk, cs, refresh, reset : in std_logic;
           ras, cas, ready : out std_logic);
  end ras_cas;

  architecture sequential of ras_cas is
  -- Define the states of the state machine:
  type state_type is (s0, s1, s2, s3, s4);
  signal present_state, next_state : state_type;
  begin
    state_reg : process (clk, reset) is
    begin
      -- process to update the present state
  ```
if (reset = ‘1’) then
    present_state <= s0;
elsif clk’event and clk = ‘1’ then
    present_state <= next_state;
end if;
end process;

transitions: process (present_state, refresh, cs) is
begin
    -- process to calculate the next state and the outputs
    case present_state is
    when s0 =>
        ras <= ‘1’; cas <= ‘1’; ready <= ‘1’; -- To prevent latch formation,
        -- must assign all outputs and next state for all conditions given.
        if (refresh = ‘1’) then
            next_state <= s3;
    when s1 =>
        ...
    end case;
end process;
end sequential;
To synthesize a state machine, a state encoding must be specified. The default encoding is onehot which uses a flip-flop per state. Otherwise, encoding can be defined by the following attributes:

- **type_encoding_style** (state machine encoding type: binary, gray, one-hot, random)
- **type_encoding** (bit-by-bit manual encoding for the states)
- **logic_type_encoding** (used for other than state encoding).

**Example – Type_Encoding_Style Use:**

```vhd
attribute type_encoding_style of state_type: type is onehot;
```

### Results for type_encoding_styles:

<table>
<thead>
<tr>
<th>States</th>
<th>One-Hot</th>
<th>Binary</th>
<th>Gray</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>000</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>s1</td>
<td>010</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>s2</td>
<td>011</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>s3</td>
<td>111</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

**Example – Type_Encoding**

```vhd
type state_type is (IDLE, ADD, SHIFT, HOLD, MULT);
attribute type_encoding of state_type: type is
("000", "010", "011", "10-", "11-”);
```
**ECE 554**

**DESIGN SPECIFICS**

**Xilinx Modgen Use**

- Special Xilinx features such as SRAM and fast carry logic can be reached only by using the Module Generation (Modgen) capability of Leonardo.
- Modgen supports `+`, `-`, `-1`, `=`, `\=`, `>`, `=>`, `<`, `<=`, `*`, `/`, `**`, `mod`, `rem`, `abs`, `srl`, `sll`, `sra`, `sla`, `rol`, `ror`.
  
  Also, RAMs and counters.
- Modgen can be activated by loading a Modgen library using the GUI or command line:
  
  modgen\_read \textit{modgen\_library}
  
  where in our case \textit{modgen\_library} is xi4e.

---

**ECE 554**

**REFERENCES**


