7.5 Constructing a basic arithmetic logic unit

(Original section[1])

> “ ALU n. [Arthritic Logic Unit or (rare) Arithmetic Logic Unit] A random-number generator supplied as standard with all computer systems.
> *Stan Kelly-Bootle, The Devil's DP Dictionary, 1981.*

The *arithmetic logic unit (ALU)* is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. This section constructs an ALU from four hardware building blocks (AND and OR gates, inverters, and multiplexors) and illustrates how combinational logic works. In the next section, we will see how addition can be sped up through more clever designs.

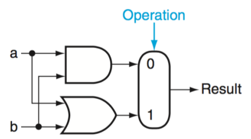
Because the LEGv8 word is 64 bits wide, we need a 64-bit-wide ALU. Let's assume that we will connect 64 1-bit ALUs to create the desired ALU. We'll therefore start by constructing a 1-bit ALU.

A 1-bit ALU

The logical operations are easiest, because they map directly onto the hardware components in COD Figure A.2.1 (Standard drawing for an AND gate ...) .

The 1-bit logical unit for AND and OR looks like the figure below. The multiplexor on the right then selects *a* AND *b* or *a* OR *b*, depending on whether the value of *Operation* is 0 or 1. The line that controls the multiplexor is shown in color to distinguish it from the lines containing data. Notice that we have renamed the control and output lines of the multiplexor to give them names that reflect the function of the ALU.

Figure 7.5.1: The 1-bit logical unit for AND and OR (COD Figure A.5.1).



The next function to include is addition. An adder must have two inputs for the operands and a single-bit output for the sum. There must be a second output to pass on the carry, called *CarryOut*. Since the CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called *CarryIn*. The figure below shows the inputs and the outputs of a 1-bit adder. Since we know what addition is supposed to do, we can specify the outputs of this "black box" based on its inputs, as COD Figure A.5.3 (Input and output specification for a 1-bit adder) demonstrates.

Figure 7.5.2: A 1-bit adder (COD Figure A.5.2).

This adder is called a full adder; it is also called a (3, 2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2, 2) adder or half-adder.

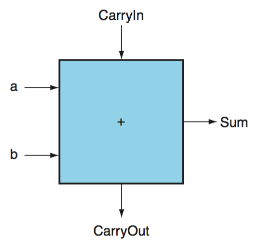
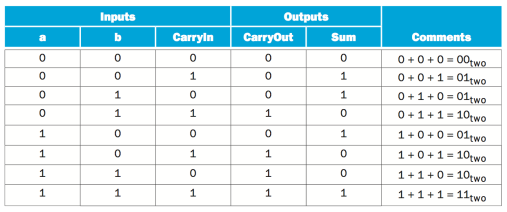


Figure 7.5.3: Input and output specification for a 1-bit adder (COD Figure A.5.3).



Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

We can express the output functions CarryOut and Sum as logical equations, and these equations can in turn be implemented with logic gates. Let's do CarryOut. The figure below shows the values of the inputs when CarryOut is 1.

Figure 7.5.4: Values of the inputs when CarryOut is a 1 (COD Figure A.5.4).

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

We can turn this truth table into a logical equation:

$$\text{CarryOut} = (\text{b} \cdot \text{CarryIn}) + (\text{a} \cdot \text{CarryIn}) + (\text{a} \cdot \text{b}) + (\text{a} \cdot \text{b} \cdot \text{CarryIn})$$

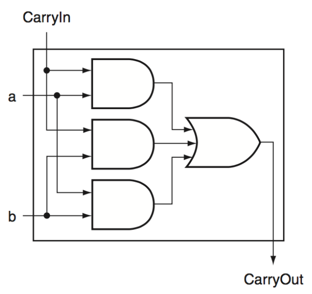
If $a \cdot b \cdot CarryIn$ is true, then all of the other three terms must also be true, so we can leave out this last term corresponding to the fourth line of the table. We can thus simplify the equation to

$$\text{CarryOut} = (\text{b} \cdot \text{CarryIn}) + (\text{a} \cdot \text{CarryIn}) + (\text{a} \cdot \text{b})$$

The figure below shows that the hardware within the adder black box for CarryOut consists of three AND gates and one OR gate. The three AND gates correspond exactly to the three parenthesized terms of the formula above for CarryOut, and the OR gate sums the three terms.

Figure 7.5.5: Adder hardware for the CarryOut signal (COD Figure A.5.5).

The rest of the adder hardware is the logic for the Sum output given in the equation below.



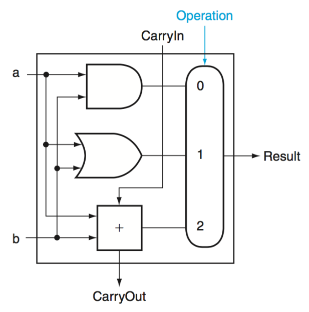
The Sum bit is set when exactly one input is 1 or when all three inputs are 1. The Sum results in a complex Boolean equation (recall that $\overline{a}$ means NOT a):

$$\text{Sum} = (\text{a} \cdot \overline{\text{b}} \cdot \overline{\text{CarryIn}}) + (\overline{\text{a}} \cdot \text{b} \cdot \overline{\text{CarryIn}}) + (\overline{\text{a}} \cdot \overline{\text{b}} \cdot \text{CarryIn}) + (\text{a} \cdot \text{b} \cdot \text{CarryIn})$$

The drawing of the logic for the Sum bit in the adder black box is left as an exercise for the reader.

The figure below shows a 1-bit ALU derived by combining the adder with the earlier components. Sometimes designers also want the ALU to perform a few more simple operations, such as generating 0. The easiest way to add an operation is to expand the multiplexor controlled by the Operation line and, for this example, to connect 0 directly to the new input of that expanded multiplexor.

Figure 7.5.6: A 1-bit ALU that performs AND, OR, and addition (see above figure) (COD Figure A.5.6).

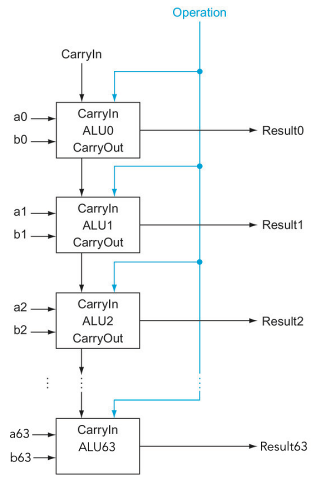


A 64-bit ALU

Now that we have completed the 1-bit ALU, the full 64-bit ALU is created by connecting adjacent "black boxes." Using *xi* to mean the *i*th bit of *x*, the figure below shows a 64-bit ALU. Just as a single stone can cause ripples to radiate to the shores of a quiet lake, a single carry out of the least signif cant bit (Result0) can ripple all the way through the adder, causing a carry out of the most significant bit (Result31). Hence, the adder created by directly linking the carries of 1-bit adders is called a *ripple carry* adder. We'll see a faster way to connect the 1-bit adders in COD Section A.6 (Faster addition: Carry lookahead).

Figure 7.5.7: A 64-bit ALU constructed from 64 1-bit ALUs (COD Figure A.5.7).

CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.



Subtraction is the same as adding the negative version of an operand, and this is how adders perform subtraction. Recall that the shortcut for negating a two's complement number is to invert each bit (sometimes called the *one's complement*) and then add 1. To invert each bit, we simply add a 2:1 multiplexor that chooses between b and $\bar{b}$, as the figure below shows.

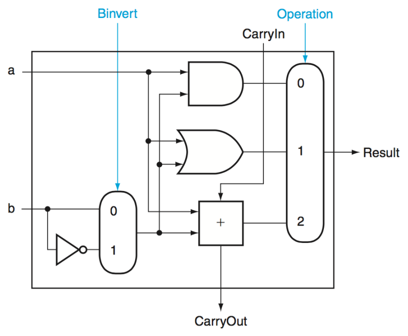
Suppose we connect 64 of these 1-bit ALUs, as we did in the figure above. The added multiplexor gives the option of b or its inverted value, depending on Binvert, but this is only one step in negating a two's complement number. Notice that the least significant bit still has a CarryIn signal, even though it's unnecessary for addition. What happens if we set this CarryIn to 1 instead of 0? The adder will then calculate a + b + 1. By selecting the inverted version of b, we get exactly what we want:

$$\mathrm{a} + \bar{\mathrm{b}} + 1 = \mathrm{a} + (\bar{\mathrm{b}} + 1) = \mathrm{a} + (-\mathrm{b}) = \mathrm{a} - \mathrm{b}$$

The simplicity of the hardware design of a two's complement adder helps explain why two's complement representation has become the universal standard for integer computer arithmetic.

Figure 7.5.8: A 1-bit ALU that performs AND, OR, and addition on a and b or a and not b (COD Figure A.5.8).

By selecting $\bar{b}$ (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.



A LEGv8 ALU also needs a NOR function. Instead of adding a separate gate for NOR, we can reuse much of the hardware already in the ALU, like we did for subtract. The insight comes from the following truth about NOR:

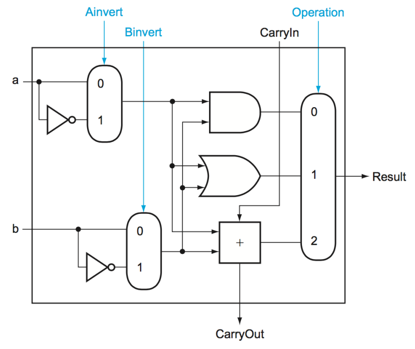
$$(\overline{\mathrm{a} + \mathrm{b}}) = \bar{\mathrm{a}} \cdot \bar{\mathrm{b}}$$

That is, NOT (a OR b) is equivalent to NOT a AND NOT b. This fact is called DeMorgan's theorem and is explored in the exercises in more depth.

Since we have AND and NOT b, we only need to add NOT a to the ALU. The figure below shows that change.

Figure 7.5.9: A 1-bit ALU that performs AND, OR, and addition on a and b or not a and not b (COD Figure A.5.9).

By selecting $\bar{a}$ (Ainvert = 1) and $\bar{b}$ (Binvert = 1), we get a NOR b instead of a AND b.



Tailoring the 64-bit ALU to LEGv8

These four operations—add, subtract, AND, OR—are found in the ALU of almost every computer, and the operations of most LEGv8 instructions can be performed by this ALU. But the design of the ALU is incomplete.

One instruction that still needs support is the compare and branch zero (`CBZ`). Recall that the operation the ALU just passes the register input value. For the ALU to perform `CBZ`, we first need to expand the three-input multiplexor in COD Figure A.5.8 (A 1-bit ALU that performs ...) to add an input for the `CBZ` result. We call that new input Pass and use it only for `CBZ`.

The top drawing of the figure below shows the new 1-bit ALU with the expanded multiplexor. Figure A.5.11 shows the 64-bit ALU.

Figure 7.5.10: (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or not b, and (bottom) a 1-bit ALU for the most significant bit (COD Figure A.5.10).

The top drawing includes a direct input that is connected to perform the set on less than operation (see the figure below); the bottom has a direct output from the adder for the less than comparison called Set. (See COD Exercise A.24 at the end of this appendix to see how to calculate overflow with fewer inputs.)

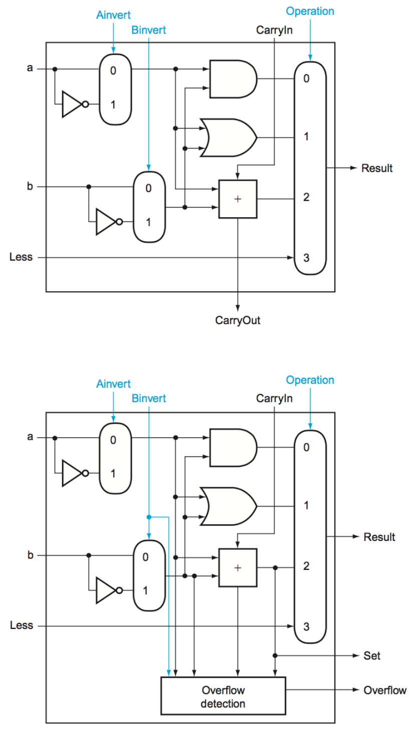
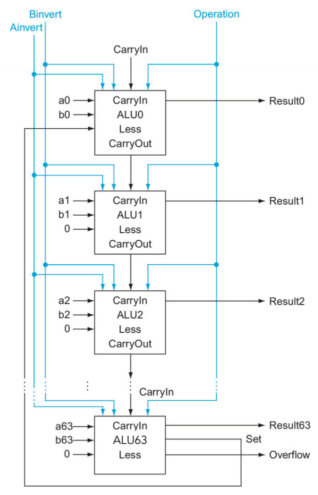


Figure 7.5.11: A 64-bit ALU constructed from the 63 copies of the 1-bit ALU in the top of the figure above and one 1-bit ALU in the bottom of that figure (COD Figure A.5.11).

The Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit. If the ALU performs a - b and we select the input 3 in the multiplexor in the figure above, then Result = 0 … 001 if a < b, and Result = 0 … 000 otherwise.



To further tailor the ALU to the LEGv8 instruction set, we must support conditional branch instructions. These instructions branch either if two registers are equal or if they are unequal. The easiest way to test equality with the ALU is to subtract b from a and then test to see if the result is 0, since

$$(\mathrm{a}-\mathrm{b}=0) => \mathrm{a}=\mathrm{b}$$

Thus, if we add hardware to test if the result is 0, we can test for equality. The simplest way is to OR all the outputs together and then send that signal through an inverter:

$$\mathbf{Zero} = \overline{(\mathbf{Result63}+\mathbf{Result62}+\ \ldots\ +\mathbf{Result2}+\mathbf{Result1}+\mathbf{Result0})}$$

The figure below shows the revised 64-bit ALU. We can think of the combination of the 1-bit Ainvert line, the 1-bit Binvert line, and the 2-bit Operation lines as 4-bit control lines for the ALU, telling it to perform add, subtract, AND, OR, or set on less than. COD Figure A.5.13 (The values of the three ALU control lines ...) shows the ALU control lines and the corresponding ALU operation.

Figure 7.5.12: The final 64-bit ALU (COD Figure A.5.12).

This adds a Zero detector to the figure above.

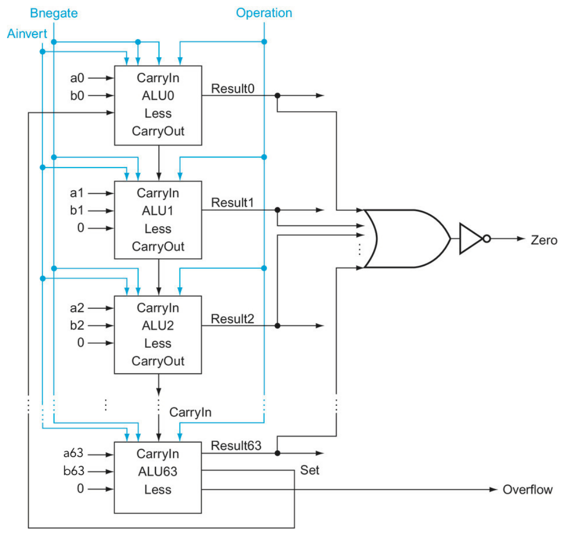


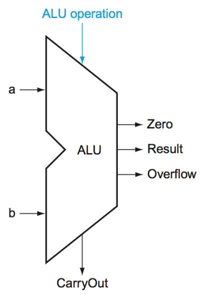
Figure 7.5.13: The values of the three ALU control lines, Bnegate, and Operation, and the corresponding ALU operations (COD Figure A.5.13).

ALU control lines	Function

Finally, now that we have seen what is inside a 64-bit ALU, we will use the universal symbol for a complete ALU, as shown in the figure below.

Figure 7.5.14: The symbol commonly used to represent an ALU (COD Figure A.5.14).

This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.



Defining the LEGv8 ALU in Verilog

The figure below shows how a combinational LEGv8 ALU might be specified in Verilog; such a specification would probably be compiled using a standard parts library that provided an adder, which could be instantiated. For completeness, we show the ALU control for LEGv8 in COD Figure A.5.16 (The LEGv8 ALU control ...), which is used in COD Chapter 4 (The Processor), where we build a Verilog version of the LEGv8 datapath.

Figure 7.5.15: A Verilog behavioral definition of a LEGv8 ALU (COD Figure A.5.15).

```
module LEGv8ALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [63:0] A,B;
   output reg [63:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```

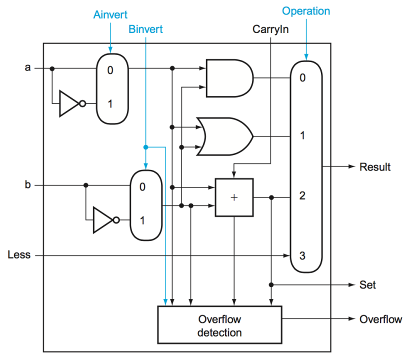
Figure 7.5.16: The LEGv8 ALU control: a simple piece of combinational control logic (COD Figure A.5.16).

```
module ALUControl (ALUOp, FuncCode, ALUCtl);

   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;

   always case (FuncCode)

   32: ALUOp<=2; // add
   34: ALUOp<=6; //subtract
   36: ALUOP<=0; // and
   37: ALUOp<=1; // or
   39: ALUOp<=12; // nor
   42: ALUOp<=7; // slt
   default: ALUOp<=15; // should not happen
   endcase
endmodule
```

The next question is, "How quickly can this ALU add two 64-bit operands?" We can determine the a and b inputs, but the CarryIn input depends on the operation in the adjacent 1-bit adder. If we trace all the way through the chain of dependencies, we connect the most significant bit to the least significant bit, so the most significant bit of the sum must wait for the *sequential* evaluation of all 64 1-bit adders. This sequential chain reaction is too slow to be used in time-critical hardware. The next section explores how to speed-up addition. This topic is not crucial to understanding the rest of the appendix and may be skipped.

PARTICIPATION ACTIVITY 7.5.1: Check yourself: Additional ALU operations.

Given the following 1-bit ALU:



1) What changes are needed to the ALU to support the operation NOT (a AND b), called NAND?

- ◯ No change. NAND is already supported by the ALU.
- ◯ The 4-to-1 multiplexor must be expanded to add another input , and then add new logic to calculate NAND.

(*1) This section is in original form.

Provide feedback on this section