Module #9: Basic Digital System Design

Revision: October 22, 2004

Overview

This lab presents several more involved circuit designs that build on the experience and knowledge you've gained in previous labs. But this lab also goes a bit further by requiring an additional step at the beginning of the design process – before you can start working towards a design solution, you must develop a clear understanding of the design requirements. In previous exercises, requirements were provided with a fair amount of detail and guidance, in order to emphasize certain circuits or design methods. But going forward, design problems will be more open-ended, and you will need to spend more time thinking and learning about the design before starting any specific tasks. As you will come to appreciate, the time spent in gaining a clear understanding of the requirements is inversely related to the time spent in designing (and redesigning) circuits.

In attempting to solve more general design problems, you will find increasing benefit in establishing and following "good design practices" (or GDPs). Simply stated, GDP's are guidelines that provide methodical plans for creating new designs. Designs that follow GDPs are easy to document, easy to follow, and easy to retrace. Engineers that adopt GDPs are typically far more productive than those that use the more common "iterate, try, and hope" method. This lab encourages you to begin trying to identify and use GDPs.

Before beginning this lab, you should:

- Be able to design and simulate both structural and behavioral circuits in the Xilinx ISE/WebPack tool;
- Be able to download circuits to the Digilab board using the Xilinx WebPack tool;
- Be familiar arithmetic circuits and the bit-slice design method.

After completing this lab, you should:

- Be comfortable in approaching more complex design problems;
- Understand the value of partitioning a design properly;
- Appreciate the utility and trade-offs in top-down vs. modular design methods
- Understand the role of ALU circuits in more complex digital systems, and be able to design a simple ALU circuit.

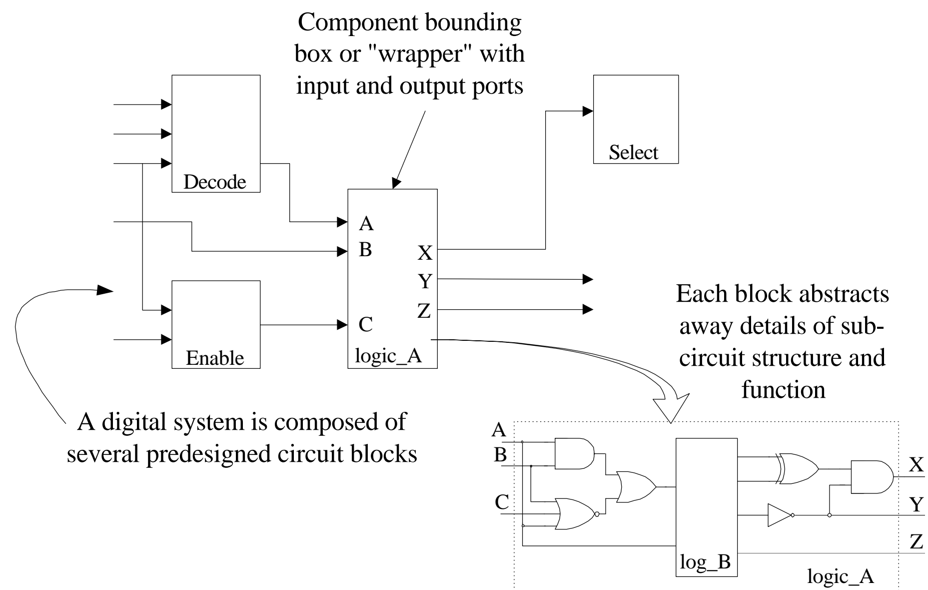
This lab exercise requires:

- A windows computer running Xilinx WebPack 4.2 or later;
- The Digilab XCRP board.

Background

Digital System Design Approaches

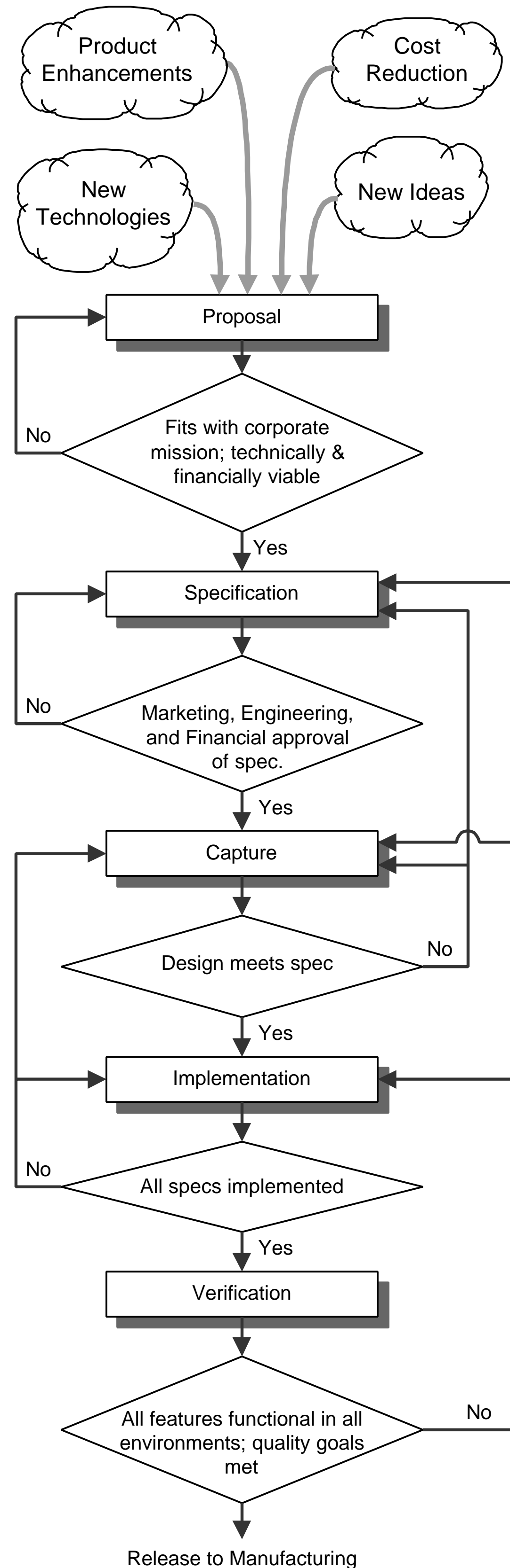
A digital system is a larger circuit that is composed of smaller, independent and/or pre-existing sub-circuit blocks. Because they are more complex, digital systems typically require a more disciplined design effort. The sub-circuit blocks used in a digital system are designed and simulated separately, and then “wrapped” inside a component or macro shell so that only a bounding box and the input and output ports (or pins) are visible. Finished components are stored in a project library, and from there they may be added to a larger system design as needed. System designs must be partitioned early in the design phase into sensible sub-circuit blocks. A good partition can make a design flow very naturally, and significantly lessen the amount of work required. A poor partition can make an already difficult design nearly impossible to complete. Any given design can be partitioned in a number of ways, and the “best” partition for one designer may not be the best for another. Experience and personal design style contribute to such design decisions. In general, a design should be partitioned around natural functional boundaries, and each component should have a clearly defined and limited function. After gaining more experience designing digital systems, the job of partitioning a design, although always a judgment call, will become more intuitive.



Digital systems can be designed following the top-down approach, the bottom-up approach, or (more typically) and combination of the two. In the **top-down** approach, only knowledge of the specification and block diagram are assumed. The top-level representation is divided (or partitioned) into smaller and smaller pieces, such that at any stage the current set of partitions can be re-assembled into the top-level circuit. The partitioning process continues by creating ever more detailed circuit/block diagrams, until the design has been reduced to an interconnection of simple (or at least familiar) components. Choosing proper partition boundaries represents the largest challenge: a well-crafted partition can make a design flow smoothly; a poor partition can create unnecessary work and lead to failure. In general, partitions are chosen so that circuit requirements are mapped to pre-existing or well-known circuit blocks, or to smaller, readily defined “black boxes” whose contents can be readily designed at a later point. A good partition uses circuit blocks that can simply be inserted from a design library, or that can be described in just a few words (like “input decoder” or “control sequencer”). A partition should show all signals that must be routed between all circuit blocks and to the top-level inputs and outputs. The process is complete when the design can be readily implemented in a given design tool.

In the **bottom-up** approach, the designer typically has a list of components that will be used to implement the design. The design partitioning process is then dictated by the list of available components rather than by the criteria mentioned above. With this approach, the lowest level circuit blocks are added to the design first, and the design takes form as these smaller blocks are interconnected into the growing circuit.

In practice, a mix of bottom-up and top-down approaches are used, and this hybrid method can be termed the **modular** design method. For more involved designs, it can be very advantageous to associate particular design requirements with pre-existing and well-known circuit blocks (such as muxes, decoders, adders, etc.). By identifying and associating such requirements relatively early in the design, you can focus attention on those features that require custom treatment. A new component must only be designed when a sub-circuit block is identified for which no pre-existing component exists. Using the modular method has several advantages: less circuitry must be designed, which saves time; the circuit is usually divided into smaller and well-understood modules, which makes it easier to debug; and since other engineers are also familiar with common modules, the design is easier to document.



The New Product Design Process

The New Product Design Process

Ideas for new products can arise from any one of a number of motivating circumstances, from any one of a number of sources. As examples, most businesses are continually looking for opportunities to cut production costs, or to gain competitive advantage by exploiting new technologies or ideas; and personnel from marketing, engineering, sales, or any other department within a company might originate a new product idea. Any new product idea, whatever its source, must typically pass through the challenges and critiques of a formal **proposal** phase. Ideas that fit well within a company's mission, and that are judged to be technically and financially sound, can emerge from the proposal phase and graduate to the next stage of development. Many more product ideas are rejected than are accepted, which helps ensure that ideas accepted for further development do indeed offer the promise of financial returns.

Ideas that emerge from the proposal phase typically enter a **specification** phase. In this phase, the design intent,

goals, features, schedules, resource requirements and risk factors are all identified. Preparing a proper specification is the most important and most difficult design challenge in the new product design process, and this duty is typically entrusted to more experienced personnel. Most often, a small team comprised of engineering, marketing, and business development personnel develops the specification. In lab exercises, designs projects are smaller and well bounded, so specifications are minimal and provided as a part of the exercises. In later classes, or early in your engineering career, you will gain exposure to product specifications and their creation.

Once a product idea has been described in a detailed specification, all of the important features have been identified, and the product can enter the **design** (or **capture**) phase. Beginning with this phase, the product development effort enters the engineering process. The engineering process starts with a clear specification, and ends with a finished design (and probably one or more prototype products) that can be passed to the manufacturing department.

The Engineering Process

The engineering process can be broken down into the three major phases of capture, implementation, and verification. In the design **capture** phase, the requirements put forth in the specification are synthesized into a circuit using any of several design approaches, methods, and CAD tools. Once captured, a circuit can be simulated and verified before it is implemented. In a typical design capture phase, larger circuits are decomposed into smaller, well-defined blocks that may be implemented and verified independently. These smaller, pre-verified blocks are then assembled into a final circuit at some later point. In the **implementation** phase, the captured design is constructed using "target" hardware. Often, the target hardware is simply meant to serve as a circuit proto-type or test bed so that more design and verification activities can proceed using physical (instead of simulated) circuits. After a new design has been implemented, it enters the **verification** phase. In this phase, detailed simulation studies are used to demonstrate design performance under all conceivable input conditions. If a prototype was built, then the physical device can be tested, measured, challenged, and verified. Note that the verification process can result in changes to the product specification, the product design, or the implementation.

The engineering design process is applicable to a new project or design requirement at any level (from a first assignment in an entry-level engineering class through a newly conceived, complex product). When working on the lab exercises, try to remain aware of the phases of the engineering process. In particular, be aware when you are crossing the "spec-to-capture", "capture-to-implementation", and "implementation-to-verification" phase boundaries. At each boundary crossing, slow down, challenge your assumptions, and ensure that you are meeting all requirements up to that point.

Good Design Practices

The engineering process begins with a clear specification, and ends with a circuit that is described in a format that can be directly implemented in a given technology (for our purposes, we will consider the design process "finished" when we have a detailed circuit definition that can be downloaded to a Xilinx chip for verification). In between, you must digest the design specifications and the design intent, create engineering formalisms like truth tables and block diagrams to precisely describe the problem, create a circuit that meets the specifications, and verify that the circuit works correctly. For most any design challenge, simple and elegant solutions are possible, but so are complex and confusing solutions. Which solution you develop will depend on many factors, including your

understanding of the requirements, your ability to associate the requirements with designs and methods that you are already familiar with, and your design practices.

A GDP can use any of the design methods presented in previous labs, including the “first principles” method, where circuits are developed from truth-table definitions to meet a specific need; the “bit slice” method, where circuits that operate on binary numbers are broken down into bit-wise operations; and the “behavioral” method, where a circuit’s behavior is stated in the vernacular of VHDL. The first-principles method is useful when designing simpler circuits that combine inputs from independent sources like switches and sensors, or for circuits that produce outputs to drive simple devices like displays. The bit-slice method, which forfeits efficiency for tractability, is useful when designing circuits that operate on signals grouped into busses (it is too difficult to design bus-wide circuits using the first-principles method). Behavioral methods are generally applicable to any design, and they can greatly reduce the amount of design effort required. But since behavioral methods abstract away virtually all structural detail, a designer’s ability to accurately simulate and model a circuit’s performance is limited. Whichever of these methods is used, it is only a part of the GDPs required to create a successful design.

Developing a GDP that works for you takes experience, and you are encouraged to seek input from colleagues or more senior engineers (or lab assistants) along the way. GDPs have certain key features: they start with a clear understanding of the problem statement and/or design requirements; they proceed in measured steps that do not require any assumptions or leaps; and they are retraceable. These features are based on several points of collective wisdom, namely: the more time spent gaining a clear understanding of the design requirements, the less time and effort spent on the design; any assumptions or “best-guesses” in the design process betray a lack of clarity; and since designs rarely proceed directly from initial concept to flawless implementation, it should be possible to retrace design steps to a decision point from which an alternative approach can be attempted. In developing design practices that work for you, you are encouraged to adopt these minimal guidelines:

1. Read, re-read, and read again the design requirements, and extract as much understanding of every point, no matter how small it seems, before proceeding.
2. Create a **first-cut partition** of the design in the form of a detailed block diagram, and label all circuit blocks and all interconnects. At this point, some blocks may be well known circuits like decoders, multiplexors, adders, etc., but some may be “black boxes” whose I/O requirements can be defined, but whose circuits are left to later. At this point, it may be helpful to describe any “black box” requirements in a truth table or other engineering formalism.
3. Measure the block diagram against the design requirements, challenge all assumptions, and ensure that all requirements have been addressed.
4. Create a **design-level partition** that contains existing or readily designed circuit blocks, and ensure that all design requirements are still met. This will be the blue-print for the capture phase, so this partition must be very clear and precise.
5. Capture the circuit blocks requiring new designs one at a time, and simulate each one so that you have confidence in its performance. Give meaningful names (or labels) to new circuit blocks, and associate these circuit blocks with the information in your design notes that was used to create the circuit (e.g., comments, state diagrams, timing diagrams, etc.). Always be sure to use generous comments in all source files, including schematics.
6. For all but the simplest designs, store completed circuit blocks in a separate project library so that older “work” versions of circuit blocks cannot get mixed up with current “finished” versions.

7. When all required sub-circuit blocks are available, begin assembling them into the larger circuit. Rather than adding all blocks at once, identify smaller combinations of circuit blocks (both designed and added from a library) that can be added to the design and simulated. It is much easier to capture, simulate, and debug an incrementally growing circuit, rather than the entire thing.
8. Continue adding blocks and simulating until all blocks are added and all design requirements are met.
9. When the circuit is complete, re-read the design specifications, and challenge all of your design decisions. Make sure you believe the design that you have created meets the design intent. If you are unsure, seek input from the lab assistant (or fellow students); if you remain unclear, add a note to your submission stating the issue, and your particular solution (generally, no points will be deducted if you state your case well).
10. Prepare all required documentation, and ensure all design files are safely stored in a stable directory.

A Typical Design Process

As always, try to gain a thorough understanding of the design specification and design intent before undertaking any formal design activity. A new design typically starts with several block diagram sketches, first of the overall system, and then of smaller blocks that show more and more functional detail. When a suitably detailed block diagram has been created, pause and rethink all of your assumptions and decisions, and challenge the block diagram and your understanding of the design. Try to visualize the finished design in operation, and try to imagine all the ways it might fail. If you are still confident in the design, then keep going forward. Remember that the initial design activity is the most crucial to your understanding of the problem, and therefore to your eventual solution to the problem. It is not an activity to hurry through in order to meet the schedule requirements of an assignment – it is, rather, your chance to learn very clearly what you do and do not know about the problem.

As a next step, try to associate particular circuit requirements with known, pre-existing circuit blocks. Identify any and all circuit functions that map exactly (or very nearly) to the kinds of functions that might be found in a library (comparators, adders, decoders, etc.). Once identified, add those components as blocks to the evolving circuit/block diagram. For any blocks that require custom circuits, perform the design activities required to arrive at a suitable circuit. The circuit/block diagram will very likely need to change as you gain more knowledge and insight about the problem during the course of the design. Eventually, after you have formed a clear picture of exactly what components and circuits are required, you can begin to design the circuit.

Once a circuit has been designed, it can be implemented and then verified and debugged. Verification and debug time is usually inversely proportional to design time – the less time spent designing, the more time spent debugging. It is extremely rare that a design is implemented correctly the first time, which is why so many powerful test and measurement tools exist (logic analyzers, oscilloscopes, etc.). At the very least, some unanticipated behavior always turns up after you have the chance to interact with a newly designed circuit. Discoveries made in the debug stage often require design modifications that, depending on your choice of design methods, can be relatively easy or vastly difficult.

Since you must implement and test your own designs, the benefits to properly structuring the design in the first place are obvious: you get to do less overall design work; the work you do perform is targeted more directly at solving the required problem and not at designing/debugging some dubious circuit; and when simulating and/or verifying, only circuits and signals performing essential functions need be

considered. Since it is exceeding rare to produce a working design during the first several attempts, especially when you are in the learning mode, the importance of adopting solid, well-thought and methodical design practices cannot be overstated.

Time spent on a typical design problem might be proportioned as follows:

- Study and fully comprehend design specification and intent 15%
- Detailed block diagram with all components identified 15%
- Implementation of circuit 25%
- Simulation 20%
- Download and verification 10%
- Documentation 15%

Note that about 1/3 of the overall time is spent before any design tool (other than your brain and pencil and paper) is used. Note also that nearly 1/2 of the time is spent after the circuit has been implemented.

Adder/Subtractor Circuits

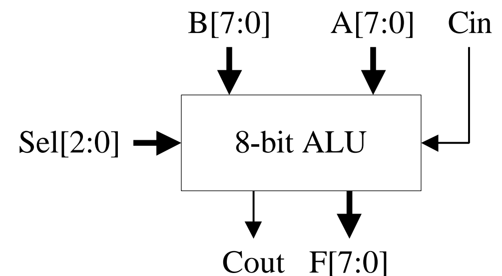
An adder circuit can be easily modified to create a 2's compliment subtractor. Recall that if numbers are represented in 2's compliment form, then one number can be subtracted from the other if the number to be subtracted (the subtrahend) has all bits inverted, and then '1' added to the inverted-bit result (i.e., A – B can be written as (A + (B' + 1)), where B' means invert all bits in the B vector). To modify an adder circuit to perform either A + B or A – B, an accessory circuit can be added to the B input to invert all the B bits and add '1' whenever a control input is asserted (recall that a "1" can be added simply by driving the adder carry-in to a '1').

Problem 1. Sketch a simple two-input, one output circuit that can invert an input A to produce a single output A or A' based on the logic state of a select input.

Problem 2. Sketch a circuit that can add or subtract two 8-bit vectors A and B based on the logic state of a select input.

ALU Circuits

Arithmetic and Logic Units (or ALUs) are found at the core of microprocessors, where they implement the arithmetic and logic functions offered by the processor (e.g., addition, subtraction, AND'ing two values, etc.). An ALU is a combinational circuit that combines many common logic circuits in one block. Typically, ALU inputs are comprised of two N-bit busses, a carry-in, and M select lines that select between the 2^M ALU operations. ALU outputs include an N-bit bus for function output and a carry out.



ALUs can be design to perform a variety of different functions. A typical ALU includes several useful arithmetic functions and several useful logic functions. Which functions are included in a given ALU depends on the application. The 8-bit ALU design presented in this lab, although rather simple, is not unlike many of ALUs that have been designed over the years for all sizes and performance ranges of processors. Our ALU will feature two 8-bit data inputs, an 8-bit data output, a carry-in and a carry out,

and three function select inputs (S2, S1, S0) providing selection between eight operations (four arithmetic and four logic).

The targeted ALU operations are shown in the operation table below. Note the three control bits used to select the ALU operation are called the "operation code". This is in reference to the fact that if this ALU were used in an actual microprocessor, the control bits would come from the "opcodes" (or machine codes) that form the actual low-level computer programming code. (Computer software today is typically written in a high-level language like "C", which is compiled into assembler code. Assembler code can be directly translated into machine codes that cause the microprocessor to perform particular functions).

Since ALUs operate on binary numbers, the bit-slice design method is indicated. ALU design should follow the same process as other bit-slice designs: first, define and understand all inputs and outputs of a bit slice (i.e., prepare a detailed block diagram of the bit slice); second, capture the required logical relationships in some formal method (e.g., a truth table); third, find minimal circuits (by using K-maps or espresso) or write VHDL code; and fourth, proceed with circuit design and verification.

Operation Code	ALU function
000	A PLUS B
001	A PLUS 1
010	A MINUS B
011	A MINUS 1
100	A XOR B
101	A'
110	A OR B
111	A AND B

Problem 3. Sketch a block diagram for a bit-slice ALU circuit based on the operation table shown.

Problem 4. Sketch a 4-bit ALU composed of 4 of your bit slice blocks.

Problem 5. Transfer the ALU requirements to the truth table in the submission form.

Problem 6. By referring to the truth table, complete the K-maps for the functions F and C_{out}.

From this point, we could proceed in several ways to create a bit-slice circuit:

- minimum circuits could be looped from the EV K-maps;
- 8:1 muxes could be used to implement the circuit outputs directly;
- espresso could be used to find a minimal circuit for this six input, two output logic circuit;
- or we could forget all the structural work and code a VHDL behavioral model.

We will briefly investigate these methods.

Problem 7. Loop the K-maps of problem 6, finding minimal expression in each case (hint: note the points awarded for this exercise, and spend your time accordingly).

Problem 8. Sketch a single bit-slice circuit using a mux for both F and Cout. For Cout, use a 4:1 mux with an enable.

The above problems illustrate starting points for completing structural designs. Each of the methods involves a fair amount of detailed work. The ALU can also be coded behaviorally in VHDL,

and this exercise becomes very simple if two VHDL features are used: the "selected signal assignment" statement, and the standard logic "arithmetic" library package. Both are discussed below.

Selected Signal Assignments in VHDL

```
With sel select
  Y<= A when "00",
      B when "01",
      C when "10",
      D when others;
```

A 4:1 mux using a selected signal assignment

When a single output or bussed output must take on a value based on the state of some number of select inputs, a conditional assignment statement can be used to minimize and clarify the VHDL code. A selected signal assignment can assign one of 2^N possible outputs based on the state of N select bits. As a simple example, a multiplexor can easily be coded using a selected assignment statement as shown. In this example statement, the input sel must be of type STD_LOGIC_VECTOR(1 downto 0); and the Y, A, B, C, and D signals can be vectors or individual signals. The statement functions by comparing the value of the sel input to the value shown in the **when** clause: the output variable Y gets assigned A, B, C, or D depending on whether sel = "00", "01", "10", or "11" (in a selected signal assignment statement, "when others" is used for the final case "11" for reasons that will be explained later). In addition to assigning values to individual signals or busses, the selected signal assignment statement can also be used to assign the result of arithmetic and/or logic operations to an output.

Simple arithmetic in VHDL

Any VHDL source file can use previously written functions and procedures that have been stored in a library. The IEEE library, included as a part of every VHDL tool, contains functions that allow arithmetic to be performed on standard logic signals and vectors. To use these functions, you must make them visible to the VHDL analyzer by including a **library** statement and an **use** statement at the top of your VHDL source file. Since you will typically use the two statements

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

at the top of every VHDL source file, you must add only a second "use" statement to access the arithmetic functions:

```
use IEEE.std_logic_arith.all;
```

After this statement has been added to your source file, addition ('+'), subtraction ('-'), multiplication ('*'), and division ('/') operators can be used with STD-LOGIC types. For example, two vectors can be added by writing a statement like "Y<=A + B;" (assuming Y, A, and B are all standard logic vectors of the same size). The use and contents of libraries will be covered in more detail later.

Behavioral ALU example

The code on the right shows a working example of an 8-bit, four function ALU. By studying this example, you can learn one way to code an ALU or similar circuit. This code can easily be modified to create a more complex ALU.

Example VHDL code for an 8-bit, four function ALU

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

entity MY_ALU is
  port ( A, B: in STD_LOGIC_VECTOR (7 downto 0);
         sel: in STD_LOGIC_VECTOR (1 downto 0);
         Y: out STD_LOGIC_VECTOR (7 downto 0));
end ALU;

architecture ALU_arch of MY_ALU is
begin
  With sel select
    Y <= (A + B)          when "00",
         (A + "00000001") when "01",
         (A OR B)         when "10",
         (A AND B)        when others;
end ALU_arch;
```

Problem 9: Design and implement a 4-bit ALU using the Xilinx VHDL tools and the Digilab board. The ALU must perform the operations shown in the operation table presented earlier.

Problem 10. Design the digital system described in the submission form.

Problem 11. Design the digital system enhancement described in the submission form.