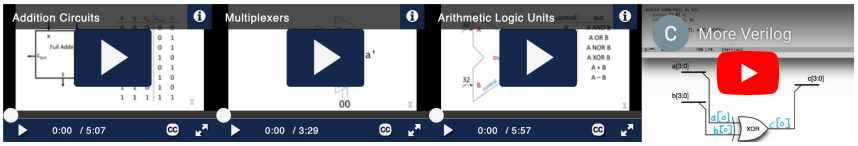
PRE03.1. 03 Text and Videos

Videos



Video credits: Geoffrey Herman and Craig Zilles, Text credits: Geoffrey Herman

The Big Picture

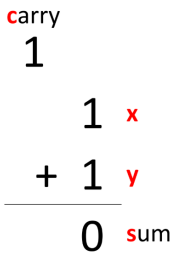
Computers can do two things: store state and manipulate state. The primary machanism for manipulating state is the Arithmetic Logic Unit.

The Arithmetic Logic Unit is the main muscle of the processor. It performs all of the arithmetic operations and bitwise logical operations that we taught you so far. To understand how an ALU works, you need to understand the concepts of data and control. Data is **what** information we are processing and control tells us **which** operation to perform in processing that data. For example, the ALU is responsible for implementing C code like `X = A + B;` or `X = A & B;`. The ALU takes in two pieces of data (e.g., `A` and `B`) and produces one new piece of data (e.g., `X`). We use control bits to decide which operation to perform (e.g., `+` or `&`). To help you recognize when a wire is treated as data or as control, we will use red for data and blue for control.

To build the ALU, we use modular design. We design both a 1-bit arithmetic unit and a 1-bit logic unit. We then chain those circuits together.

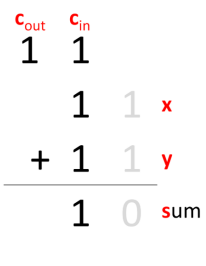
Binary Addition Circuit for one bit slice

If we want to add two N-bit binary numbers, we typically start from the least-significant bits from each number. If we add these two bits together, we have three possible outcomes: their sum is 0_{10}, 1_{10}, or 2_{10}. Because we cannot represent 2 (i.e., 10_2) with one bit, we carry the most significant bit from the addition to the next bit position. This process reveals that we need two output bits to represent the sum of these bits: a sum bit and a carry bit. We can represent the behavior of a circuit that implements this addition operation with the following truth table. We call the circuit represented by this truth table a half adder, because it does only half of the job of addition.



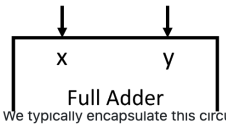
x	y	c	s	
0	0	0	0	$(0)_{10}$
0	1	0	1	$(1)_{10}$
1	0	0	1	$(1)_{10}$
1	1	1	0	$(2)_{10}$

If we continue our addition operation to the next position, we realize that we must add three bits together: `x`, `y`, and `cin`. Consequently, this operation can output the binary numbers 0-3. Fortunately, we can still encode those numbers with just two bits – our sum and carry bits. We can represent the behavior of a circuit that implements this addition operation with the following truth table. We call the circuit represented by this truth table a full adder.

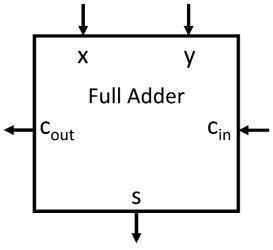


x	y	c_{in}	c_{out}	s	
0	0	0	0	0	$(0)_{10}$
0	0	1	0	1	$(1)_{10}$
0	1	0	0	1	$(1)_{10}$
0	1	1	1	0	$(2)_{10}$
1	0	0	0	1	$(1)_{10}$
1	0	1	1	0	$(2)_{10}$
1	1	0	1	0	$(2)_{10}$
1	1	1	1	1	$(3)_{10}$

We can implement this circuit using logic gates. We typically encapsulate this circuit as a module that has the following appearance, often using the abbreviation FA for Full Adder.

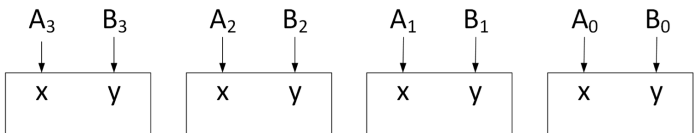


We can implement this circuit using logic gates. We typically encapsulate this circuit as a module that has the following appearance, often using the abbreviation FA for Full Adder.



N-bit binary addition circuits

To perform N-bit binary addition, we use modular design and chain N full adders together. We need to input 0 into the c_{in} of the full adder of the least-significant bits. For example, a 4-bit addition and an 8-bit addition circuit are shown below.



PRE 03

Assessment overview

Total points: 0/75

Score: 0%

Question PRE03.1

Value: 1

Total points: — /1

Auto-graded question

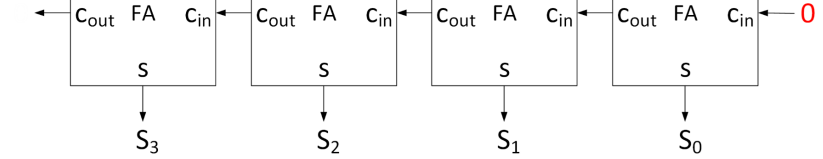
Previous question Next question

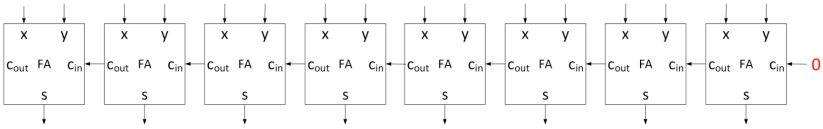
Personal Notes

No attached notes

Attach a file

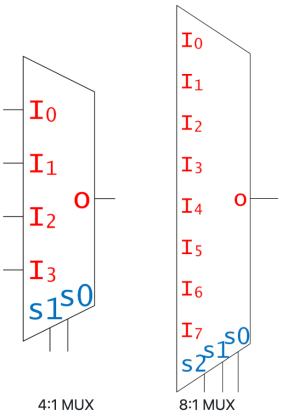
Add text note





Multiplexer (MUX)

A common design pattern in hardware is to compute many different values in parallel and then choose only the value that we need using a multiplexer (MUX). We call the collection of values that we want to select from **data inputs**. A MUX that receives N data inputs will need $\log_2 N$ **selection input** bits to encode which data input to select. We label the data inputs with decimal number subscripts starting with zero indexing (e.g., `I0` or `I2`) that should be interpreted as decimal numbers. We label the selection inputs with subscripts (e.g., `s2` or `s0`) that correspond to the exponents of the N-bit unsigned binary positional notation (i.e., `s2s1s0` correspond `s2*2^2+s1*2^1+s0*2^0`), where N is the number of selection bits. For example, if `s2s1s0 = 101`, it should be interpreted as 5 (i.e., `1*2^2+0*2^1+1*2^0`), which indicates that data input `I5` should be selected.



4:1 MUX behavior table

s_1	s_0	o
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

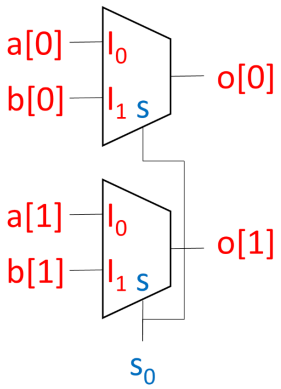
8:1 MUX behavior table

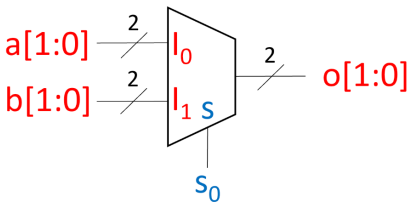
s_2	s_1	s_0	o
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

Buses and N-bit wide MUXes

To represent N-bit signals like with our N-bit adder, we bundle wires into groups called buses. A bus is often represented as a wire with a slash through it and labeled with the number of wires in that bus. Other times a bus is represented as a thicker wire in the circuit diagram. When wires are bundled as a bus, we use an array notation to indicate the bit positions of the wires. For example, if we had a 4-bit unsigned binary number 1011 stored on bus `A`, then `A[3]==1`, `A[2]==0`, `A[1]==1`, `A[0]==1`. If needed, we could indicate the size of bus `A` by writing it as `A[3:0]`. The 0 bit position is always interpreted as the least-significant bit position.

When performing operations like bitwise logical operations or other multi-bit operations, we may want to perform the same operation on every wire in the bus. For example, in the image below we are using a single selection input bit for multiple MUXes so that they either choose all of the `a` inputs or all of the `b` inputs (and never mix and match). This is fine to draw for a 2-bit signal but painful to draw for signals with more bits, so we use a shorthand like on the right where we show whole buses going into a single MUX. While the schematic shows only one MUX, in reality our circuit will have N MUXes. We call this a N-bit wide MUX. Whenever we use this shorthand with gates or MUXes, we are indicating that we are performing the same operation to every bit-slice in the bus.

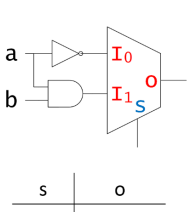


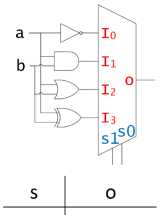


1-bit Logic Unit

A logic unit is a module that lets us choose between L bitwise logical operations. We implement a logic unit by creating a 1-bit logic unit and then replicating that module many times to make a N-bit logic unit. For example, we would use a 32-bit logic unit to perform bitwise logical operations on 32-bit `int` variables (e.g., `int A` and `int B`).

We implement a 1-bit logic unit by performing each of the bitwise logical operations we wish to perform on 1 bit from each of the two input variables (e.g., `a[0]` and `b[0]` from `A` and `B` respectively) and selecting the one we actually want to use with a multiplexer (see figure below). This design may be a little counterintuitive (isn't it wasteful to compute all of the bitwise logical operations just to throw most of them away?), but this design pattern of performing many calculations in parallel to ignore most of them is the fastest way to perform computations in hardware.





0	NOT a
1	a AND b

00	NOT a
01	a AND b
10	a OR b
11	a XOR b

A logic unit can have an arbitrary size or ordering of Boolean operations. As we build towards implementing a MIPS datapath, we will implement a logic unit with the following implementation.

s	o
00	a AND b
01	a OR b
10	a NOR b
11	a XOR b

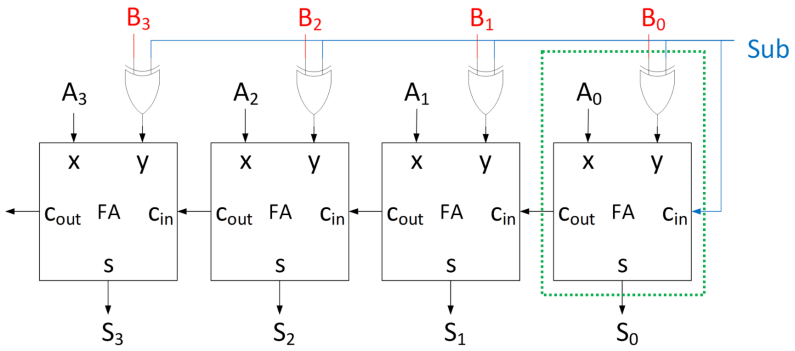
Arithmetic Logic Unit (ALU)

The computational muscle of the computer is the Arithmetic Logic Unit (ALU). The ALU combines several smaller modules - Full Adders, XOR gates, MUXes, and Logic Units - to create a circuit that lets us choose between a set of basic arithmetic operations (addition and subtraction) and bitwise logical operations (AND, OR, NOR, XOR). We will design a 1-bit ALU and then chain those ALUs together to create N-bit ALUs.

Arithmetic Unit

Below is a 4-bit arithmetic unit that can perform 4-bit addition and subtraction. `Sub` is a control signal that makes the circuit perform addition when it is 0 and subtraction when it is 1. When `Sub==0`, the XOR gates do not complement the B_i signals, so the adders perform `A+B`. When `Sub==1`, the XOR gates bitwise complement the B_i signals, so that the adders perform `A+~B`. Also, the `Sub` signal is an input to the least-significant c_{in}, which adds 1 when `Sub==1`. Together, when `Sub==1` the adders perform `A+~B+1`

Below is a 4-bit arithmetic unit that can perform 4-bit addition and subtraction. `Sub` is a control signal that makes the circuit perform addition when it is 0 and subtraction when it is 1. When `Sub==0`, the XOR gates do not complement the B_i signals, so the adders perform `A+B`. When `Sub==1`, the XOR gates bitwise complement the B_i signals, so that the adders perform `A+~B`. Also, the `Sub` signal is an input to the least-significant c_{in}, which adds 1 when `Sub==1`. Together, when `Sub==1` the adders perform `A+~B+1 = A+(~B+1) = A-B`.

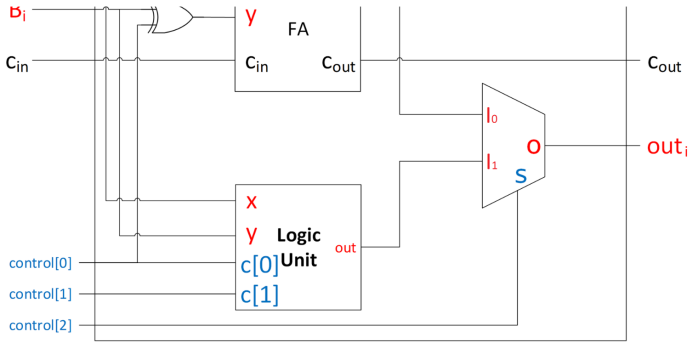


We can take just the full adder and the XOR gate (green-dotted box) to make a 1-bit ALU.

1-bit Arithmetic Logic Unit

To construct a 1-bit ALU, we combine a 1-bit arithmetic and a 1-bit logic unit by using a 2:1 MUX. This is but one example of a 1-bit ALU. There are many different possibilities, but this is the one we will use in this class. This 1-bit ALU uses a 1-bit logic unit with the following behavior table.

c[1:0]	o
00	a AND b
01	a OR b
10	a NOR b
11	a XOR b



By combining the arithmetic unit with the logic unit using a MUX, the behavior table of this 1-bit ALU would be the following. Notice that each 1-bit ALU does not perform subtraction on its own, but performs a_i plus the complement of b_i. Without the extra "+1", the subtraction is incomplete. The "+1" only happens at the N-bit ALU level.

control[2:0]	out_i
000	undefined
001	undefined
010	$a_i + b_i$
011	$a_i + \sim b_i$
100	a_i AND b_i
101	a_i OR b_i
110	a_i NOR b_i
111	a_i XOR b_i

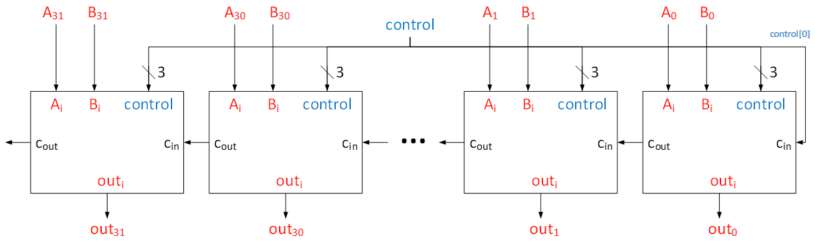
In this behavior table, undefined means that any behavior that the circuit does is okay. It DOES NOT mean that the circuit has an uncertain output. If you trace the circuit carefully, you will see that `control==000` also implements addition and `control=001` also implements subtraction. This redundancy is okay because we left that behavior as undefined.

Implements subtraction. This redundancy is okay because we left that behavior as undefined.

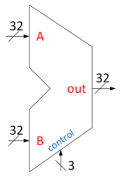
N-bit ALU

We construct N-bit ALUs by chaining together N 1-bit ALUs. The image below shows a 32-bit ALU. Notice how each 1-bit ALU receives different bits from buses `A[31:0]` and `B[31:0]` but every 1-bit ALU receives the same `control` signal. By giving each 1-bit ALU the same control signal, every 1-bit ALU acts as part of a whole performing the same operation.

Notice how `control[0]` is sent to the c_{in} of the least-significant bit just like what we did with the `Sub` signal in the arithmetic unit. When c_{in}==1, it provides the "+1" in `A + (~B + 1)` to implement subtraction. c_{in} is `Sub` for the Arithmetic Unit portion of the ALU.



Rather than draw all 32, 1-bit ALUs, we draw an ALU with the following shape with buses.



In this example, the 32-bit ALU has the following behavior table.

control[2:0]	out
000	undefined
001	undefined
010	A + B
011	A - B
100	A AND B

Mark as read

- [ ] (a) I've read this!

Select all possible options that apply.

Save & Grade *Unlimited attempts* | Save only