The primary objective of this course can be summarized as: Learn and practice the design of digital hardware using VHDL. While this course is focused on an ASIC design flow, most of what you learn is equally applicable to design using programmable logic devices such as FPGAs and CPLDs.

The secondary objective of this course is to make sure you can briefly explain characteristics and advantages/disadvantages of ASICs and CMOS based designs.

Definition:

CMOS = Complementary Metal Oxide Semiconductor
Warning!

- This is a digital logic design course not a software design course
- If you have forgotten how to design things like:
  - combinational logic
  - state machines
  - counters
You better start remembering.
Application Specific Integrated Circuit

What is “Application Specific”?
- Specially designed for a particular use, i.e., custom

What is the opposite of ASIC?
- Design using an off-the-shelf FPGA, PLD, microcontroller, or microprocessor - general purpose chips that can be reprogrammed for many tasks

Examples of ASICS could include

A custom signal processor for a cell phone
A USB 2.0 or Firewire bus controller chip
A gigabit network interface chip
An MPEG encoder/decoder
A calculator
A digital watch
A graphics accelerator for a computer or game console

Typically you do an ASIC if you need extremely large quantities (a million or more units), extremely small physical size (as in a watch), extremely high performance (such as high speed network interfaces), or extremely low power (such as in a cell phone or watch)

Some examples of things you probably would not design with an ASIC (because they can be done quicker/easier/cheaper with software running on a microcontroller or microprocessor)

An elevator controller
A pocket poker game
A microwave oven controller
A remote control car or airplane

People who design new microprocessors or new types of FPGAs may be called ASIC designers even though their product is a general purpose chip.
Instructor: Mark Johnson
- Manager, Digital & Systems Labs since 1999
- Ph.D. EE 1998 at Purdue, CAD for low power VLSI
- Taught for year at Rose-Hulman 1998-99
- 11 years total at Boeing and at Thomson Consumer Electronics

TAs: Edwin Tjandranegara (coordinator), Georgios Karakonstantis, Wei-Foong Thoo

Log into WebCT and check the ECE495D site for hours, syllabus, handouts, etc.
On-line resources

Course web page is in WebCT

- lab hours, TA info, handouts
- posting of grades and outcomes
- posting of course announcements
- online discussion groups/question & answer
Syllabus Highlights

- Beware of ABET outcomes & Plagiarism
- Grading: 65% lab, 35% lecture
  - 10% quizzes & in-class exercises
  - 25% exams
  - 25% final project
- Text:
  - Wakerly 3rd Edition (should already have from EE270)
  - Online course notes
  - Other online references
- Recommended References:
  - VHDL for Logic Synthesis – Andrew Rushton
  - VHDL Made Easy - Pellerin & Taylor
  - VHDL Coding and Logic Synthesis with Synopsys – W.F. Lee
  - Student Guide to VHDL, Peter Ashenden
- Pre-requisite knowledge:
  - “Structured” programming skills
  - Combinational & sequential logic design
“Real Programmers” need not apply

Description of “real programmers” adapted (and sanitized) from http://www.baetzler.de/humor/

- Real Programmers don’t write in strongly typed languages. Strong variable typing is for people with weak memories.

- Real Programmers distain structured programming. Structured programming is for compulsive neurotics. They wear neckties and carefully line up sharp pencils on an otherwise clear desk.

- Real Programmers don’t like the team programming concept. Unless, of course, they are the Chief Programmer.

- Real Programmers have no use for managers. Managers are sometimes a necessary evil. Managers are good for dealing with personnel, bean counters, senior planners and other inferiors.

- Real programmers ignore schedules.

- Real Programmers use C since it’s the easiest language to spell.

- Real Programmers only curse at inanimate objects.

Strongly typed means that you can’t mix data types within an expression without explicit type conversions.
“Real Programmers” need not apply

Description of “real programmers” adapted (and sanitized) from http://www.baetzler.de/humor/

- Real Programmers don’t write in strongly typed languages. Strong variable typing is for people with weak memories. VHDL is strongly typed

- Real Programmers distain structured programming. Structured programming is for compulsive neurotics. They wear neckties and carefully line up sharp pencils on an otherwise clear desk. Write very structured code or suffer (or both)

- Real Programmers don’t like the team programming concept. Unless, of course, they are the Chief Programmer. You will be working in teams.

- Real Programmers have no use for managers. Managers are sometimes a necessary evil. Managers are good for dealing with personnel, bean counters, senior planners and other inferiors. I am the manager in this class. I guess this makes me a necessary evil.

- Real programmers ignore schedules. Ignore schedules, get a 0.

- Real Programmers use C since it’s the easiest to spell. VHDL is difficult to spell.

- Real Programmers only curse at inanimate objects. Ok, but watch your language.
<table>
<thead>
<tr>
<th>ASIC design basics</th>
</tr>
</thead>
</table>


Here is a brief summary. See module 8 ASIC Design Process for more detail.

Block Diagram (HDL Designer from Mentor Graphics) – draw a schematic containing a block for each major subsystem of your design and add wires to show how they are connected.

VHDL Code – write VHDL source code describing the required behavior of each subsystem or subcircuit.

Simulation (Modelsim from Mentor Graphics) – debug your code. It is in some ways similar to source code debuggers you may have used (i.e., you can set breakpoints, watch variables, etc) except that you can also display waveforms for the inputs and outputs of your design.

Synthesis (Design Compiler from Synopsys) – takes your source code, figures out what logic gates and storage elements are needed to make a circuit out of it. The output is a netlist, i.e., a list of logic circuits and the connections between them.

Place and Route (SOC Encounter from Cadence) – takes the netlist that came from synthesis and produces a complete silicon layout from it. The layout shows graphically where all of the transistors and wires are supposed to go during manufacturing.

Timing (Pathmill from Synopsys) – identifies timing problems (setup and hold violations) and determines the maximum clock rate for the IC layout that was produced.

Note: there are several additional layout verification steps not described in this figure.

Fabrication (not enough time to do in one semester) – send the layout data and chip packaging specifications to a fabrication company to manufacture the chip.

Real world – after fabrication, you would need to test the chips.
CAVEAT EMPTOR

Not all VHDL will work with synthesis

- Much VHDL syntax only used for...
  - Simulation
  - Test benches (for design verification)
- A functional simulation proves little
- Poorly structured designs...
  - May not translate well, if at all, to hardware
  - Especially bad for CPLD’s & FPGA’s

Only a small subset of VHDL syntax can actually be used for circuit design. A large portion of the language is targeted exclusively for simulation modeling and design verification.

Even within the allowable subset of VHDL syntax, there are severe restrictions on how you can write your code and still expect it to translate into hardware in a predictable manner.

In this class, we will focus almost exclusively on the type of code needed to create a circuit design. We will cover a small amount of additional syntax to make it easier to write test benches for verifying your design.

Definition:

Test Bench: code that is written specifically for the purpose of generating test waveforms for input to a simulation of your VHDL designs. Test Benches can also be used to automatically check outputs of your designs. Later in the semester, your teaching assistants will use specialized test benches to evaluate your designs.
Advice

Some tips to improve chances of synthesis

- Think of each block of code as a circuit rather than as code.
- VHDL is not C! It is a circuit description language.
- Until proficient, stick with styles and syntax from this course.
- Major difference from sequential programming languages,
  - Each concurrent statement executes in parallel to every other concurrent statement.
  - Order of concurrent statements has nothing to do with order of execution!

1. Before writing VHDL code, think very hard about how you would build the circuit first. Try to write your code in a way that describes, as directly as possible, the behavior of each part of the circuit.

2. Think about how digital circuits work – all of the circuits are operating simultaneously. When an input signal changes, the change propagates through various stages of the circuit and finally ends up at an output. Your VHDL code works in a similar way – as it should since you are using the VHDL to describe a circuit.
Before writing VHDL code for a circuit, try to draw a high level (RTL) schematic – i.e., a schematic that consists of registers and combinational logic. Wherever possible for the combinational logic, think in terms of higher level combinational functions such as multiplexers, decoders, adders, etc.

Definitions:

RTL = Register Transfer Level – a fancy way of saying that a sequential digital design can be described entirely in terms of clocked registers and unclocked transformations on data going from one register to another (or back to itself as in a state machine)

Combinational logic is logic that consists only of logic gates such as AND, OR, NAND, NOR, INV, and that has no feedback paths. Combinational logic is used to implement the unclocked transformations on data going from one register to another (or back to itself as in a state machine)

Register – a group of one or more edge-triggered flip-flops used to store data such as the value of a counter, the state of a state machine, etc.
Basic VHDL syntax

Attention! The vast majority of the VHDL syntax you learn will be for the purpose of specifying combinational logic or wiring.

When memory is needed, you will write separate (very simple) blocks of code for it (we will get to this later).
What is Combinational logic?

- Logic that can be built entirely out of AND, OR, and Inverter circuits
  - with NO feedback paths
  - with NO memory elements
- Given enough time/space, each output of combinational logic can be written:
  - as a boolean equation containing only the names of the circuit inputs
- There are NO Clock Signals
Basic Structure of VHDL

Two basic units of code:
- entities and architectures.
- Similar to relationship between symbol and schematic
- The entity declaration (analogous to symbol) represents a subcircuit and identifies the inputs and outputs.
- The architecture (analogous to a schematic) determines the actual logical function of the symbol.
- Also similar to function prototype/function body in C

Consider the following design
- Is something wrong with it?
- Draw a schematic of it

Regarding the C code analog –

An entity declaration is roughly equivalent to a function header in C. A function header specifies the name of the function, the names of all inputs and outputs, and the data types of all inputs and outputs. An entity declaration specifies the name of the VHDL component to be created, the names of all inputs and outputs, and the data types of all inputs and outputs.

An architecture is roughly equivalent to a function body in C. A function body contains C code that specifies the behavior of the function. An architecture contains the VHDL code that specifies the behavior of a digital circuit.
A SAMPLE COMBINATORIAL CIRCUIT:  
2 input NAND/NOR

```vhdl
ENTITY nand_nor IS  
    PORT (Sel, A, B: IN std_logic;  
    Y: OUT std_logic);  
END nand_nor;

ARCHITECTURE dataflow OF nand_nor IS  
    SIGNAL Y1, Y2, Y3, Y4 :std_logic;  
    BEGIN  
        Y <= Y3 OR Y4;  
        Y1 <= A AND B;  
        Y2 <= A OR B;  
        Y3 <= NOT Y1 AND Sel;  
        Y4 <= NOT Y2 AND NOT Sel;  
    END dataflow;
```

Here is a detailed explanation of this code.

1. The **ENTITY** declaration specifies the following:
   - The entity name is **nand_nor** (you can think of this as the name of your circuit).
   - The input **PORTs** to this circuit are **Sel, A, and B**. Each is one bit wide.
   - The only output **PORT** is named **Y**.
   - All of the inputs and outputs in this example have a data type of **std_logic**.
   - You will learn more about the **std_logic** type later, but for now it is enough to know that this is the data type we will be using for all one bit values.

   If you know ABEL, a **PORT** in VHDL is similar to a “pin” in ABEL.

   As a matter of convention, we always make sure that the file name for our source code matches the entity name. So, for this example, we would call the file **nand_nor.vhd**.

2. The **ARCHITECTURE** declaration specifies a separate name, **dataflow**, for this particular implementation of entity **nand_nor**. The architecture name is entirely arbitrary, but in ece495d our usual convention is to use a name that describes the coding style being used (see definition of coding style below).

3. For the moment you can think of a **SIGNAL** in VHDL as being roughly equivalent to a variable in C-code, but it is more appropriate to think of each signal as a wire in a circuit. If you are familiar with ABEL, signals are similar to the “identifiers” used in ABEL logic expressions.

   In this example, four 1-bit signals, **Y1, Y2, Y3, and Y4** are declared.

4. The rest of the code in this example consists of assignment statements. Note that the assignment statement for a **SIGNAL** is the operator `<=`
Pay attention to punctuation.

- Signals in a list are separated by commas.
- Signals are separated from the type by a colon.
- Signal type grouping is separated by semi-colons.
- No semicolon after the last signal type grouping.

Signal types you will use most of time

- Std_logic
- Std_logic_vector

The previous example illustrates all of these elements of syntax except for std_logic_vector which is a data type you will learn about soon.
A SAMPLE COMBINATORIAL CIRCUIT:
2 input NAND/ NOR

ENTITY nand_nor IS
  PORT (Sel, A, B: IN std_logic;
        Y: OUT std_logic);
END nand_nor;

ARCHITECTURE dataflow OF nand_nor IS
  SIGNAL Y1, Y2, Y3, Y4 :std_logic;
BEGIN
  Y <= Y3 OR Y4; -- Y is NAND for Sel='1'
                 -- and NOR for Sel='0'
  Y1 <= A AND B; -- Y1 is AND
  Y2 <= A OR B;  -- Y2 is OR
  Y3 <= NOT Y1 AND Sel;
  Y4 <= NOT Y2 AND NOT Sel;
END dataflow;

Look through the code to find an example of each of these elements of VHDL syntax:

Signals (or PORTS) in a list are separated by commas.
Signals are separated from the type by a colon.
Signal type grouping is separated by semi-colons.
No semicolon after the last signal type grouping.
More syntax

**Architecture body specifies logic**
- Must assign a name to the architecture (dataflow)
- FYI – you can have multiple architectures for the same circuit
- Must identify the entity that it goes with (nand_nor).
- Must declare internal signals.
- The rest describes functionality.

The real reason that the architecture body has a separate name from the entity is that VHDL allows you to create multiple implementations (architectures) for a single entity. We will only be creating one architecture for each entity in ece495d, so this is less of any issue for us (but the syntax requires it).

In the “real world”, the same circuit may initially be coded for simulation purposes only and then later re-coded in more detail for synthesis of hardware. Having different architecture names allows you to choose which implementation you want to use during simulation.
A SAMPLE COMBINATIONAL CIRCUIT:
2 input NAND/ NOR

ENTITY nand_nor IS                          -- entity declaration, circuit name
  PORT (Sel, A, B: IN std_logic;
       Y: OUT std_logic);           I/O pin names, types
END nand_nor;

ARCHITECTURE dataflow OF nand_nor IS       -- architecture body
  SIGNAL Y1, Y2, Y3, Y4 :std_logic;                -- define internal signals
BEGIN                                                     -- The following are concurrent
  Y <=      Y3       OR      Y4;                -- Y is NAND for Sel='1'
  Y1 <=  A     AND      B ;                       -- Y1 is AND
  Y2 <=  A     OR     B;                        -- Y2 is OR
  Y3 <=   NOT    Y1     AND    Sel;              -- Y3 is NAND for Sel='1'
  Y4 <=   NOT    Y2     AND    NOT  Sel;          -- Y4 is NOR for Sel='0'
END dataflow;
Comments

Comments in VHDL preceded by "--".

- Meaningful comments essential – others need to understand your code.
- Header comments for name, purpose of circuit, designer, date, history of changes. (didn’t have room for this in example). ch script creates one
- In-line comments for non-obvious logic or syntax
- Credit for code from elsewhere (text, web)
- Adequate commenting will be mandatory.
- Labs will be deducted points for poor comments.
A SAMPLE COMBINATORIAL CIRCUIT:
2 input NAND/ NOR

ENTITY nand_nor IS
  PORT (Sel, A, B: IN std_logic;
        Y: OUT std_logic);
END nand_nor;

ARCHITECTURE dataflow OF nand_nor IS
  SIGNAL Y1, Y2, Y3, Y4 : std_logic;
BEGIN
  Y <= Y3 OR Y4;   -- Y is NAND for Sel='1'
...

Here is a sample of the kind of header comments you will be expected to create in ECE495D. For a more complex function, the description will need to be a bit more detailed.

Note: you are not expected to have a comment on every single line of code.
Dataflow is just one VHDL coding style for synthesis. We will look at two other coding styles that support synthesis into a gate level hardware design.
Coding “style” may seem foreign to you, but some partial analogies can be made to sequential computer programming languages.

Consider C++ (assuming most of you have had C++) – Even though C++ includes a lot of object oriented programming capability, one can still write code using strictly C style syntax.

If you were writing C code for a small microcontroller, you might want to restrict yourself to data types and operations that are natively supported by the target microcontroller. This could be considered a particular style of C code. Failure to stay within this style may result in much less efficient code.

In VHDL, a particular coding style will restrict you to a particular subset of the language syntax and to particular ways of structuring your code. The reasons for doing this are similar to the microcontroller example given previously – you need to limit your code to a subset that correlates closely with digital logic circuit structures. However, VHDL synthesis is much less forgiving than a C compiler – straying from the synthesizable subset of VHDL results in code that may not synthesize at all, or worse yet – hardware that doesn’t work the way that you intended.

Dataflow style VHDL code specifically consists entirely of concurrent assignment statements. Very conservative or paranoid VHDL coders will do everything except for registers using dataflow statements. Dataflow is the hardest to write initially, but it is the easiest to get working in synthesis and in hardware.
Is there anything wrong with this code?

NO!

ENTITY nand_nor IS
  PORT (Sel, A, B: IN std_logic;
        Y: OUT std_logic);
END nand_nor;

ARCHITECTURE dataflow OF nand_nor IS
  SIGNAL Y1, Y2, Y3, Y4 :std_logic;
BEGIN
  Y <=      Y3       OR      Y4;  -- Y is NAND for Sel='1'
                           -- and NOR for Sel='0'
  Y1 <=  A     AND      B ;  -- Y1 is AND
  Y2 <=  A     OR     B;  -- Y2 is OR
  Y3 <=   NOT    Y1     AND    Sel;
  Y4 <=   NOT    Y2     AND    NOT  Sel;
END dataflow;

If this were sequential code, the order of the assignment statements would not make sense. It would appear that Y3 and Y4 are being used before a value has been assigned to them. However, in VHDL these statements are treated as if they all operate in parallel – the way that circuits work. For this reason, assignment statements are known as concurrent statements (but these are not the only kind of concurrent statement).

The way to understand how this code works is to first visualize (or draw) the equivalent gate level schematic and then think about how the circuit works.
Other Dataflow statements

-- Conditional assignment
-- NAND (when in doubt about operator precedence, use parenthesis)
Y1 <= '1' when (A = '0' or B = '0') and Sel = '1' else '0';
-- NOR
Y2 <= '0' when A = '1' or B = '1' or Sel = '1' else '1';
Y <= Y1 or Y2

-- Selected signal assignment
Y1 <= not (A AND B);  Y2 <= not (A OR B);
with Sel select
  Y <= Y1 when '1',
      Y2 when '0',
      '0' when others; -- good practice

There are several other dataflow style concurrent statements that can be very useful in coding more complex logic structures. The selected signal assignment statement is particularly convenient as one can directly implement a truth table or a multiplexer with it (we will get to that later).

See the VHDL language reference for concurrent statements, http://www.acc-eda.com/vhdlref/index.html (there is a link to this on the course web site), and section 4.7 of Wakerly for more examples.
Data types in VHDL

Several described in Wakerly 4.7.3
We will normally only be using

- `std_logic` (see p. 273)
  - can take on 9 different values in simulation
  - for synthesis to hardware, we usually only care about ‘1’ and ‘0’
    (we will consider use of “don’t care” later)
- `std_logic_vector`
  - an array of `std_logic` values
- `unsigned` or `integer` for convenience of use in arithmetic
  operations (not for input/output)
- User defined enumerated types – mostly to identify states in
  a state machine
- We may on rare occasions use `std_ulogic` and
  `std_ulogic_vector`
  - we will only need this for tri-state busses

VHDL supports a wide variety of data types but for ece495d you will normally be restricted to `std_logic` and `std_logic_vector` for inputs and outputs. Most of the time, you can think of `std_logic` type as a simple binary value (‘1’ or ‘0’), but only in the sense that a wire in a digital circuit can be thought of as having a binary value. In practice, a wire can be in a high impedance state (‘Z’ in VHDL), a resistive pull-up to logic 1 (‘H’), a resistive pull-down to logic 0 (‘L’), and an unknown or indeterminate state (‘X’). For simulation, flip-flop or latch outputs may be in an uninitialized state (‘U’) before being reset. For synthesis, it is useful to specify an output as being a don’t-care (‘-’) – do you recall how don’t-cares in a Karnaugh map allow you to optimize logic more efficiently?

The `std_logic_vector` is just a one dimensional array of bits and these bits can be addressed using an array index (examples are coming).

See the “Using Standard Logic” section of the online VHDL Language Reference for a lot more detail.
vec1, vec2, and vec3 are all declared as an 8 bit array. In the actual circuit, these will correspond to 8 bit wide data busses. The array index range can be specified as 7 downto 0 or as 0 to 7. The first index specifies the most significant bit (MSB), the 2nd index in the range specifies the least significant bit (LSB). As a matter of convention (and convenience when interpreting a bus as a binary value), we will use the largest index for the MSB.

An array value can be written as a string in binary format using double quotes as shown for vec1 and vec2.

The assignment to vec3 illustrates several things that can be done with std_logic_vector arrays. One can specify a range of indices to extract a set of bits from a vector, such as vec1(7 downto 4). One can use the concatenation operator, &, to combine substrings and individual bits together to form a single string. Notice again that a single bit must be contained within single quote marks whereas multiple bit values require double quotes.

The assignment to uno shows how you can extract a single bit from an array and assign it to a single bit std_logic signal.
"&" operator

"&" operator in VHDL is used for concatenation between bits.
Useful for writing codes that involve shifting and rotating bits.

- \( X \leq A(14 \text{ DOWNTO } 0) \& A(15); \)
- \( X \leq A(15) \& A(14 \text{ DOWNTO } 0); \)
- \( X \leq A(14 \text{ DOWNTO } 0) \& '0'; \)
- \( X \leq '0' \& A(14 \text{ DOWNTO } 0); \)

In the example above, identify each of the following cases:

1 bit rotate left
1 bit rotate right
1 bit shift left
1 bit shift right
Example usage for barrel rotator

-- This will rotate the bits in "A" according to how many bits
-- specified by "B".

description rotator is
port (  
    A: in STD_LOGIC_VECTOR(15 downto 0);  
    B: in STD_LOGIC_VECTOR(3 downto 0);  
    output : out std_logic_vector (15 downto 0); 
);  
end rotator;

architecture behavioral of rotator is
signal X,Y,Z:STD_LOGIC_VECTOR(15 downto 0);  
begin  

Here is an example of a more sophisticated use of concatenation to implement a variable shifter block. As an exercise, try to draw a schematic for this code.
The assignment to X implements a one bit rotate left of input A if B(0) = ‘1’
The assignment to Y implements a two bit rotate left of X if B(1) = ‘1’
The assignment to Z implements a four bit rotate left of Y if B(2) = ‘1’
The assignment to Output implements an eight bit rotate left of Z if B(3) = ‘1’

Taken as a whole, this circuit performs a rotate left of input A by the number of bits indicated by input B (interpreted as an unsigned binary integer)

The schematic for this circuit would look like four 2-way, 16 bit wide multiplexers (mux) cascaded from left to right. The select line for each mux is taken from a different bit from input B.
This is a variation of the barrel shifter example given previously, but what is different about it?
VHDL – Behavioral Style

Similar to C programming style
- can use constructs including "if", "for", "case" etc.
- order of statements is significant
- Best used for complex combinational logic
- Easiest to write, Easy to foul up
- In general, code has to translate into combinational logic!
- Keep your code simple
- Beware – easy to accidentally create latches

Consider example of 2-1 MUX

Behavioral style VHDL code is the most flexible coding style in VHDL, but it is also the one style for which you have to be very careful to avoid synthesis problems. Remember – if you can’t synthesize your code, then you will not be able to build hardware for it.

Behavioral style VHDL code looks a lot like C code, but if you intend to create hardware from it, you are severely limited as to the coding structures you can use. Do not write behavioral code from scratch. Instead, use code examples that are already known to synthesize, then modify expressions and assignment statements as needed to implement your required logic function.
A SAMPLE COMBINATORIAL CIRCUIT:
2 to 1 MUX

ENTITY mux IS
    PORT (d0, d1, sel: IN bit;
         q: OUT bit);
END mux;

ARCHITECTURE behavioral OF mux IS
BEGIN
f1:
    PROCESS (d0, d1, sel)
    BEGIN
        IF sel = `0'
            THEN
                q <= d1;
            ELSE
                q <= d0;
        END IF;
    END PROCESS f1;
END behavioral;

Advice – read this section of notes over and over until you understand it.

Some new elements of syntax are introduced here: a process block with a sensitivity list, an if-then-else-if structure, and a label for the process block. Here is an explanation of each:

Think of a process block as representing a subcircuit within your design that is described using behavioral (C like) syntax. There are other coding units (such as functions or procedures) that fit this description, but we will primarily use process blocks for behavioral style code. A subcircuit described by a process block should be of one of the following types: a register (a collection of edge triggered flip-flops), purely combinational logic (no clock signal, no memory, and no feedback within the circuit), or a well-defined sequential block such as a counter, statement, or shift register. If you try to use a process block for any other purposes, or if you try to combine several of these functions into one process, you can look forward to sleepless nights debugging the synthesis of your code.

The function described by this process is purely combinational, i.e., it could be built entirely out of nand gates with no feedback paths.

The process block itself begins with the PROCESS statement and ends with the END PROCESS statement. Between the BEGIN and the END PROCESS, one can insert behavioral style code.

Process Inputs: Any signal that is on the write hand side of an assignment statement in the process or included in a conditional expression is an input to the process. So in this example, sel, d0, and d1 are inputs to the process.

Process Outputs: Any signal that is on the left hand side of an assignment statement in the process is an output (or can be used as an output) of the process. In this example, signal q is the only output.

The function described by the if-then-else structure is a simple 2-way multiplexer. When sel is 1, the output is taken from input d1. When sel is 0, the output is taken from d0. If you think about it, this is another way of describing what a multiplexer does – it selects one of several inputs and passes that value to an output.

Sensitivity list: The sensitivity list is a list of those process inputs that can cause an immediate change in one or more output signals. In technical terms, the sensitivity list is the list of all asynchronous inputs to the process. In this example and for any combinational logic, all inputs are asynchronous inputs. Later on you will see that for clocked subcircuits, only certain input signals are required in the sensitivity list. WARNING – DON’T REMOVE SIGNALS FROM THE SENSITIVITY LIST TO GET A SOURCE CODE VERSION OF YOUR DESIGN TO WORK. The sensitivity list is ignored by synthesis – it is only used by the simulator. Consequently, if you make your source code work by removing a signal from the list, your synthesized version of the design will still be broken.

Process Label: The label in front of the process consist of an alphanumeric string starting with a letter and terminated by a colon. The label here is entirely optional, but it can sometimes be helpful in interpreting synthesis messages that refer to this process block.

Equivalent circuit: A simple if-then-else structure can always be mapped to a two-way multiplexer, but more complex if-then-else-if-else-end if structures and nested if-then else structures will result in either several layers of two-way multiplexers or in a priority-encoder structure (see Wakerly for information on priority encoders)

Note: A 2 to 1 multiplexer can be done much more compactly with data flow code.
More behavioral style syntax

**Process Block** (translates into a block of circuitry)
- As a whole, is a concurrent statement that merely contains sequential statements (e.g. the if/then/else).
- This is the only place for sequential statements
- The Process block, as a whole, is a “concurrent” statement
- Reminder:
  - concurrent statements are interpreted as parallel
  - sequential statements are interpreted in order

**Sensitivity list.**
- Here the list contains: d0, d1, sel.
- Any time a signal in the sensitivity list changes states, the code in the process is re-evaluated (for simulation purposes)
- What happens in the circuit implementation?

Most of this was discussed in the previous page of notes, there are two more important details:

1. During simulation, output signals from a process DO NOT CHANGE until all of the code in the process has been evaluated. Consequently, you can’t assign a value to a signal early in the process and then read that new value from that signal later in the process. If you try to do this, you will get the signal value that was assigned during the previous execution of the process.

   The reasons for output signals being handled this way will seem mysterious for a while, but hopefully it will become clearer with more examples and more practice.

2. The behavioral code inside of a process block consists entirely of sequential statements. In this context, sequential only means that the statements are interpreted in the order that they appear (it has nothing to do with the concept of clocked logic circuits). This is very different from dataflow (concurrent) statements for which the order of statements is irrelevant. However, the process block, taken as a whole, is a concurrent statement.
An analogy can be made between a process block and a hierarchical schematic. A process block corresponds to a block that you might include in a block diagram or schematic. It has inputs and outputs. It is operating all of the time. Any time an input changes, the change propagates through the block and appears at the outputs. The underlying schematic is a gate level schematic that implements the same logic function as what was described by the behavioral code inside the process block.
The case structure is a sequential, behavioral syntax that maps nicely into many-way multiplexers. This example presents a 4-way multiplexer. Two selection bits, `sel`, determine which of the four mux inputs are passed to the output.

Note the `when others` branch. In this case, the `when others` is not strictly necessary for synthesis since all of the two bit boolean combinations are covered. However, if your code doesn’t cover all combinations, it is critical that you use a `when others` to specify output values for all left-over input combinations. Failure to do so will result in either unintended latches or feedback paths in your circuit (for reasons to be described later).

In general, a Case structure is to be preferred over an extended if-then-else structure because the Case structure maps directly to a multi-way multiplexer whereas the if-then-elseif-... maps to of a chain of nested two-way multiplexers or a priority encoder that has sort of a ripple-carry kind of structure. Ripple-carry or high nested structures result in long combinational logic delays.
For loop notice the use of variables

signal y: std_logic_vector(15 downto 0); signal d: std_logic_vector(63 downto 0);
signal d0,d1,d2,d3: std_logic_vector(3 downto 0);
signal Sel: std_logic_vector(1 downto 0);
...
d <= d3 & d2 & d1 & d0; -- concatenation of data inputs
mux: process(d0,d1,d2,d3,Sel)
  variable I,iSel: integer range 0 to 3;
  variable ibot: integer range 0 to 63;
begins
  iSel := CONV_INTEGER(Sel);  -- variable is assigned immediately
  y <= (others => '0');  -- signal isn't assigned till end of process
  ibot := 0;
  for i in 0 to 3 loop  -- for hardware, for loop range must be fixed
    if I = iSel then
      y <= d(ibot + 15 downto ibot);
    end if;
    ibot := ibot + 16;
  end loop;
end process; -- at this point, y takes on the last value assigned to it
-- consequently, it is essential that somewhere along the
-- way, a value is assigned to y

Here is non-obvious way of implementing the same multiplexer as before. I wouldn’t recommend coding a 4-way mux it this way, but for a
much larger mux it might make sense (imagine coding a 1024 way mux). Either way, this is useful as an example of several different
aspects of behavioral coding.

1. Most importantly, this example illustrates the difference between the way variables and signals are used:
   • Variables can only be declared and used locally within a process, function, or procedure
   • Variable assignments in VHDL are interpreted (and simulated) the way variables are in sequential programming languages such as C. I.e.,
     a value can be assigned to a variable and then that new value can be used later within the same block of code.
   • Signal assignments in VHDL, even inside of a process, are NOT interpreted the variable assignments are in C. During simulation, a signal
     assignment inside a process does not become effective until the process exits. This has a few non-intuitive consequences: a) any time you
     use a signal, you get the value that was in effect prior to the current execution of the process, and b) you can make multiple assignments
     to the same signal inside a process, but only the last assignment becomes effective.
   
   For further explanation, see the Language Overview->Sequential Statements->Signal and Variable Assignments section of the online
   VHDL language reference (which can be found on the ece495d web site)

2. Notice that I, iSel, ibot are restricted range integers. The range specification prevents the creation of 32 bit logic to handle these variables
   (VHDL integers are 32 bit by default).

3. The CONV_INTEGER function does not add any circuitry to the design – no bits are changed, just the interpretation of the bits. Making an
   integer out of Sel allows you to use the value as an array index.

4. the (others => '0') is just a convenient way of assigning all bits of a std_logic_vector to '0';

5. the statement "y <= d(ibot + 15 downto ibot);" shows how you can extract a subset of bits from a bus.

6. There are at least two reasons I wouldn’t usually recommend writing a mux this way:
   a. It is a lot easier to make a mistake in the coding that may render the design non-functional, or may make the resulting circuit much larger
      than necessary.
   b. The synthesis software has to work a lot harder to interpret and optimize this style of code.

7. In order to produce hardware, the range on the for loop must be fixed. Why? Each iteration of the for loop actually results in additional
circuitry in hardware. However, once hardware is built, you cannot generally add circuitry on the fly to accommodate a change in the
   number of iterations of a loop.
Signals vs. Variables

**Variables** – only use when necessary
- can only be declared in process where used
- must be local to the process
- must be initialized inside the process
- value is assigned immediately

**Signals** – Use signals whenever possible
- all inputs/outputs of entities & processes
- implies existence of a wire in the circuit
  - though it might be optimized out
- in a process, signal is not assigned until end of process

This slide summarizes some of the comments on the previous slide.

For further explanation, see the Language Overview->Sequential Statements->Signal and Variable Assignments section of the online VHDL language reference (which can be found on the ece495d web site)
VHDL promotes the re-use of code by allowing one to compile functional code and type definitions into “libraries”. Libraries allow you to partition VHDL code in a way somewhat similar to what you do with C code.

In C, you may have numerous pre-compiled functions in a linkable object code file. When writing C code, you must use an include statement to pull in the requisite function definitions. Later when you compile and link your code, the compiler will link pre-compiled versions of each function library to your program.

In VHDL, you can have pre-compiled entities, architectures, functions, procedures, and data types located in special library directories. Unlike C, VHDL libraries can be further subdivided into packages. When writing the VHDL code, you must use a library statement to include the intended library and a use statement to identify what portion of the library you intend to use. In the example above, the library name is IEEE and the package name is std_logic_1164.

Later in the semester there may be one or two additional VHDL packages that you will want to use, but for now you will just need the library/use statements exactly as given in the example above.
Dataflow vs. behavioral

- Pay attention to text:
  - Wakerly 4.7.7 – Dataflow, 4.7.8 – Behavioral

- Dataflow style code
  - contains only concurrent statements
  - all statements are signal assignments
  - no variables in dataflow statements

- Behavioral style code
  - all statements interpreted sequentially (i.e., in order)
  - can use simple dataflow style assignment statements

- Mix of behavioral & dataflow
  - behavioral inside process blocks
  - data flow outside of process blocks

In some VHDL references a different terminology is used: behavioral code is referred to as “sequential” and dataflow code is referred to as “concurrent”. This terminology is somewhat confusing because in digital circuit design “sequential” refers to logic that requires a clock and storage elements. In VHDL, we will quite often use sequential statements to implement combinational (i.e., non-sequential) digital logic. In this course, I will make the following distinction:

sequential statement refers to VHDL statements that are interpreted in an order dependent manner such as statements inside a process block

sequential logic refers to digital logic circuits which require a clock and storage elements (flip-flops or latches). State machines, counters, and shift registers are all examples of sequential logic.

concurrent statement refers to any VHDL assignment statement that is not contained inside of a process, function, or procedure. Concurrent statements are like NAND gates on a schematic – the NAND gate is operating at all times regardless of whatever else is happening at the same time. Note: the process block, taken as a whole, is considered a concurrent statement.
Dataflow signal assignment

- Statements equivalent to boolean expression
- Order of statements doesn’t matter
- Dataflow code generally synthesizes easily

```vhdl
-- Table 4-51 from Wakerly text
architecture prime3_arch of prime is
signal N3L_N0, NEL_N2L_N1, N2L_N1_N0, N2_N1l_N0: STD_LOGIC;
begin
N3L_N0 <= '1' when N(3)=='0' and N(0)=='1' else '0';
N3L_N2L_N1 <= '1' when N(3)=='0' and N(2)=='0' and N(1)=='1' else '0';
N2L_N1_N0 <= '1' when N(2)=='1' and N(1)=='0' and N(0)=='1' else '0';
F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1l_N0;
end prime3_arch;
```

The simplest dataflow statements closely resemble a boolean expression. Two different types of assignment syntax are given in this example. One type uses a `when/else` structure in a manner that is similar to an `if/then/else` structure. The other type treats signals on the right hand side as if they were boolean variables.

While the logic function to be implemented is fairly obvious, there are some details of syntax that are not obvious. To fully understand the reasons for the syntax in these statements, you have to understand the data type constraints in VHDL. Any time you make an assignment to a signal of type `std_logic`, the expression on the right hand side has to produce a result of type `std_logic`. Most students use dataflow statements without thinking very hard about the data types involved, but understanding the data types reduces the chances of mistakes.

Consider the first assignment statement:

```vhdl
N3L_N0 <= '1' when N(3)=='0' and N(0)=='1' else '0';
```

We can break it up across multiple lines to try to make things clearer:

```vhdl
N3L_N0 <=
    '1' when N(3)=='0' and N(0)=='1'
else '0';
```

'`1`' is one of the possible values for a `std_logic` type signal. So, `N3L_N0 <= '1';` would be valid VHDL syntax.

`N(3)=='0'` is a comparison that produces a value of type `boolean` (similar to the result generated by comparisons in C). Consequently, `N3L_N0 <= (N(3)=='0');` would be invalid because the type of result produced by the expression `N(3)=='0'` doesn’t match the type of the signal `N3L_N0`.

The `when` clause requires an expression that produces a `boolean` result. The expression `N(3)=='0' and N(0)=='1'` produces a `boolean` value, so it can be used inside the `when` clause.

'`0`' is also a possible value for a `std_logic` type signal, so it can be used inside the `else` clause as a possible return value for `N3L_N0`.

Now consider the last assignment statement:

```vhdl
F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1l_N0;
```

All of the signals in this expression are of type `std_logic`.

When you perform a logic operation such as `and`, or, `not`, or `xor` on `std_logic` values, the result is also of type `std_logic`.

Hence, the assignment statement `F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1l_N0;` is valid syntax for this example.
Dataflow select statement

with/select construct – easy to construct from truth table

-- Table 4-53 from Wakerly text
architecture prime4_arch of prime is
begin
with N select
F <= '1' when "0001",
   '1' when "0010",
   '1' when "0011" | "0101" | "0111",
   '1' when "1011" | "1101",
   '0' when others;
end prime4_arch;

Here is another type of dataflow statement that does not look nearly so much like a boolean expression. Instead, it looks a lot like a truth table, which if you remember anything from your introductory digital logic design course, could be translated into a boolean expression if needed. This syntax is also rather similar to the case statement that is available in behavioral VHDL.

Here is how to interpret this code fragment.

N is the only input to the circuit described by this code. F is the output of the circuit. The value of N is used to select the value of output F from a list. If N has a value of “0001”, “0010”, “0011”, “0101”, “0111”, “1011”, or “1101”, the output F will take on a value of ‘1’. For all other cases, the output F will be ‘0’. The pipe “|” symbols can be read as “OR” in this type of statement, but it really just serves as a separator in the list of selection values in a with/select statement.

As you might suspect, this code could have been shortened to:

with N select
F <= '1' when "0001" | "0010" | "0011" | "0101" | "0111" | "1011" | "1101",
   '0' when others;

The longer version was presented just to show you that you can have an arbitrary number of when branches.

Although the declaration of N is not given, you should be able to infer that it probably is a 4-bit std_logic_vector type of signal. (it could also be a bit_vector, but will not be using that data type in ece495d). Likewise, you may infer that F is a std_logic signal.

Whether or not you noticed, this code fragment implements a 4-bit prime number detector.
This code example is a behavioral implementation of a 4 bit prime number detector presented in previous examples, but there is a good chance that this code would not synthesize for the simple reason that the modulus operator, \texttt{mod}, may not be supported by circuit synthesis software.

If modulus is supported, there is one reason one might want to set up the code this way. If you want to be able to change the number of bits of $N$, all you would need to do is to change the range of the integer variable and the range of the for loop. However, the modulus operation may produce a lot more circuitry than really needed.
Structural VHDL

In the class lecture you learned about:
- basic VHDL structure and syntax
- dataflow style VHDL code
- behavioral style VHDL code

There is one more major style, and it comes the closest to describing the organization of hardware:
- structural
Structural VHDL

Directly corresponds to a schematic.
- May not want to create structural VHDL by hand.
- Create a schematic, let software generate the VHDL.
- Will do this in lab 2

Usually used for high levels of design, not gate level.
- 1st create low level blocks (dataflow or behavioral style)
  - combinational logic, registers, state machines, counters, etc.
- Use schematics to tie blocks together.
- Generate VHDL netlist from the schematic
- Verify correctness of design by simulation.

Structural style VHDL is code that explicitly describes how sub-circuits are connected to each other. Imagine taking a schematic, making a list of components in the schematic, and then listing the input and output connections of each component. This is exactly how structural VHDL statements work.

The concept of structural VHDL may be simple, but the coding is very tedious. For this reason, schematic editors (such as Cadence composer) or special purpose graphical design software (such as HDL designer) are used to draw the design. The structural code is then generated automatically from the diagram.
Here is yet another version of the 4 bit prime number detector. This example introduces two new elements of VHDL syntax – the component declaration, and the port map.

A component declaration is very similar to an entity declaration – it specifies the name of a component (INV, AND2, AND3, OR4 in the example above) and a list of the input and output signals along with their data type. In the case of the INV component, there is one input \( I \) of type std_logic and one output \( O \) of type std_logic.

Actually, the component declaration information is redundant but it is required. The name and input/output specifications must match the name, inputs, and outputs of an entity (and associated architecture) that must already exist somewhere else in a VHDL library.

A port map is a statement describing how a particular component is used in your circuit. A port map consists of:

- an arbitrary but unique label such as \( U1 \). The label only needs to be unique within the scope of the current architecture body.

- the component (and corresponding entity) name such as INV

- the key words port map

- a list of signals (think of them as wires) connected to the component

Simply including a list of signals in a port map is not enough, it is necessary to be able to identify which signal is connected to which I/O pin of the component. There are two ways of doing this.

The shortest way is to list the signals in the same order as the pins to which they are connected (as given in the component declaration). For example, look at U8: Wire N3L_N1L is connected to input I1 of component OR4. The disadvantage of this syntax is that you have to refer back to the component declaration to interpret the code.

A more readable and less error prone syntax (but more verbose) is to list each pin name together with the corresponding wire (signal) name. The exact syntax for each entry in the port map consists of the pin name first, separated from the signal name by a "\=>".
Compare this schematic to the preceding code example. Look at each block and see if the label, component name, and inputs and outputs match the code. Notice that the ports $N$ and $F$ in the entity declaration are shown as input/output arrows on the diagram.
Coding style affects synthesized schematics.

- Synthesis software tends to take code very literally unless optimization constraints are chosen well.

Examples

- Integer variables are likely to produce 32 bit busses.
  - To combat this - create an integer type that is range limited.

- "+" operations are likely to result in adder circuits.

Choose coding style wisely
VHDL Style Example

First draw the symbol and a gate level schematic for a 4 input mux
- Inputs: D(3:0), Sel(1:0)
- Output: Y

Write VHDL for the same 4 input mux

Which style (dataflow, behavioral, or structural) would you choose & why?

What style would you use & why:
- 2x1 mux?
- 1024x1 mux?

Why draw the schematic first? In order to write code that describes hardware, you have to first understand the hardware that you are trying to describe.
Mux Symbol/Schematic
This structural code almost directly describes the preceding schematic except that the inversion bubbles correspond to inverters in this code.
Datapath 4 input mux
(using with/ select syntax)

entity mux4 is
  port ( Sel: in STD_LOGIC_VECTOR (1 downto 0);
       D: in STD_LOGIC_VECTOR (3 downto 0);
       Y: out STD_LOGIC);
end mux4;

architecture datapath of mux4 is
begin
  with sel select
    Y <=
      D(0) when "00",
      D(1) when "01",
      D(2) when "10",
      D(3) when "11",
      'U' when others; -- (output undefined if inputs undefined, ignored by synthesis)
end datapath;

!! There are several ways of writing dataflow code for this Multiplexer !!
This is a very concise (and flexible) way of coding the multiplexer, but the relationship to hardware is not at all obvious. In fact, if all you did was look at this code, you might come to the conclusion that there was no hardware involved at all except for some wires. WRONG! In hardware, when you select an element from an array, you are really performing a multiplexing operation. This is actually rather similar to the way that addresses are used to select locations in RAM.

Actually, you didn’t have to use a process and behavioral code to take this approach. Consider the following fragment dataflow code:

```
I <= CONV_INTEGER(Sel);
Y <= D(I);
```

where I is signal of type integer.
2 bit mux vs. 1024 bit mux
(i.e., why use behavioral?)

2 bit mux
- structural (schematic) easy to derive
- could be done in one dataflow statement

1024 bit mux
- schematic would be huge & tedious
- could do in one data flow statement, but would be huge (unless you use an array index)
- could be done compactly in behavioral style
ENTITY mux1024 IS
  PORT ( D: in STD_LOGIC_VECTOR (1023 downto 0);
        Sel: in STD_LOGIC_VECTOR (9 downto 0);
        Y: out STD_LOGIC);
END prime;

ARCHITECTURE behav OF mux1024 IS
begin
  process (D, Sel)
  variable i: INTEGER range 0 to 1023;
  begin
    i := CONV_INTEGER(Sel);
    Y <= D(i);
  end process;
end behav;
Latches, Flip-Flops, and Registers

Latches are your Enemy
Flip-Flops and Registers are your Friend
Registers are just a collection of Flip-Flops

Now, finally, we get to syntax for the design of memory elements. When writing VHDL (or Verilog) code for hardware synthesis, DO NOT simply rely on signals or variables to implicitly remember a value – use a flip-flop or register. The syntax for a flip-flop or register is actually one of the simplest things you will learn, but figuring out when and where a register is needed is one of the more difficult things. First, make sure you remember how a flip-flop and a latch work.

One more comment on latches – we will talk about how a latch works and how to code it, but mostly so that you will understand why and how to avoid latches.
Latches & Flip-Flops (FF)

Similarities

- Both are Sequential Elements (i.e., require a CLOCK signal)
- Store/hold a value for a certain time interval.

Differences

- FF: edge sensitive
  - Latches: “TRANSPARENCY” Window
- FF: MAXIMUM 1 transition/clock period
  - Latches: multiple
- FF: holds data FULL clock cycle
  - Latches: HALF cycle.
- Latches request less circuitry than FFs

A bit of terminology: In some texts and papers you may find the terms flip-flop and latch used interchangeably. In ECE495D, we will maintain a strict distinction. The term flip-flop will only be used to refer to a clock edge triggered device. “latch” will only refer to memory elements that are level sensitive.

In real-world designs, there is a trade-off to be made between flip-flops and latches. Flip-flops are generally far safer to use than latches, mainly because they are edge triggered. This characteristic protects you against a lot of the potential timing problems that can occur with level sensitive devices. However, latches require considerably less circuitry than flip-flops. Very large blocks of memory will use level sensitive circuits to keep circuit area under control. However, the safety of flip-flop based design outweighs the are benefits of latches for many applications.

If you are the least bit uncertain about flip-flop operation, dig out your introduction to logic design book and review this topic.
This slide illustrates the functional difference between a flip-flop and a latch.

When enabled, a latch passes an input straight through to the output. The output value is held while the latch is disabled.

A flip-flop only passes the input through to the output only on a selected clock edge. The output is stable all of the time except for a short time after the selected clock edge.
Latches & Flip-Flops

- In ECE495d & 437, always use FF’s
  - in spite of larger size
- Problems with latches
  - easy to accidentally create loops
    - consider counters & state machines
  - quite often you need a feedback path
    - in such cases, timing becomes tricky
  - makes automated testing difficult
    - big issue in industry

Two major problems make it difficult to produce reliable circuits with latches:

1. Feedback paths. Most sequential designs (think of state machines) require feedback paths from the output of a memory element, through some combinational logic, and back to the input. With level sensitive device, this can easily lead to race conditions.

2. Though we will not have time to cover automated hardware defect testing techniques, it is enough to say for now that latches can make it a lot more difficult to implement automated testing.

Some time ago, on an online ASIC design forum that I read, an ASIC designer said that anybody who uses latches should be shot. Another designer, responding to this statement, said that designers who use latches will wish that they had been shot.
Latches & Flip-Flops

Example from EE Times, 12/23/02

What if both latches are enabled?

Here is an illustration from an EE times article that shows how latches can lead to a combinational loop.

In case you don’t know, combinational logic loops can lead either to oscillation or to implied latches. Consider a couple of simple examples:

1. An even number of inverters connected in a loop will act as a latch that stores a value until the loop is broken.

2. An odd number of inverters in a loop will create an oscillator with a period equal to twice the total delay of the inverters in the loop.

Neither of these behaviors are desirable as they can be extremely sensitive to both the timing and particular input values to the circuit. They can also lead to very hard to debug problems after synthesis or in hardware.
VHDL Syntax for FF/ Latch

- 'EVENT indicates transition on a signal
  - (CLOCK'EVENT and CLOCK = '1') = TRUE if rising CLOCK edge
  - (CLOCK'EVENT and CLOCK = '0') = TRUE if falling CLOCK edge

- RISING_EDGE & FALLING_EDGE indicate particular edge
  - RISING_EDGE(CLOCK) = TRUE if rising CLOCK edge
  - FALLING_EDGE(CLOCK) = TRUE if falling CLOCK edge

- Use in “if” statements to test for clock transition
  - Use only for flip-flops, registers, and state machines

In order to describe a flip-flop in VHDL, you need to be able to specify a particular transition of the clock. A flip-flop can be rising or falling edge sensitive, but not both. In general, it is safest and simplest to make all flip-flops in a design sensitive to the same clock edge.
The **REG** process describes a rising edge sensitive flip-flop along with an active high asynchronous reset.

The **LATCH** process describes a simple level sensitive latch with no reset input.

Notice that neither of these code fragments include an `else` branch in the `if/then` structure. In the case of **REG** this is done because whenever Rst=’0’ or Clk is not rising, we want the flip-flop to simply hold the most recently specified value of Q. In **LATCH** we expect the circuit to hold the output value whenever the latch is disabled (Clk=’0’ in this example). In VHDL, if you fail to specify the value of an output for certain conditions, it will usually cause a latch to be inserted (or worse, a combinational logic loop will be created).

Notice that in **REG**, the reset condition is evaluated before the clock edge. This is because an asynchronous reset always takes precedence over the clock.

Combinational logic, by definition, must not contain any memory elements which would be implied by leaving off the else clause of an `if/then` structure. In addition, an asynchronous reset is meaningless for combinational logic because combinational logic should not contain any memory elements that can be reset.
Advice

- 'EVENT, RISING_EDGE, or FALLING_EDGE:
  - Do NOT use on multiple signals in your DESIGN (unless we tell you to)
  - i.e., there should usually be only a single clock
- Do NOT mix clock edge with other signals in an if() or elsif() expression
  - if (RISING_EDGE(CLK) and A='1') then you lose; end if;
- Signal for 'EVENT, RISING_EDGE, or FALLING_EDGE
  - MUST be a CLOCK signal
  - Must not be a random signal.
- There are exceptions, but few in EE495d understand
- Violators usually create unsynthesizeable or erratically behaving designs

If you hadn’t noticed already, one of the major differences between VHDL (for hardware synthesis) and C is that in VHDL you are NOT free to code things in any arbitrary way that happens to simulate correctly. Similar restrictions are true in Verilog (lest you think that life would be beautiful if we all switched to Verilog). The reason for these restrictions is that the code has to be translated into hardware. Even if you were designing digital logic using schematics, there are still severe design restrictions and “best practices” that are needed to ensure that you get a reliable digital circuit that is not overly sensitive to timing variations and is not unreasonably difficult to test. Restrictions on digital logic design translate directly into restrictions on what you can code in VHDL or Verilog.

The restrictions on this slide are a direct result of restrictions on logic circuit design. In the “real” world there are exceptions to these rules, but a lot of additional analysis and circuitry and required to ensure that violations to these rules will not cause a problem. At this point, you have no idea how to do these things safely, so do not do them.

1. Use of multiple clocks within the same digital circuit is very dangerous and is to be avoided.

2. Data signals should not be mixed with the clock. For example – you should never AND the clock with another data signal and feed the result to the clock input of a flip-flop.
Up to now, you know how to create combinational logic using a process or dataflow statements or structural code. In addition, you know how to create a flip-flop with a process. However, in a real digital system design, you will need multiple blocks of combinational logic as well as multiple registers or flip-flops. Something as simple as a counter requires at least two blocks – a register to keep track of the count and some combinational logic to increment the value of the counter. Most of the rest of the design and coding examples in this course module will involve multiple blocks of combinational logic and registers.
Concurrent Processes

Use multiple processes in an entity for
- State machines & other sequential functions
- One process for registers or storage elements
- One or more process for combinational logic
  - Generate register inputs (e.g., next state logic)
  - Generate output signals

You may be putting too much in one entity
- If you have multiple sequential processes
- If you have more than 3 or 4 processes

In an VHDL entity/architecture, one can use an arbitrarily large number of process. However, too many processes (more than 3 or 4) in one architecture is usually indicative of bad design – you probably have not thought adequately about how to partition your design into subcircuits. In addition, putting too much logic in one entity/architecture can make it much harder to understand how your design works and it makes it much harder to debug.

Why use multiple processes in a single entity? There are many common digital logic functions that lend themselves to being partitioned into 2 or 3 blocks. In particular, state machines always require a state register, next state logic, and sometimes output logic.
A circuit to code as VHDL

General comment: if this was the only kind of circuit we ever designed, there would be very little value in using a language like VHDL. However, it is useful to understand how such a simple circuit translates into VHDL. A VHDL description of this circuit will include the following elements:

1. An entity declaration with inputs X, Y, CLK, RST, and output Q.
2. Signal declarations for int_d and Q_int needed to connect the flip-flop and logic gate to each other.
3. A process (or dataflow code) to implement the logic gate which is just the NOR of X, Y, and Q_int
4. A process to implement the D flip-flop

Note: You can think of this circuit as a very simple state machine. The flip-flop stores the state of the state machine. The NOR gate computes the next state for the state machine. Since the output Q comes directly from the flip-flop, there is no output logic block required.
This code implements exactly the elements listed previously, included below for reference:

1. An entity declaration with inputs X, Y, CLK, RST, and output Q.
2. Signal declarations for int_d and Q_int needed to connect the flip-flop and logic gate to each other.
3. A process (or dataflow code) to implement the logic gate which is just the NOR of X, Y, and Q_int
4. A process to implement the D flip-flop

Look closely at the NEXT_D process. It would appear that this code violates one of the recently stated rules for coding combinational logic. However, the \texttt{int\_d <= '0'}; statement near the beginning eliminates the need for the else clause. Why is this the case? Remember that signal assignments do not become effective until the end of the process and that the last value assigned to a signal is the value that becomes effective. So, if the condition on the if statement is false, the output of the process, \texttt{int\_d}, will take on the value '0'. I.e., assigning a "safe" or "default" value at the beginning of a process is an easy way to make sure that the output value is defined for all conditions (thus eliminating possible inferred latches on signal \texttt{int\_d}).

In this example, the entire process \texttt{NEXT\_D}, could be replaced by the following very simply line of data flow code:

\begin{verbatim}
int\_d <= '1' when X='1' and Y='0' and Q_int='0' else '0';
\end{verbatim}

another way of writing it in dataflow style is

\begin{verbatim}
int\_d <= not X and not Y and not Q_int;
\end{verbatim}
Another way to code this

architecture behav of cp_example is
  signal Q_int: std_logic;
Begin
  Q <= Q_int;
  REG: process(Clk,Rst)
  begin
    if Rst = '1' then
      Q_int <= '0';
    elsif rising_edge(Clk) then
      Q_int <= '0';
      if X = '0' and Y = '0' and Q_int = '0' then
        Q_int <= '1';
      end if;
    end if;
  end process REG;

While this is a very common way to code this kind of function, I strongly advise against writing code in this style until you are very clear on how the code corresponds to the hardware.

If you encounter code that combines combinational logic and a register in this way, here are some tips on how to interpret it:

1. Any time you see an if or elsif expression based on a rising or falling clock edge, a register or flip-flop will be created.

2. Any signal assignment statements inside of a rising or falling clock edge condition will be stored in a register. In synthesis, the register name will usually be based on the signal name to be stored. In this example, the values assigned to Q_int will be stored in a flip-flop (since Q_int is just one bit wide).

3. Any expressions on the right hand side of an assignment statement (none in this example) and any conditional expressions inside of a rising or falling clock edge condition end up describing the next state logic.

Compare this code to the schematic given 2 slides earlier and see if you can match up each part of the code with the circuits and wires in the schematic.
Almost any sequential digital logic function can be treated as a simple state machine provided that there is just one clock signal and that all of the memory elements inside are edge triggered from the same clock edge. For relatively low complexity digital circuits, this is very powerful and useful way to approach things. Consequently, you can take the VHDL code for a state machine and adapt it to perform a wide range of functions.

Consider a counter: a counter can be viewed as a state machine where the register stores the current value of the count and the next state logic computes the next value of the count.

Consider a bidirectional shift register with a control input to select between shift right, shift left, and no shift. Viewed as a state machine, the next state logic will take the current value in the register and move bits left, right, or not at all and feed them back to the input side of the register.

Fairly soon, you will encounter functions that are too complex to practically treat as a single state machine. At that point, we will start to look at examples of how to use multiple state machines, registers, and combinational logic blocks to implement a function.
State Machine Review

If you can’t design a state machine...
- you can’t do much

Assignment (expect a quiz)
- Know difference between Moore & Mealy
- Know how to use & create
  - state diagram
  - state transition table
  - state transition equations
  - Forgot how to do? Look it up in Wakerly book

Possibly the two most important and useful skills that you should have learned in your introduction to digital logic design were:

1. The ability to define or specify a combinational logic function in a variety of formats including a truth table and boolean equations.

2. The ability to specify a state machine in a variety of formats including state diagrams, state transition tables, and next state (or excitation) equations.

I.e., you still need to be able to do by hand most of what you learned in the introduction to digital logic design except for the final optimization and preparation of detailed gate level schematics. VHDL (or Verilog) code can be written almost directly from the types of design representations you learned to create in the digital logic course.

Note: The ability to optimize combinational logic and state machines, while important to understand, is less critical because synthesis software will do this optimization for you.
State Machine Review

Generic Block Diagram for one type of State Machine:

Is this a Mealy machine? A Moore machine? How could you tell?

Burn this diagram into your brain. You will use it and refer to it many times this semester.

A finite state machine is a logic circuit consisting of a state register (to remember the current state or mode of operation), next state logic (to compute a new state based on the current state and the values of any data or control inputs), and output logic if any outputs are needed in a format different from what is available in the state register.

Notice! The clock and asynchronous reset signal only go to the register. Next state logic and output logic are combinational – thus there should be no clock or reset input to them.

Regarding Mealy vs. Moore, remember that in a Moore machine, the output depends exclusively on the current state stored in the state register. In a Mealy machine, the output depends on the current state along with some or all data and control inputs.
State Machine Review

Generic Block Diagram for one type of State Machine:

Is this a Mealy machine? A Moore machine? How could you tell?

Clue: look at what is connected to the output logic. In a Moore machine, outputs derive exclusively from the state. In a Mealy machine, outputs derive from both the state and external inputs.
The main problem with Mealy machines is that there are combinational signal paths that run from input to output without having to go through the edge-triggered register. This has several side effects, including but not limited to:

1. glitches on the inputs can propagate through to the outputs. In a Moore machine, this cannot happen.

2. carelessly designed external circuitry (state machines are usually part of a larger system) can create feedback paths from the state machine outputs back to the state machine inputs. As a result, you can end up with combinational feedback that can cause no end of problems (discussed several slides back). With a Moore machine, this cannot happen.

3. maximum clock speed (in a larger system) is likely to be slower because of long delay paths.

4. automated testing of digital systems often use registers that are specially modified to allow you to insert test inputs directly into parts of the system. However, the Mealy machine may bypass these registers that may be used for testing.
So, how do we actually code a state machine in VHDL? Look at the block diagram shown earlier—we need code for the register (a process), code for the next state logic (either a process or dataflow code), and code for the output logic (either a process or dataflow code).
State Machines Example

Consider the design of a ‘0110 Detector’.
- Pattern detection is a common state machine application
- Output F asserted high 0110 is detected

Do a Moore Machine representation
- Recall Moore has synchronous output
- Mealy has asynchronous output.

```vhdl
entity s_0110_detector is
  port(   clk, reset, I: in    std_logic;
          F: out   std_logic);
end s_0110_detector;
```
In ECE495D I find that while students have a general understanding of how to interpret or draw a state diagram, they don’t comprehend the timing or the circuitry implied by the state diagram. Here are a few important things to keep in mind that may help:

1. Each state (represented by a bubble) corresponds to one specific value stored in the state register. Unless there is a self loop in the state transitions (such as on state rcv0 above), the state machine will only be in this state for one system clock cycle.

2. The arrows (and the conditions written next to them) are what determine when the value in the state register will change and to what value.

3. The arrows (and the conditions written next to them) are what determine the design of the next state logic.

4. State transition conditions are only checked at time of the active clock edge for the system. Between active clock edges, data and control input values are ignored.

5. For a Moore machine, the output logic is determined by the output value specified in each state bubble.
Data types for State Reg.

- **STD_LOGIC_VECTOR**: options
  - use literal value strings: “0010”
  - declare a constant name for readability
    - will see an example of this

- **Enumerated data types**:
  - values need not be numbers.
  - don’t have to worry about coding of states
  - type all_states is
    (WAITING, RCV0, RCV01, RCV011, RCV0110);
  - will see an example of this

When designing a state machine, you have to decide upon the most advantageous representation for the values to be stored in the state register. This is determined by the data type you choose for the register (state) signal.

If you want or need to directly specify the binary pattern to represent each then a `std_logic_vector` type is appropriate.

If you don’t want or need to specify the binary pattern to represent each state, use an enumerated data type. You haven’t seen enumerated types in VHDL before, but they are very similar to enumerated types in C. You just specify a list of names for each possible value. There is an important advantage to using an enumerated type. It gives the synthesis software freedom to choose codes for each state in a way that optimizes the overall circuit design. However, sometimes you can simplify the design of other parts of your system by choosing convenient binary values for each state yourself.

One disadvantage of using an enumerated type occurs when you re-simulate your design after synthesis. If you try to view the waveforms for the state register, you won’t know how to interpret the binary values that are displayed.
On to the actual code for the 0110 detector. We have decided to use an enumerated data type for the state (and next_state) signal. Take note of the fairly simple syntax for doing this. In particular, look at the type statement and the signal declaration for state and next_state.

The StateReg process looks pretty much like register code you have seen before except for one thing – in the reset branch, state is assigned to a value of WAITING (one of the enumerated values for state_type), instead of a binary value. Any kind of signal assignment inside of a clock edge condition will produce a register regardless of the data type of the signal.

The output logic is a simple dataflow statement that can be generated almost directly from the state transition diagram. Compare the output logic here to the 0110 detector state diagram provided earlier.
Compare this code fragment for next state logic to the state diagram presented earlier. Each \texttt{when} branch in the \texttt{case} structure corresponds to one state in the state diagram.

Consider the \texttt{when} branch for the state \textit{WAITING}:

the code inside this \texttt{when} branch describes all of the outgoing transitions (arrows) from the \textit{WAITING} state bubble to other “next” states.

each assignment to the signal \texttt{nextstate} corresponds to one of the arrows in the state diagram.

Other observations:

1. Please note that the next state logic is \textit{combinational}. There is no clock or asynchronous reset. Given state assignments (binary values) for each of the state names, one could directly derive boolean expressions for each bit of the \texttt{nextstate} output signal.

2. The \texttt{when others} branch is used to cover any possible unused state assignments. In this state machine, there happen to be five state, which require at least 3 bits in the state register to represent. Since a 3 bit state register can represent 8 possible states, there are three unused state assignments. The \texttt{when others} branch defines what happens if the state machine finds itself (perhaps on power-up) in one of the undefined states.

In general – no matter the value of \texttt{state} or control input \texttt{I}, the value of \texttt{nextstate} is always defined, thus preventing any inferred latches on the signal \texttt{nextstate}. 

Combining Reg & Next State

- Experienced coders & text book examples often combine the state register and next state code
- Advantage: somewhat less typing
- Trade-off:
  - can't accidentally create inferred latches
  - sloppy code can add unintended flip-flops
    - & reset won't be defined for those flip-flops
    - can cause unpredictable behavior on startup or reset
    - even harder to identify than unintended latches
- Following is a version of the 0110 detector
  - Only the state reg process is shown since rest of code is minimally changed (nextstate signal no longer needed)

In ECE495D, the state register and combinational logic are coded using separate processes of dataflow code. This is done in order to make the relationship to hardware as obvious as possible. However, most experienced coders will combine the next state and register logic together. If you are reading this, you are probably NOT an experienced VHDL coder. Until you are very comfortable with the relationship between the code and the hardware, keep the register and next state logic separate. They can be part of the same entity/architecture, but do not combine them into a single process.

Note: output logic is usually NOT combined with the register process unless you intentionally want to insert a flip-flop or register in the output line. Why is this? The only place to incorporate the output logic is inside the clock edge branch. However, any signal assignments inside a clock edge conditional will cause registers or flip-flops to be created.
Compare this code to the earlier sample code for the state machine. What is different?

1. There is no separate next state process.

2. The next state logic is embedded inside of the clock edge branch.

3. No `nextstate` signal is required. Next state values are assigned directly to `state`.

What is the same?

1. Except for the change of signal name, the next state logic inside the clock edge branch is identical to logic inside the next state process from the previous example.

2. After synthesis, you should get exactly the same circuit as before consisting of a register, next state (combinational) logic, and output (combinational) logic.
More State Machines

Up-down counter.
- Counts from 0 to 2.
- UP=1, DOWN=0 means count up and hold at 2
- UP=0, DOWN=1 means count down and hold at 0
- For all other cases of UP and DOWN the count holds.
- Outputs:
  - TOP and BOT, indicates count at max or min
  - ST, the number we are currently on (matches state).

Constant declarations for states
- constant State_1 : std_logic_vector(2 downto 0) := "001";
- Allows outputs based directly on state value.

Here is another state machine example that shows how a small counter can be coded in the same way as the previous state machine example.

The state diagram is left for you to draw as an exercise.

This example also illustrates how constant declarations can be used to define the state coding.

Note: usually, a counter will be coded using either an integer, signed, or unsigned data type for which one can directly express the next state logic in the style of $\text{nextstate} \leftarrow \text{state} + 1$;
entity updn02 is
    port (UP, DOWN, Clk, Rst: in std_logic;
          TOP, BOT: out std_logic;
          St: out std_logic_vector(1 downto 0));
end updn02;
architecture behavioral of updn02 is

constant NUM0: std_logic_vector(1 downto 0) := "00";
constant NUM1: std_logic_vector(1 downto 0) := "01";
constant NUM2: std_logic_vector(1 downto 0) := "10";

signal state, nextstate: std_logic_vector(1 downto 0);

begin
  TOP <= state(1) AND NOT state(0); -- Top when at NUM2
  BOT <= NOT state(1) AND NOT state(0); -- Bottom when at NUM0
  St <= state; -- send current state to output

REGS: process(Clk,Rst)
begin
  if Rst='1' then
    state <= NUM0;
  elsif rising_edge(Clk) then
    state <= nextstate;
  end if;
end process;

Observations:

1. Each state is represented by a separate **constant** value for which the number of bits matches the number of bits in the declarations of the **state** and **nextstate** signals.

2. The output logic is given using dataflow style code.

3. The line “St <= state;” does not actually generate any logic. Thus it does not constitute “output logic”. It is merely included to make up for VHDL restrictions on the use of **out** port signals which prohibit any use of an **out** port value inside the architecture body.

3. The **REGS** process is the same register code as you have seen several times already.
If you didn’t already draw a state diagram for this counter, draw it now and compare it to the next state code presented in this slide.

It should be apparent to you by now that next state code in VHDL can be written almost directly from a state diagram.

Comments on “don’t care” value for nextstate:

By specifying a don’t care ‘-’ for each state bit, you allow the synthesis software to choose a value that allows for the greatest optimization. This is similar to what you should have learned in your introduction to digital logic design course regarding the use of don’t care values in a K-map – it allows for more optimization than would otherwise be possible.

There is one inconvenient aspect to using the don’t care value – it makes it harder to compare simulation results for the source and mapped (synthesized) versions of your design. In source code simulation, the don’t care will be displayed as a don’t care. In the mapped version, the value will appear as either a ‘1’ or ‘0’, whichever proved to be more convenient for optimal circuit mapping.
Controllers

Even though state machines are a very powerful tool in digital system design, a single state machine becomes unwieldy when implementing more complex systems. It becomes necessary to take a complex system and break it up into components that can be implemented in a reasonable manner using state machines, other simple sequential blocks, and blocks of combinational logic.
Controller Overview

Controllers are used everywhere

- Automobiles, appliances, communication interfaces, inside CPUs etc.
- Used to control a sequence of events, subject to user or sensor inputs

Implementation alternatives:

- Firmware + μprocessor or μcontroller
- Custom logic (as in ece495d) – ASIC or FPGA

It should be noted that a lot of, maybe most pure controller applications are probably best served by a microcontroller because the performance requirements are not severe enough to justify either an ASIC or an FPGA. However, most ASIC applications require multiple controllers to manage functions within the ASIC such as a computational datapath or a bus interface. Consequently, it is important for an ASIC designer to understand how to implement control logic.
Many relatively simple control systems can be made practical by separating the clocking or timing functions from the main state machine. This eliminates the need to add large numbers of states to the state machine to keep track of elapsed time. The timers or clock dividers can generally be implemented as fairly simple counters that generate an output pulse after a certain number of clock cycles.

Usually external control inputs or sensor inputs are not synchronized to the system clock. This can occasionally lead to unpredictable system behavior for reasons to be explained in the next few slides.

This diagram is just a starting point. There are many ways this structure can be expanded:

1. Complex timing requirements may require numerous timing signals that are triggered under a variety of conditions.

2. External sensor inputs (or external outputs) may require entire control systems just to accommodate the I/O timing requirements and protocols of the external devices.

3. In additional to external devices, the main control unit state machine outputs might be used to control other computational circuits such as memory, arithmetic logic units, multiplexers, etc. You will see this approach applied if/when you take a computer architecture course.
Common elements

- **Main controller**
  - state machine that directly controls sequence of operations

- **Clock divider (a state machine)**
  - continuous counter produces pulse at uniform slow interval
  - needed because high frequency clocks are easier to produce

- **One or more timers (more state machines)**
  - counts off a time interval and generates a pulse when done
  - may be continuous running or may use a restart signal
  - most systems require different time intervals for different purposes

- **Synchronizers**
  - Asynchronous inputs can arrive near clock edge, making next state unstable at clock edge, unpredictable results
Think back to your introduction to digital logic design course. You should have learned that two critical timing specifications for a flip-flop (or register) are setup time and hold time. Setup time specifies that the data input to a flip-flop must be stable (not changing) for a certain period of time prior to the active clock edge. Hold time specifies that the data input must be stable a certain amount of time before the active clock edge. Violation of setup or hold constraints will sometimes (not always) result in a flip-flop that latches an indeterminate signal level rather than a logic 1 or 0.

Note: a metastable state is a state that is only stable in a weak sense – a little bit of noise is enough to push a flip-flop to switch to a 0 or 1 state. It is a bit like a ball at the peak of a hill – a slight nudge is enough to cause it to roll down the hill on one side or the other.
This example presents a deliberate violation of setup and hold constraints for the left most flip-flop in the synchronizer circuit shown. As a result, the left flip-flop is left in a metastable state after the first clock edge. Within a fairly short time (less than the minimum clock period for a given technology) the flip-flop will switch to a 0 or 1 state with an extremely high probability. Consequently, when two flip-flops are chained together as above, the chances are exceedingly small for a metastable state on the right most flip-flop. Thus this synchronizer prevents indeterminate states from propagating into the rest of the system.

This synchronizer design is really only suitable for single bit asynchronous inputs. Multiple bit inputs such as a data bus require additional steps to ensure that changes to a parallel input are all latched on the same clock edge. Otherwise, garbage values may be latched in a register for one cycle. The Wakerly text includes a much more in depth treatment of synchronizer design.
Controller design rules

- All components must use system clock as their “clock” (easier to test, less susceptible to glitches)
- Timer and clock divider outputs should only be connected to the next state logic of other blocks

```vhdl
if timerOut = '1' then
    -- execute normal next state logic
else
    nextstate <= state;
end if;
```

It is very tempting to try to clock registers using outputs from a time circuit, but this ultimately will make your design much harder to debug if there is an error (for the same reasons why it is generally bad practice to gate the clock with some other logic signal).

Instead of feeding the timer signal directly to the clock input of a register, make it an input to the next state logic for that register – most registers in your designs will have some kind of next state logic in front of them anyway. In the code fragment above, when `timerOut = '0'`, the next state logic is forced to feed the register output right back to its own input. This assumes that there is a register with `nextstate` as the input and `state` as the output. Only when `timerOut = '1'`, is the register allowed to change state. This approach makes it easy to use a timer to control when state changes are allowed to occur in a state machine.
Design tips

To divide down by a large number
- chain multiple clock dividers together
- each clock divider must obey previous design rules

Make the timer and clock divider outputs be 1 system clock cycle long
- saves you from having to do some kind of edge detection on timer outputs

Sometimes a state machine will need to change states at a uniform rate that is slower than the available clock signal. A counter with appropriate output logic can be used to repeatedly produce pulses of one cycle in duration that are separated by some multiple of the clock period. Occasionally, the delay between pulses is a very large multiple of the clock period. Rather than coding a single very large counter, it may be more reasonable to use one small clock divider to enable/disable state changes in a second clock divider. Multiple clock dividers may be chained in this way to achieve very long delays between consecutive pulses.

It is fairly easy to make the output of a timer to be just one clock cycle long. Take a simple counter and specify output logic that only produces a ‘1’ when the count is equal to a certain value. If the counter increments every clock cycle, then the resulting output will on be one clock cycle in duration (actually, somewhat less, but that is ok).
Sensitivity Lists Revisited

- Simple minded, safe approach
  - identify all inputs to the process
    - signals appear on right hand side of <= or :=
    - signals appear in conditional expressions
      - if rst='1' then etc.
      - case Sel is
        - when “00” => Y <= d0; etc.
  - Always do this for combinational logic
- Never include output in sensitivity list
  - can lead to infinite iteration
  - may imply combinational feedback loop

Never include output in sensitivity list
This process represents a purely combinational logic circuit (there is no clock or data storage) so all inputs MUST be listed.
Sensitivity Lists Revisited

A more sophisticated approach

- only applies to sequential circuit blocks
  - changes triggered on edge of clock
  - i.e., flip-flops or registers
- identify asynchronous inputs & clock
  - all inputs to signal assignments and conditional expressions that are NOT inside of a rising_edge(), falling_edge(), or clk'event branch
REGS: process(Clk,Rst)
    begin
        if Rst='1' then
            state <= NUM0;
        elsif rising_edge(Clk) then
            state <= nextstate;
        end if;
    end process;

-- nextstate could be included but not needed because
-- it is only used when process is triggered by clock edge
Big Mistake

Leaving signals out of sensitivity list
- leads to differences between source code and mapped (synthesized) version

Why?
- Sensitivity list, when used wisely, makes for more efficient source code simulation
- Synthesis ignores sensitivity list
- Missing signals change source code simulation but not synthesized version

Bigger Mistake:
- hacking on the sensitivity list to make your simulation work (BAD BAD BAD)

Sometimes people discover that by cleverly (ha!) leaving signals out of the sensitivity list, they can make the source code version simulate in a desirable manner. However, as stated previously, the sensitivity list is just a tool for making simulation more efficient, and is ignored by synthesis. The resulting synthesized (or mapped) version can almost be guaranteed not to work.
Mising nRst from sens. list

FF: process(Clk)
    begin
    if nRst='0' then
        q <= '0';
    elsif rising_edge(Clk) then
        q <= d;
    end if;
    end process;

For source code simulation, simulator only evaluates this code at instant that Clk changes

When simulating this source code, nRst will only be evaluated on the rising clock edge, making nRst behave like a synchronous reset. However, the synthesis software will look at the if-then-else construct and determine that nRst is in fact supposed to be asynchronous. The following slide illustrates the behavior.
Effect of missing signal

nRst = 0 not noticed by simulator till clk changed

nRst = 0 ignored by simulator
VHDL Libraries and Packages
VHDL Libraries & Packages

- Overview in Wakerly sec 4.7.5
- Have been using all along
  - for small designs & 1 or 2 designers, could get away without understanding very well
  - important for organizing large designs with many designers
  - important for re-using design data
- For test, make sure you understand purpose of libraries/packages and how to use `library` and `use` statements
Libraries

- A place for storing compiled (or synthesized) VHDL code
- By default, everything goes in **work** library, located in \./work (this is why you didn't have to understand libraries very well)
- Don't need a “library” or “use” statement in VHDL code to use components from **work** library
Creating Libraries

For modelsim

- vlib foolib -- creates library directory
- vmap logical-name library-directory
  - vmap foo ./foolib
  - adds entry to modelsim.ini specifying that library foo is located in ./foolib
- vcom -work ./foolib sourcefile.vhd
  - compile sourcefile.vhd & put result in ./foolib
- in source code, need
  - library foo; -- gives access to entities architectures
  - use foo.all; -- gives access to all type definitions etc.
Creating Libraries

For Synopsys design Compiler

- in .scr file or .synopsys_dc.startup...
- define_design_lib foo -path ./foolib
  - specifies library name foo located in ./foolib
- analyze -format vhdl -lib foo
  {sourcefile.vhd}
  - analyzes (~ compiles) sourcefile.vhd & puts result in library foo
- in VHDL code, use “library” and “use”
Packages

Platform packages:
- define types, global signals, constants, functions, procedures, component declarations

To use a package:
- use lib-name.package-name.object-name
  - e.g., use ieee.std_logic_1164.std_ulogic
    - means, use signal type “std_ulogic” from package “std_logic_1164” in library “ieee”
Packages

To create a package

- package foopack is
  - -- type, signal, constant, component, function,
  - -- and procedure declarations that you want to
  - -- use in other vhdl source code
- end foopack;
- package body foopack is
  - -- local declarations,
  - -- function/procedure definitions
- end foopack;

Compile or analyze into target library
Generics

- Make entities more general purpose
  - Allows you to specify input parameters
- Useful for synthesis
  - Example: variable width register
- Useful for simulation
  - Example: variable delay component
Variable width register

Entity declaration

definition

entity dffr is
  -- parameter name wid, type is integer
  -- synopsys only takes integer (not “positive”)
generic (wid: integer := 8); -- default of 8
port (Rst, Clk: in std_logic;
  -- keyword signal optional here
  signal D: in std_logic_vector(wid-1 downto 0);
  signal Q: out std_logic_vector(wid-1 downto 0));
end dffr;
architecture behavior of dffr is
begin
process(Rst,Clk)
variable i: integer;
begin
if Rst = '1' then
  Q <= (others => '0'); -- notice this trick!
elsif Clk = '1' and Clk'event then
  for i in Q'range loop -- another trick!
    Q(i) := D(i);
  end loop;
end if;
end process;
end behavior;
Using a “generic” component

architecture sample of reg is
component dffr
  -- even with default specified in entity,
  -- vcom unhappy unless given here or in instance
  generic (wid: integer := 8); -- synopsys only allows integer
  port (Rst,Clk: in std_logic; -- keyword signal optional here
       signal D: in std_logic_vector(wid-1 downto 0);
       signal Q: out std_logic_vector(wid-1 downto 0));
end component;

constant WID16: positive := 16;
signal D8,Q8: std_logic_vector(7 downto 0);
signal D16,Q16: std_logic_vector(15 downto 0);
begin
  FF8: dffr port map(Rst,Clk,D8,Q8); -- uses default of 8
  FF16: dffr generic map(wid 16) port map(Rst,Clk,D16,Q16);
end sample;
Bidirectional Bus
Bidirectional bus using tri-state

Device 1

Transmitter

Receiver

Device 2

Receiver

Transmitter

xmt1

xmt2

!xmt1

!xmt2

serial

tri-state buffer
Tri-state buffers

Basic concept of operation:

How is it done in CMOS? (some alternatives)

in each case, when ena=0, out is left floating
Bidirectional bus using tri-state

What happens if:

- \( \text{xmt1} = 1, \text{xmt2} = 0 \) (1 -> 2)
- \( \text{xmt1} = 0, \text{xmt2} = 1 \)
- \( \text{xmt1} = 1, \text{xmt2} = 1 \) (bad)
- \( \text{xmt1} = 0, \text{xmt2} = 0 \) (ok)
Bidirectional bus using tri-state

What happens if:

- $xmt1 = 1$, $xmt2 = 0$
- $xmt1 = 0$, $xmt2 = 1$ (2 -> 1)
- $xmt1 = 1$, $xmt2 = 1$ (bad)
- $xmt1 = 0$, $xmt2 = 0$
Bidirectional bus using tri-state

What happens if:
- $xmt_1 = 1$, $xmt_2 = 0$ ($1 \rightarrow 2$)
- $xmt_1 = 0$, $xmt_2 = 1$ ($2 \rightarrow 1$)
- $xmt_1 = 1$, $xmt_2 = 1$ (bad)
- $xmt_1 = 0$, $xmt_2 = 0$
Bidirectional bus using tri-state

What happens if:  

- $xmt_1 = 1, xmt_2 = 0$ (1 -> 2)  
- $xmt_1 = 0, xmt_2 = 1$ (2 -> 1)  
- $xmt_1 = 1, xmt_2 = 1$ (bad)  
- $xmt_1 = 0, xmt_2 = 0$ (floating, bad)
Coding a tri-state bus

serial should be inout port

-- tri-state buffer code
if (xmt1='1') then
  serial <= serout;
else
  -- ‘Z’ disconnects output
  serial <= ‘Z’;
end if;

-- prevent reading from own xmitter
serin <= serial and not xmt1

note: serial should probably have a weak pull-up (large resistance) to avoid undefined state if serial line is floating
Use of ‘H’ or ‘L’

‘H’ or ‘L’ is equivalent to pull-up or pull down resistor at output of tri-state

When ena=‘0’, output is a weak ‘1’ (called ‘H’ in vhdl)

When ena=‘1’, buffer output overpowers the pull-up resistor and forces output to near ‘0’ or to ‘1’

you might use this in a test bench, but since our synthesis libraries don’t support this, you won’t use it for synthesis
Reminder!

To improve chances of synthesis
- Think of each block of code as a circuit rather than as code.
- Until proficient, stick to styles and syntax from this course.
- Major difference from sequential programming languages,
  - Order of concurrent statements has nothing to do with order of execution!
- Use data flow whenever possible
- Keep your behavioral blocks as simple as possible
- Most code has to translate into combinational logic
  - (except where you intentionally create latches or registers)