Module 7: Combinational Arithmetic Circuits

Revision: August 31, 2009

215 E Main Suite D | Pullman, WA 99163
(509) 334 6306 Voice and Fax

Overview

This module examines several combinational circuits that perform arithmetic operations on binary numbers, including adders, subtractors, multipliers, and comparators. Arithmetic circuits typically combine two or more data busses of 8, 16 or 32 bits to produce outputs that use similar sized busses. They present special design challenges, because there are simply too many inputs to list all possible combinations in a truth table (for example, a circuit combining two 8-bit busses would require a truth table with 2^{16} or 65,0000 rows). This module introduces a divide-and-conquer design method known as the "bit slice" method that is well suited to arithmetic circuit design, as well as to any other circuit that operates on binary number inputs. In applying this method, bus-wide operations are broken into simpler bit-by-bit operations that are more easily defined by truth-tables, and more tractable to familiar design techniques. The major challenge lies in discovering just how a given problem can be decomposed into bit-wise operations.

This lab also introduces structural VHDL design, which closely parallels schematic-based design in concept and in method. Structural VHDL designs are hierarchical, with high-level designs constructed from smaller, independently designed VHDL entity-architecture design units. As with schematic design, signals from lower-level design units can connect to overall circuit input and output signals, or to internal signals that are not visible outside the current level of hierarchy. Special VHDL statements are used to declare and instantiate components, and to define internal signals. Structural methods are often used on larger designs where pre-existing circuit blocks might already exist, or when detailed simulation studies are required.

Special attention is focused on a circuit that combines many previously encountered designs and techniques. This circuit, known as an "arithmetic and logic unit", or ALU, is found at the core of computing circuits. At first glance, an ALU design seems complex and involved, but as you will see, the design is actually a straightforward application of circuits and methods already encountered. The challenge lies in thoroughly understanding the design problem before beginning any design activities, and in following a disciplined, step-by-step design approach.

Before beginning this lab, you should:

- Be able to add, subtract, and multiply binary numbers;
- Be able to enter and simulate circuits using schematic and VHDL methods in the ISE/WebPack tool;
- Be able to download circuits to the Digilent board;
- Be familiar with design of basic combinational circuit blocks.

After completing this lab, you should:

- Know how to design circuits using structural VHDL methods;
- Know when and how to apply the bit-slice design method;
- Understand how comparators, adders, subtractors, and multipliers work, and be able to design them using schematics or VHDL.

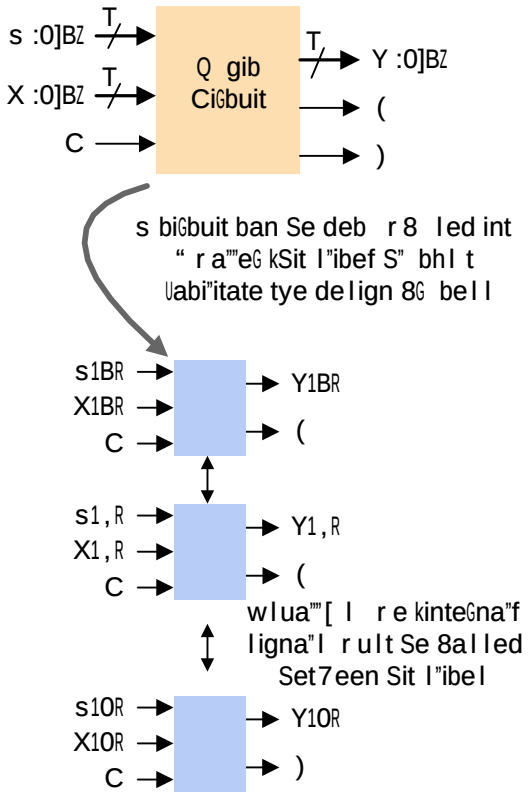
This lab exercise requires:

- A windows computer running the Xilinx WebPack tools
- The Digilent circuit board.

Background

The Bit-Slice design method

When designing circuits with bussed inputs that represent binary numbers, it is often easier to consider a circuit designed for a single pair of bits, rather than for the entire binary number. The reason is straightforward – a truth table describing a circuit operating on two 8-bit busses requires 65K rows, whereas a circuit operating on a single pair of bits requires only four rows. When considering a design for a single pair of bits, the goal is to create a circuit that can simply be replicated N times – once for each bit.

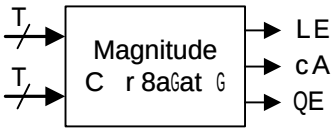


Many circuits that operate on binary numbers can be easily broken down into smaller, bit-wise operations. Some circuits defy this approach, and a bit-by-bit requirements analysis does not indicate any likely bit-slice solutions (e.g., some circuits that convert one type of data code to another fall in this category). Thus, the first goal in applying the bit-slice design method is to determine whether it is possible to express a given problem as an assemblage of bit-wise operations.

In a typical bit slice design, information must be passed between adjacent bits. For example, in a circuit that can add two binary numbers, any pair of bits may generate a “carry out” to the next more significant bit pair. Any such inter-slice dependencies must be identified and included in the design of the bit-slice module. Dealing with these additional “internal” signals may require some additional gates that would not have been needed in a non-bit slice design. But most often, the additional gates are a very small price to pay for enabling a more practicable design approach. All of the designs in this lab will use the bit-slice design approach.

Comparators

A magnitude comparator is device that receives two N-bit inputs and asserts one of three possible outputs depending on whether one input is greater than, less than, or equal to the other (simpler comparators, called equality comparators, provide a single output that is asserted whenever the two inputs are equal). Comparators are readily described in behavioral VHDL, but they are somewhat more difficult to design using structural or schematic methods. In fact, comparator design is an excellent vehicle to showcase the power of behavioral design, and the relative tedium of structural design.



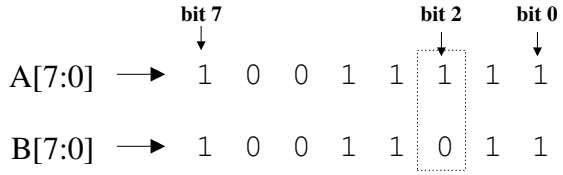
Comparators are described in behavioral VHDL using the greater-than and less-than operators (> and <, respectively) as shown in the code below. Note the "inout" mode used in the port statement for the GT and LT output port signals. The inout mode is required whenever an output port signal also must be used within the architecture on the *right side* of an assignment statement. In next-to-last line in the example, both the GT and LT outputs are used on the right side of the assignment operator to form the EQ output, and thus, the GT and LT signals are *inputs* to the EQ assignment statement. If GT and LT were simply declared as "out" mode types, the VHDL analyzer would generate an error.

```
entity my_comp is
  port ( A, B  : in std_logic_vector (7 downto 0);
         gt, lt  : inout std_logic;
         eq     : out std_logic);
end my_comp;

architecture behavioral of my_comp is
Begin
         gt <= '1' when  A > B else '0';
         lt <= '1' when A < B else '0';
         eq <= not gt and not lt;
end behavioral;
```

A structural comparator design is best attacked using the bit-slice method. Consider an 8-bit magnitude comparator circuit that creates the GT, LT, and EQ output signals for two 8-bit operands. In the illustration below, if A=159 and B=155 are presented to the comparator, then the GT output should be asserted, and the LT and EQ outputs should be de-asserted. The operand bits are equal in all slices except the bit 2 slice. Somehow, the inequality in the bit 2 slice must influence the overall circuit outputs, forcing GT to a '1' and LT and EQ to a '0'. Any bit pair could show an inequality, and any bit-slice module design must work in any bit position.

Clearly, a bit-slice design cannot work in isolation, using only the two data bits as inputs. A bit-slice design must take into account information generated from neighboring bit-slices. Specifically, each comparator bit-slice must receive not only the two operand input bits, but also the GT, LT, and EQ outputs of its less-significant bit neighbor. In the present example, the bit 3 slice in isolation would assert the EQ output, but the inequality in the bit 2 slice should force the bit 3 slice to assert GT and de-assert both EQ and LT. In fact, the outputs from any stage where the operand bits are equal depend on the inputs arising from the neighboring stage.

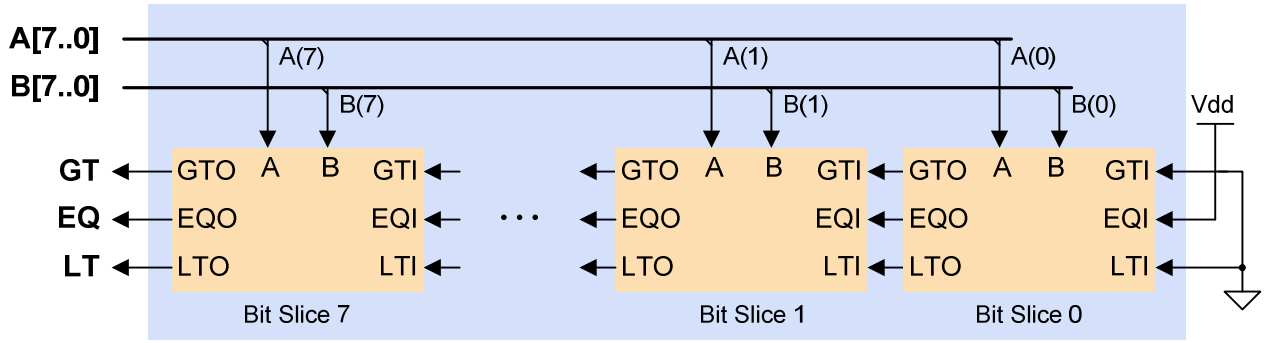


A bit-slice magnitude comparator circuit must have five inputs and three outputs as shown in the truth table. As with any combinational design, the truth table completely specifies the required comparator bit-slice behavior. Normally, a truth table for a five-input function would require 32 rows. The 8-row truth table on the right is adequate because certain input combinations are not possible (i.e., the inputs from the neighboring slice are mutually exclusive), and others are immaterial (i.e., if the current operand inputs show A > B, the neighboring slice inputs do not matter). You are encouraged to examine the truth table in detail, and convince yourself that you agree with the information it contains.

Operand inputs		Inputs from neighboring slices			Bit-slice outputs		
An	Bn	GTI	LTI	EQI	GTO	LTO	EQO
0	0	1	0	0	1	0	0
0	0	0	1	0	0	1	0
0	0	0	0	1	0	0	1
0	1	ϕ	ϕ	ϕ	0	1	0
1	0	ϕ	ϕ	ϕ	1	0	0
1	1	1	0	0	1	0	0
1	1	0	1	0	0	1	0
1	1	0	0	1	0	0	1

The truth table can be used to find a minimal bit-slice comparator circuit using pencil-and-paper methods or computer-based methods. Either way, a bit-slice circuit with the block diagram shown in

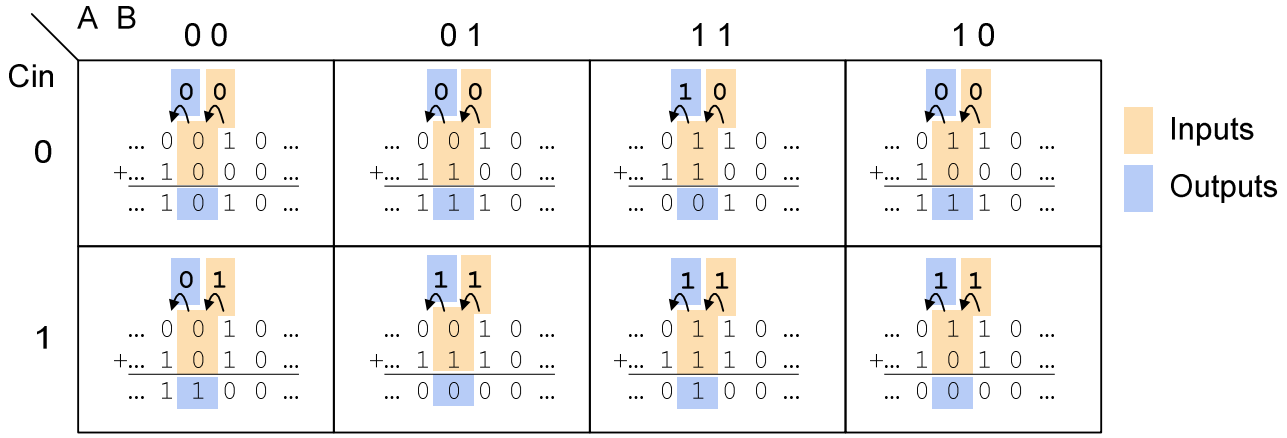
the figure below can be designed. Once designed, a bit slice circuit can be used in an N-bit comparator as shown. Note that for the N-bit comparator, no neighbor bit-slice exists for the least-significant bits – those non-existent bits are assumed to be equal. Note also that the overall comparator output arises from the outputs from the most-significant bit pair. In the exercises and lab project that accompany this module, you are asked to design a comparator bit-slice design as well as an 8-bit comparator circuit.



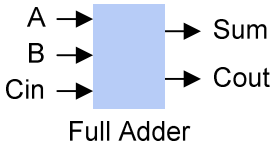
Adders

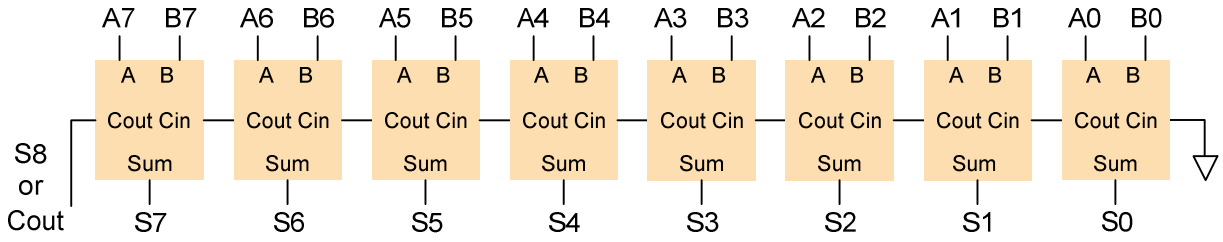
Adder circuits add two N-bit operands to produce an N-bit result and a carry out signal (the carry out is a '1' only when the addition result requires more than N bits). The basic adding circuit is one of the cornerstones of digital system design, and it has been used in countless applications since the earliest days of digital engineering. The simplest adding circuit performs addition in much the same way that humans do, performing the operation "right to left", bit-by-bit from the LSB to the MSB. Like any circuit that deals with signals grouped as binary numbers, the simplest adding circuit can most easily be designed using the bit-slice approach. By focusing on the requirements for adding a single pair of bits from two N-bit binary numbers, an otherwise complex design challenge can be divided into a more tractable problem. Once a circuit that can add any pair of bits has been designed, that circuit can be replicated N times to create an N-bit adder.

The logic graph below shows the eight different cases that may be encountered when adding two binary numbers. The highlighted bit pairs and the associated carries show that a bit-slice adder circuit must process three inputs (the two addend bits and a carry-in from the previous stage) and produce two outputs (the sum bit and a carry out bit). In the exercises and lab project, you are asked to create a truth table and circuit for various adding circuits.



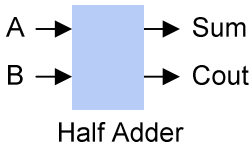
A block diagram for the bit-slice circuit is shown on the right, and it is called a Full Adder (FA). Full adders can be used to assemble circuits that can add any number of bits. The figure below shows an 8-bit adder circuit constructed from eight individual full-adder bit-slice circuits. Note that the input and output pin locations on the bit-slice block diagram have been re-arranged in the diagram to make the drawing more convenient.





The carry-out generated in the very first stage of an 8-bit adder must "ripple" through all seven higher-order stages before a valid 9-bit sum can be produced. It is this need to ripple carry information from one bit-slice to the next that gives the adder its name – the Ripple Carry Adder (RCA). This slice-by-slice processing of carry information severely limits the speed at which an RCA can run. Consider for example an 8-bit RCA adding A = “11111010” and B = “10001110”, and then consider that the least-significant-bit (LSB) of the B operand switches from a ‘0’ to a ‘1’. In order for all nine bits of the adder to show the correct answer, carry information from the LSB must ripple through all eight full adders. If each full adder requires, say, 1 nanosecond (ns) to create the sum and carry-out bits after an input changes, then an 8-bit RCA will require up to 8ns to create an accurate answer (8ns is the worst-case situation, which occurs when an operand LSB change requires the S8 output bit to change). If an 8-bit addition in a computer circuit requires 8ns, then the computer’s maximum operating frequency would be the reciprocal of 8ns, or 125MHz. Most computers today are 32 bits – a 32 bit addition using an RCA would require 32ns, limiting the computers operating frequency to no more than about 33MHz. An RCA circuit is too slow for many applications – a faster adder circuit is needed.

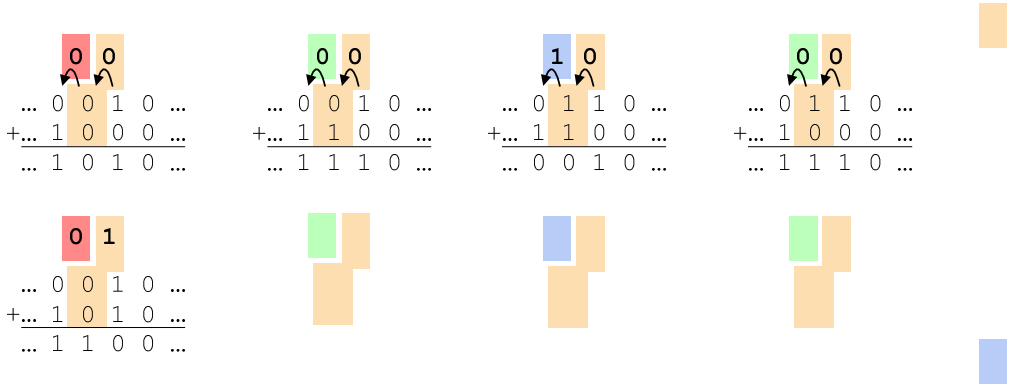
Note the carry-in of the RCA LSB is connected directly to ground, because (by definition) the carry-in to the LSB of any adder must be logic 0. It is possible to capitalize on this observation, and create a smaller bit-slice circuit for use in the LSB position that does not have a carry-in input. Called a Half-Adder (HA), this reduced-function adder circuit is often used in the LSB position.



The carry-look-ahead adder (CLA) overcomes the speed limitations of the RCA by using a different circuit to determine carry information. The CLA uses a simpler bit-slice module, and all carry-forming logic is placed in a separate circuit called the “Carry Propagate/Generate (CPG) circuit. The CPG circuit receives carry-forming outputs from all bit-slices in parallel (i.e., at the same time), and forms all carry-in signals to all bit-slices at the same time. Since all carry signals for all bit positions are determined at the same time, addition results are generated much faster.

Since a CLA also deals with signals grouped as binary numbers, the bit slice approach is again indicated. Our goal is to re-examine binary number addition to identify how and where carry information is generated and propagated, and then to exploit that new knowledge in an improved circuit.

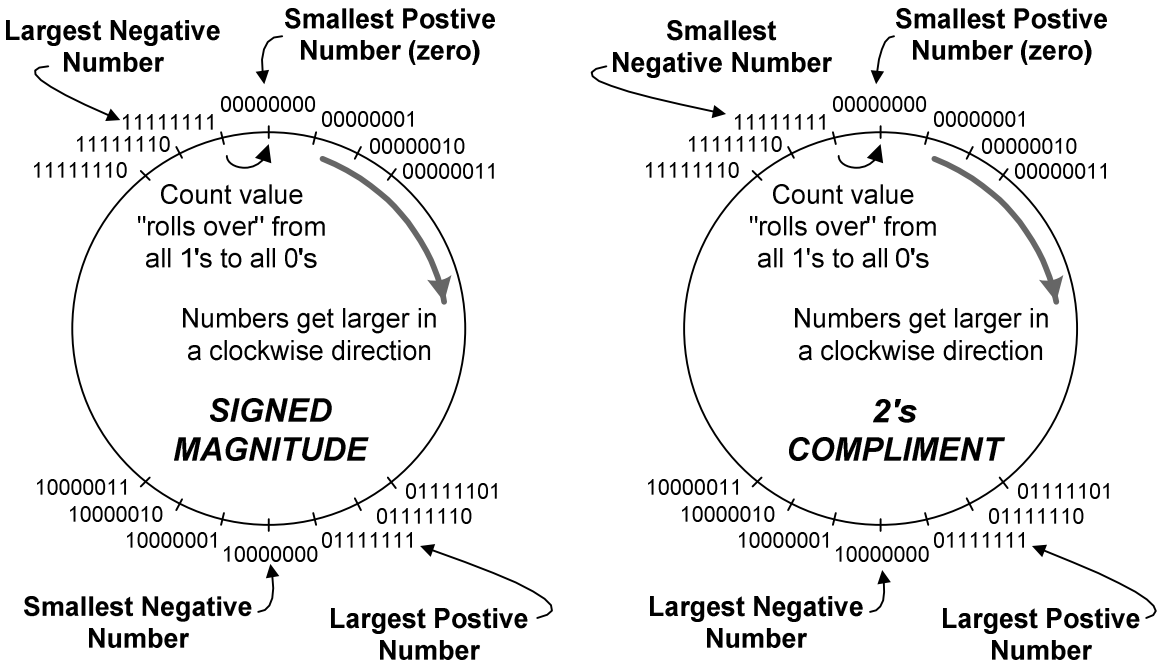
The figure below shows the same eight addition cases as were presented in the first figure. Note that in just two of the cases (3 and 7), a carry out is generated. Also note that in four cases, a carry that was previously generated will propagate through the current pair of bits, asserting a carry out even though the current bits by themselves would not have created a carry.



Digital systems have a fixed number of signals that can be used to represent binary numbers. Smaller, simpler systems might use 8-bit buses that can only represent 256 different binary numbers, while larger systems might use 16, 32, or even 64 bit busses. Whatever the number of bits, all systems have a finite number of wires, storage elements, and processing elements to represent and manipulate digital data. The number of available bits determines how many different numbers can be represented in a given system.

Digital circuits that perform arithmetic functions often must deal with negative numbers, so a method of representing negative numbers must be defined. An N-bit system can represent 2^N total numbers, so an useful encoding would use half the available codes (i.e., $2^N/2$) to represent positive numbers, and half negative numbers. A single bit can be designated as a "sign bit" to distinguish positive and negative numbers – if the sign bit is '1', the number is negative; if the sign bit is '0', positive. The most-significant-bit (MSB) is a good choice for the sign bit, because if it is a '0' (indicating a positive number), then it can simply be ignored when figuring the number's magnitude.

Of all possible encodings of negative numbers, two have been used most often: signed magnitude, and 2's compliment. Signed magnitude representations simply designate the MSB as the sign bit, and the remaining bits as magnitude. In an 8-bit signed-magnitude system, "16" would be represented as "00010000", and "-16" as "10010000". This system is easy for humans to interpret, but it has a major disadvantage for digital circuits: if the 0 to 2^N count range is traversed from smallest to largest, then the largest positive number appears halfway through the range, followed immediately by the smallest negative number. Further, the largest negative number appears at the end of the range (at binary number 2^N), and counting once more results in "rollover", since the number 2^N+1 cannot be represented. Thus, 0 follows 2^N in the count range, so the largest negative number is immediately adjacent to the smallest positive number. Because of this, a simple operation like "2 – 3", which requires counting backwards from two three times, will not yield the expected result of "-1", but rather the largest negative number in the system. A better system would place the smallest positive and negative numbers immediately adjacent to one another in the count range, and this is precisely what the 2's compliment representation does. The number wheels below illustrate signed-magnitude and 2's compliment encodings for 8-bit numbers.



Number wheels illustrating signed magnitude and 2's complement encodings for negative numbers

In 2's compliment encoding, the MSB still functions as a sign bit – it is always '1' for a negative number, and '0' for a positive number. The 2's compliment code has a single "0" value, defined by a bit pattern containing all 0's (including the leading '0'). This leaves $2^N - 1$ codes to represent all non-zero numbers, both positive and negative. Since 2^N -1 is an odd number, we end up with $(2^N - 1)/2$ negative numbers, and $(2^N - 1)/2$ -1 positive numbers (since "0" uses one of the available codes for a positive number). In other words, we can represent one more non-zero negative number than positive, and the magnitude of the largest negative number is one greater than the magnitude of the largest positive number.

The disadvantage to 2's compliment encoding is that negative numbers are not easily interpreted by humans (e.g., is it clear that "11110100" represents a -12?). A simple algorithm exists for converting a positive number to a 2's compliment-encoded negative number of the same magnitude, and for converting a 2's compliment-encoded negative number to a positive number of the same magnitude. The algorithm, illustrated in examples below, requires inverting all bits of the number to be converted, and then adding '1' to the resulting bit pattern. The algorithm can be visualized in the 2's compliment number wheel above by noting that "inverting all bits" reflects a number around an axis drawn through '0' and the largest negative number, and "adding one" compensates for the 2's compliment code containing one more negative code than positive code.

0 0 0 1 0 0 0 1	= 17 → *Convert to*	
1 1 1 0 1 1 1 0	1) Invert all bits	
+ 0 0 0 0 0 0 0 1	2) Add one	
1 1 1 0 1 1 1 1	= -17	

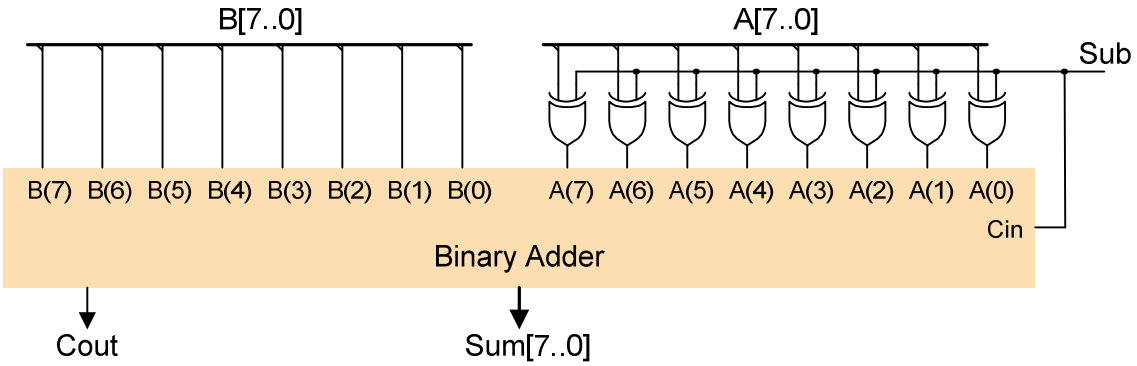
1 1 0 1 1 1 0 1	= -35 → *Convert to*	*5*
0 0 1 0 0 0 1 0	1) Invert all bits	
+ 0 0 0 0 0 0 0 1	2) Add one	
0 0 1 0 0 0 1 1	= 35	

1 0 0 0 0 0 0 1	= -127 → *Convert to*	
0 1 1 1 1 1 1 0	1) Invert all bits	
+ 0 0 0 0 0 0 0 1	2) Add one	
0 1 1 1 1 1 1 1	= 127	

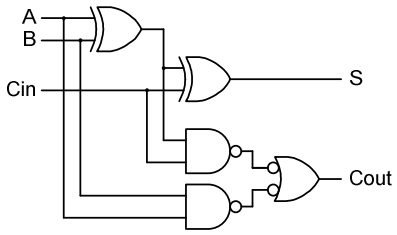
0 0 0 0 0 0 0 1	= 1 → *Convert to*	
1 1 1 1 1 1 1 0	1) Invert all bits	
+ 0 0 0 0 0 0 0 1	2) Add one	
1 1 1 1 1 1 1 1	= -1	

Adder/Subtractors

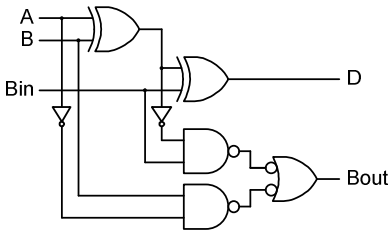
An adder circuit can easily be modified with a combinational logic circuit that can selectively implement the 2's compliment encoding of one of the input binary numbers. Recall that a two-input XOR gate can be used as a "controlled inverter", where one of the inputs is passed through to the output either inverted or unchanged, based on the logic level of the second "control" input. If XOR gates are included on all bits of one of the operand inputs to



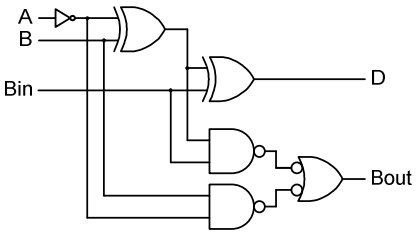
When designed from truth-tables and K-maps, a full subtractor is very similar to a full adder, but it contains two inverters that a full adder does not. When configured to subtract, an adder/subtractor circuit adds a single inverter (in the form of an XOR gate) to one input of a full adder module. A ripple borrow subtractor performs the same function as an adder/subtractor in subtract mode, but the two circuits are different as shown below. The differences can be explained by noting the carry-in to the LSB of the adder/subtractor must be set to a '1' to form the 2's complement coding of the operand, but it takes some thought to convince yourself. In the exercises, you are asked to demonstrate how the circuit structures of a ripple-carry adder circuit configured as a 2's compliment subtractor and a ripple-borrow subtractor perform identical functions.



Full Adder



Full Subtractor



Full Adder
configured for subtract

Adder overflow

When performing arithmetic operations on numbers that must use a fixed number of bits, it is possible to create a result that requires more bits than are available. For example, if the two 8-bit numbers 240 and 25 are added, the result 265 cannot be represented as an 8-bit binary number. When numbers are combined and the result requires more bits than are available, overflow (for positive results) or underflow (for negative results) errors occur. Although underflow and overflow errors cannot be prevented, they can be detected.

The behavioral requirements for an overflow/underflow detect circuit can be defined by examining several examples of addition overflow and subtraction underflow. In the simplest case, the carry-in to the MSB can be compared to the carry out of the same bit. But it is also possible to detect an overflow/underflow condition without needing access to the carry-in of the MSB. In the exercise and lab project, you are asked to design circuits that can output a '1' whenever an addition or subtraction result is incorrect due to underflow or overflow.

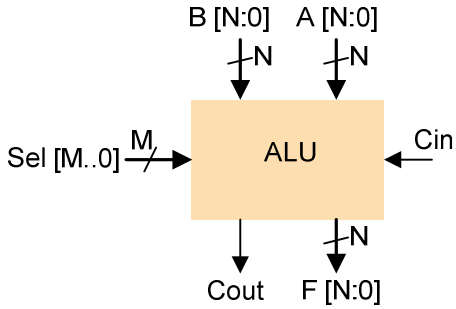
Hardware multipliers

In the circuit above, each bit in the multiplier is AND'ed with each bit in the multiplicand to form the corresponding partial product bits. The partial product bits are fed to an array of full adders (and half adders where appropriate), with the adders shifted to the left as indicated by the multiplication example. The final partial products are added with a CLA circuit. Note that some full-adder circuits bring signal values into the carry-in inputs (instead of carry's from the neighboring stage). This is a valid use of the full-adder circuit; the full adder simply adds any three bits applied to its inputs. You are encouraged to work through a few examples on your own to confirm the adder array and CLA work together to properly sum the partial products. In the lab project, you are asked to implement a multiplier circuit.

As the number of multiplier and multiplicand bits increase, so does the number of adder stages required in the multiplier circuit. It is possible to develop a faster adding array for use in a multiplier by follow a similar line of reasoning as was used in the development of the CLA circuit.

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.



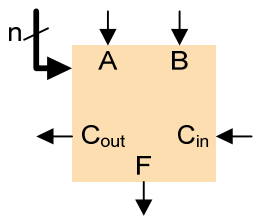
ALUs can be designed to perform a variety of different arithmetic and logic functions. Possible arithmetic functions include addition, subtraction, multiplication, comparison, increment, decrement, shift, and rotate; possible logic functions include AND, OR, XOR, XNOR, INV, CLR (for clear), and PASS (for passing a value unchanged). All of these functions find use in computing systems, although a complete description of their use is beyond the scope of this document. An ALU could be designed to include all of these functions, or a subset could be chosen to meet the specific needs of a given application. Either way, the design process is similar (but simpler for an ALU with fewer functions).

As an example, we will consider the design of an ALU that can perform one of eight functions on 8-bit data values. This design, although relatively 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 (three arithmetic, four logic, and a clear or '0').

Targeted ALU operations are shown in the operation table. The three control bits used to select the ALU operation are called the "operation code" (or Op Code), because if this ALU were used in an actual microprocessor, these 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).

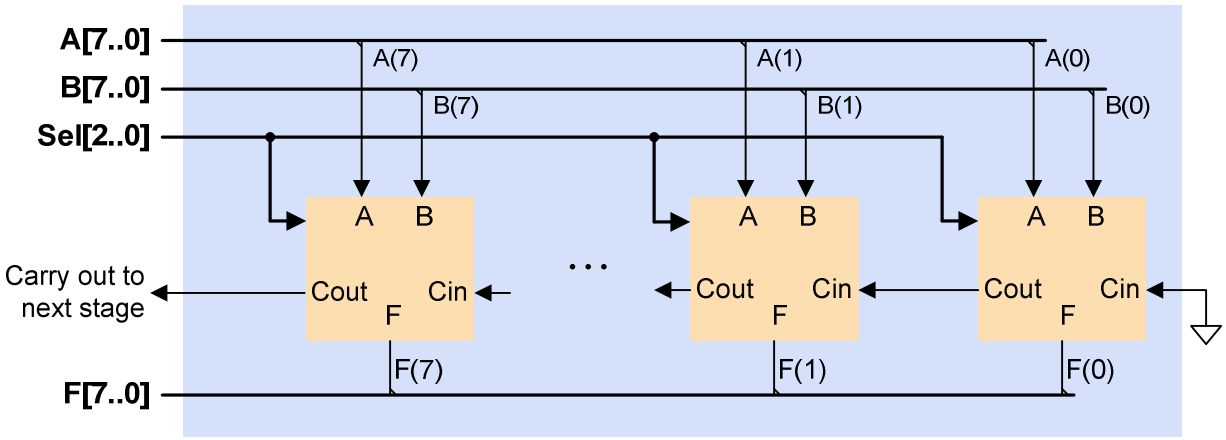
Op Code	Function
000	A PLUS B
001	A PLUS 1
010	A MINUS B
011	0
100	A XOR B
101	A'
110	A OR B
111	A AND B

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.



Op Code	Function	F	Cout
000	A PLUS B	A **xor** B **xor** Cin	(A **and** B) **or** (Cin **and** (A **xor** B))
001	A PLUS 1	A **xor** Cin	A **and** Cin
010	A MINUS B	A **xor** B **xor** Cin	(A' **and** B) **or** (Cin **and** (A **xor** B)')
011	Zero	0	0
100	A XOR B	A **xor** B	0
101	A'	A'	0
110	A OR B	A **or** B	0
111	A AND B	A **and** B	0

A block diagram and operation table for our ALU example is shown above. Note in the operation table, entered variables are used to define the functional requirements for the two outputs (F and Cout) of the bit-slice module. If entered variables were not used, the table would have required 64 rows. Since Cout is not required for logic functions, it can be assigned '0' or a don't care for those rows. A circuit diagram for an 8-bit ALU based on the developed bit slice is shown below.



Once the ALU operation table is complete, a circuit can be designed following any one of several methods: K-maps can be constructed and minimal circuits can be looped; muxes can be used (with an 8:1 mux for F and a 4:1 mux for Cout); the information could be entered into a computer-based minimizer and the resulting equations implemented directly; or we could bypass all the difficult and error-prone structural work and create a VHDL description.

Behavioral VHDL ALU description

As mentioned in the previous module, when circuit outputs are assigned a value based on some number of select inputs, a *selected signal assignment* statement can be used to minimize and clarify VHDL code. A selected signal assignment can assign one of 2^N possible outputs based on the state of N select bits. Simple multiplexor circuits can easily be coded using a selected assignment, and so can more complex multiplexors that assign the result of an arithmetic or logic function to the output.

The example code to the right shows an example of an 8-bit, four function ALU based on a selected signal assignment statement. By following this example, you can easily define any ALU or similar circuit.

This example code can easily be modified to create a more complex ALU. For example, more sel bits (and therefore more ALU functions) can be added, and/or different ALU functions can be coded.

```
entity 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 behavioral of 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 behavioral;
```

An 8-bit, four-function ALU example

More about VHDL

Structural vs. Behavioral Designs

The VHDL language can be used to define circuits in many different ways, with different levels of abstraction. Behavioral descriptions describe only the input and output relationships of a circuit, with no attention given to the ultimate structure of the circuit. Behavioral designs are abstract descriptions that are relatively easy to read and understand, and they rely on synthesizer software to define structural details. At the other end of the spectrum, structural descriptions define circuit blocks and interconnections. They are rigorous descriptions and full of detail, and they can be difficult for other engineers to read and understand. Often, the circuit's behavior gets lost in the volume of structural detail. Both methods have advantages and disadvantages. Behavioral designs can proceed quickly, since it is usually much easier to describe how something behaves rather than how it is built. Because behavioral designs can be completed relatively quickly, designers can spend more time studying various alternative design approaches, and more time ensuring all design requirements are properly implemented. But in trade, behavioral designs hide many important details, making it difficult to model and simulate circuits with a high degree of fidelity.

In our brief exposure to the VHDL language so far, we have focused on basic behavioral descriptions of circuits. For example, in an earlier lab, you were asked to create a VHDL description for a circuit that could detect the first seven prime numbers. The circuit was implemented using a signal assignment statement, and no concern was given to the actual structure of the circuit – those details were left to the synthesizer. It would have been a much more time consuming project if you were required to first find the circuit structure, and then implement it.

In some situations, designers choose to use structural VHDL to define a circuit instead of (or in addition to) defining its behavior. More work is required to create a structural VHDL definition, because a larger amount of detail must be described. But in return for that greater effort, far more accurate and powerful simulation models can be created. For example, consider a circuit to add two 4-bit numbers. A behavioral description would read "Y <= A + B;" (assuming Y, A, and B are all four-

bit standard logic vectors). After synthesis, this circuit might be composed of four full adders connected in an RCA configuration. But the simulator would not have access to the internal nodes (like the carry signals), and so timing or other problems with those signals could not easily be discovered. Contrast this behavioral statement with the structural description in the example code below. In the structural description, internal signals are explicitly named and are therefore readily available to the simulator.

In a given design, an engineer may elect to first capture a design relatively quickly using high-level behavioral VHDL to study its behavior and validate its performance against the design requirements. At this stage, different solutions can easily be coded and simulated, and their relative advantages brought to light. Once a preferred solution is discovered, some or all of the design might be recoded in structural VHDL so that more detailed simulations could occur, giving the designer a more complete understanding of hardware behavior. As a further advantage of using structural methods, completed design blocks can be reused in subsequent designs.

Structural VHDL methods are similar to schematic-based methods – individual components are designed and saved into a project library, and then added to the higher-level design as needed. Component signals can be connected directly to I/O ports in the higher-level design, or interconnected to other components using locally declared signals. Any valid VHDL entity/architecture pair can be used as a component in another VHDL source file, just like any circuit schematic can be made into a macro and then used in another schematic. In a schematic environment, a component can be added to the design by adding its graphic symbol to the schematic page. In VHDL, a component is added to a source file by first declaring it in the “declaration area”, and then by instantiating it. A component *declaration*

s e dtnr te it5 o a sourtafiit 5 c io thccr 5 a5 5 l 5a5 5 o n ao5 5 d s 5 t 5 5

```
entity  ; Cs is
  port 1 sH X     ] in std_logic_vector (3 downto BRF
         “        ]  ut std_logic_vector (3 downto BRF
        C  ut     ]  ut std_logicRF
end  ; CsF

architecture  ItGubtuGa” o   ; Cs is

  coOponent = s
    port 1sH X      ]  in std_logicF
         “H C  ut ] out std_logicRF
  end coOponentF

  coOponent > s
    port 1sH X      ]  in std_logicF
         “H C  ut ] out std_logicRF
  end coOponentF

  signal  : 0 ) std_logic_vector (3 downto ;B

 egin

  CB] = s port Oap 1s02s1BRH X02X1BRH “02“1BRH C  ut02C31BRRF
  C,] >s port Oap 1s02s1,RH X02X1,RH Cin02C31BRH “02“1,RH C  ut02C31,RRF
  Cv] >s port Oap 1s02s1vRH X02X1vRH Cin02C31,RH “02“1vRH C  ut02C31vRRF
  C9] >s port Oap 1s02s19RH X02X19RH Cin02C31vRH “02“19RH C  ut02C319RRF

end Seya:i Ga”F
```

C r8 nentI and IignaI aGe deb”aGed in tye *Declaration Area* Set7een tye sGbyitebtuGe and Xegin Itate r entI

C r8 nentI ban Se inItantiated an[7yeGe 7ityin tye sGbyitebtuGe Itate r ent

Components are connected to signals in the higher-level design using a *port map* statement. Component signals may be connected directly to I/O ports in the higher-level design, or they may connect to other components using locally declared signals. Every component instantiation statement must include a port map statement to establish all required signal connections. In the example code above, some component signals are connected to I/O port signals in the higher-level design, and some are connected to other components using the locally declared CO signals.

The component instantiation statement starts with a unique alphanumeric label terminated with a colon. Labels can use any legal characters (generally, letters and numbers), and they can be descriptive of the component, or just a sequential place holder as in the example above. The component entity name follows the label, and then the key words “port map”. The port map statement contains a list of all component signals named exactly as they were in the earlier declaration statement. Component signals are listed one by one, followed by the “assigned to” operator => and the name of the higher-level signal to which they are assigned. All port map signal assignments are contained in a list demarked by parenthesis, and the close parenthesis is followed by a semicolon.

The 4-bit RCA example above shows two component declaration statements (one for the half-adder named HA and one for the full-adder named FA) and four component instantiations. The component declaration statements will generate errors if the half-adder and full-adder entities are not named "HA" and "FA" (i.e., the HA entity statement must read "entity *HA* is"). Often, new VHDL programmers mistakenly use the windows *filename* for a component rather than the entity name. You can choose to give the source file the same name as the entity if you wish.

In a typical structural VHDL design, some components must be connected to other components using locally declared signals. The 4-bit RCA example is no exception – local signals are needed to connect the carry-out of one bit slice to the carry-in of the next. In the example, four new signals (in the form of a bus named CO) are declared. Since these signals are not included in the higher-level entity port statement, they are not "visible" outside of the entity and can only be used locally, within the entity. If such signals must be accessed outside of the entity, then their declaration must be removed from the declaration area and placed in the higher-level entity's port statement.

In summary, a structural VHDL source file uses other, pre-designed VHDL modules as components. Any pre-existing VHDL entity/architecture pair can be used as a component by first declaring the entity as a component and then by instantiating the component. An instantiation is comprised of a unique alphanumeric label, the entity name, and a port map statement that connects component signals to higher-level signals (either locally declared signals or directly to I/O port signals).

Modular Design in VHDL

The VHDL language includes several ways to reuse previously written code in other source files. In the method discussed above, you can write circuit descriptions as entity/architecture pairs and then include that code as a component in another design. In another method, you can write circuit descriptions in subprograms like functions and procedures. Subprograms encapsulate often-used descriptions in a single piece of code that can be parameterized for use in different contexts within a source file.

The creation of subprograms is beyond our current scope, but you have already used several subprograms (in the form of functions) without knowing. The VHDL language does not contain any intrinsic facilities to evaluate logic expressions in assignment statements, so logic functions like AND, OR, NAND, etc. are defined as functions, and those functions are included in a library that is distributed with every VHDL tool set.

A VHDL library is a collection of "design units" that have be prewritten and analyzed, and stored in a directory in the host computer's file system. Any design unit stored in a library can be used in any other source file. The VHDL language defines five types of design units, including entities, architectures, packages, package bodies, and configurations. You are familiar with entity and architecture design units. Packages are used for defining and storing commonly used declarations for components, types, constants, global signals, etc., and package bodies contain functions and procedures. Configurations associate an entity with a particular architecture, and are only needed in the somewhat rare case that more than one architecture is written for a given entity (configurations are occasionally used in larger or more complex designs and will be dealt with in a later module).

The std_logic type you have been using is defined in the "std_logic_1164" package that was written long ago, stored in a library named "IEEE", and transferred to your computer when you installed the ISE/WebPack tool. Logic functions like AND, OR, NAND, NOR, XOR, XNOR, etc., are stored in the 1164 package body. If the 1164 package were not available in the IEEE library, you couldn't use the std_logic types, and you couldn't write assignment statements like "Y <= A and B;".
In fact, several types and functions have been standardized by the IEEE, and they are included in packages within the IEEE library. As mentioned, the std_logic_1164 package in the IEEE library contains definitions for common data types (like std_logic and std_logic_vector) as well as common logic functions like AND, OR, NAND, NOR, XOR, etc. Another common package called "std_logic_arith" contains a collection of arithmetic functions like addition (+), subtraction (-), multiplication (*). Still other packages contain further collections of useful functions.

Libraries and packages must be declared in a source file before their contents can be accessed. Libraries are identified in a source file using a "logical name"; a library manager tool associates a logical library name with the library's physical location in the computer's file system. This way, VHDL source files need only know the logical names. The keywords *li rary* (followed by the library's logical name) and *use* (followed by the package name) must be included in a source file to make their contents available. When the VHDL analyzer encounters a word or symbol that is doesn't recognize, it will look inside the available libraries and packages for suitable definitions. For example, when the analyzer finds the "and" in "Y<= A and B;", or the "+" in "Y <= A + B;", it will look for "and" and "+" definitions in the packages that have been declared. It is common practice to include the library and use statements shown to the right in every VHDL source file so that common types and functions can be used.

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

Arithmetic Functions in VHDL

The std_logic_arith package in the IEEE library defines several arithmetic functions that can be performed on std_logic and std_logic_vector data types. If the "library IEEE" and "use std_logic_arith" statements are included in your source file, addition ('+'), subtraction ('-'), multiplication ('*'), and division ('/') operators (as well as some other operators) can be used with std_logic types. For example, the binary numbers carried by two std_logic_vectors can be summed by writing a statement like "Y<=A + B;".

When using arithmetic operators on std_logic_vectors, the output vector must be sized correctly or the VHDL analyzer will flag an error, or data will be lost. For our purposes in the accompanying lab module, ensure the output logic vector used to capture arithmetic function outputs is no smaller than the input vectors. In general, if smaller logic vectors are combined through arithmetic operations into larger vectors, the results will be right-justified in the larger output vector. If larger vectors are combined into vectors that are too small to contain all required output bits, the results will still be right justified and the more significant bits will be lost.