LECTURE 4

Logic Design

LOGIC DESIGN

We already know that the language of the machine is *binary* – that is, sequences of 1's and 0's. But why is this?

At the hardware level, computers are streams of signals. These signals only have two states of interest, high voltage and low voltage.

Binary is merely a natural abstraction for the underlying signals. Rather than talking about voltage levels, we talk about logically true signals (having a value 1) and logically false signals (having a value 0).

We might also say that logically true signals are *asserted*, while logically false signals are *deasserted*.

LOGIC BLOCKS

Logic Blocks are programmable logic components which take some input and produce some output according to a set of logical rules.

- *Combinational Logic Blocks* – depend only on a set of inputs. Any given input will always result in the same output.

- *Sequential Logic Blocks* – maintain an internal state, which may affect the output obtained for a given set of input values.

TRUTH TABLES

Defining a combinational logic block is as simple as defining the output values for all of the possible sets of input values.

Because our input takes only one of two values – 0 or 1 – for n inputs, there are 2^n possible input combinations.

As long as we can define the output for each of these combinations, our combinational logic block is fully defined.

TRUTH TABLES

Consider a logic function with three inputs, A, B, and C, and three outputs, D, E, and F. D is true if at least one input is true, E is true if exactly two inputs are true, and F is true only if all three inputs are true. Show the truth table for this function.

The truth table will contain $2^3 = 8$ entries.

TRUTH TABLES

	Inputs			Outputs	
A	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0
1	0	0	1	0	0
0	1	0	1	0	0
0	0	1	1	0	0
1	1	0	1	1	0
1	0	1	1	1	0
0	1	1	1	1	0
1	1	1	1	0	1

BOOLEAN ALGEBRA

We can also express logic functions using Boolean algebra. In Boolean algebra, all variables can either have the value 0 or 1. We also have the following operations available:

- The OR operator: $A + B$. The result is 1 if either of the variables is 1. Also known as a logical sum.

- The AND operator: $A \cdot B$. The result is 1 if both of the variables are 1. Also known as a logical product.

- The NOT operator: $\bar{A}$. The result is 1 only if the value of the variable is 0.

BOOLEAN ALGEBRA

The following laws and identities may be helpful in manipulating logic equations:

- Identity laws: $A + 0 = A$ and $A \cdot 1 = A$.

- Zero and One laws: $A + 1 = 1$ and $A \cdot 0 = 0$.

- Inverse laws: $A + \bar{A} = 1$ and $A \cdot \bar{A} = 0$.

- Commutative laws: $A + B = B + A$ and $A \cdot B = B \cdot A$.

- Associative laws: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.

- Distributive laws: $A \cdot (B + C) = A \cdot B + A \cdot C$ and $A + (B \cdot C) = (A \cdot B) + (A \cdot C)$.

DEMORGAN'S LAWS

Additionally, we have the transformation rules:

- $\overline{A \cdot B} \leftrightarrow \bar{A} + \bar{B}$

- $\overline{A + B} \leftrightarrow \bar{A} \cdot \bar{B}$

LOGIC EQUATIONS

Consider a logic function with three inputs, A, B, and C, and three outputs, D, E, and F. D is true if at least one input is true, E is true if exactly two inputs are true, and F is true only if all three inputs are true. Write the logic equations for D, E, and F.

LOGIC EQUATIONS

Consider a logic function with three inputs, A, B, and C, and three outputs, D, E, and F. D is true if at least one input is true, E is true if exactly two inputs are true, and F is true only if all three inputs are true. Write the logic equations for D, E, and F.

- $D = A + B + C$
- $E = ((A \cdot B) + (B \cdot C) + (C \cdot A)) \cdot (\overline{A \cdot B \cdot C})$
- $F = A \cdot B \cdot C$

LOGIC GATES

Logic blocks are built from *logic gates* which implement basic logic functionality.

AND

A
B
C

$$C = A \cdot B$$

OR

A
B
C

$$C = A + B$$

NOT

A
C

$$C = \bar{A}$$

Here our AND and OR gates accept two input values, but since AND and OR are both commutative and associative, they can have any number of input values.

LOGIC GATES

Note that it is common to avoid explicit NOT gates in favor of bubbles around the input output lines. For example,

can also be represented as

What logic equation does this gate sequence represent?

LOGIC GATES

Note that it is common to avoid explicit NOT gates in favor of bubbles around the input output lines. For example,

can also be represented as

What logic equation does this gate sequence represent?

$$C = \overline{\bar{A} + B}$$

LOGIC GATES

Any logical function can be represented using only the AND, OR, and NOT gates.

Furthermore, all logical functions can be constructed with only a single gate type, as long as the gate is an inverting gate with multiple inputs. Two common gates that fit these criteria are NOR and NAND. These gates are known as *universal* gates.

- NOR implements the logical function $C = \overline{A + B}$

- NAND implements the logical function $C = \overline{A \cdot B}$

<u>Fun side note</u>: you can make a similar simple statement about quantum computers! It is possible to implement any quantum gate using only single-bit gates and a two-bit controlled-NOT gate. This is a *universal* quantum gate set.

DECODERS

A *decoder* is a logic block with n input bits and 2^n output bits. Only one output bit is *set*, or *asserted*, for each combination of input bits.
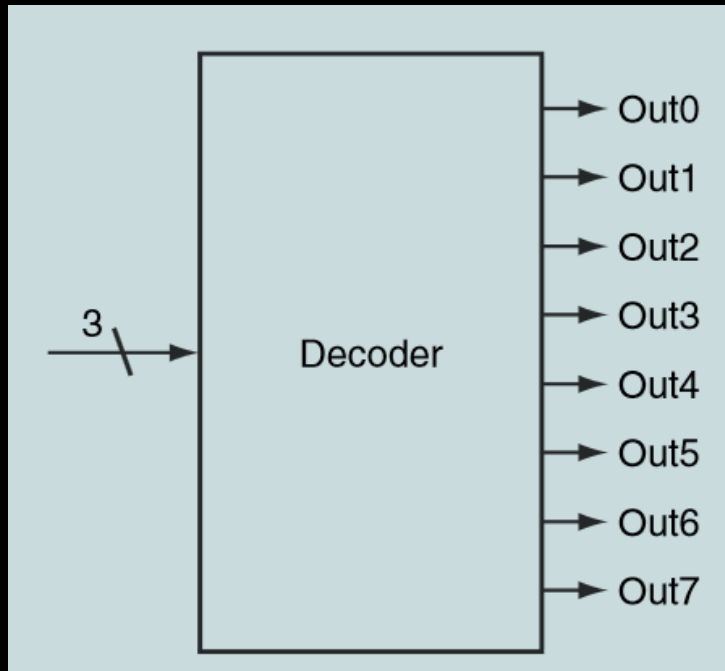

A decoder essentially translates the input signal into a signal that corresponds to the binary value of the *n*-bit input.

For example, let's say the input signal is 011 for a decoder accepting 3 bits. This corresponds to the decimal value 3. So, the output signal becomes 00001000. The output signal has $2^3 = 8$ bits and all are zeroed out except for the bit at index 3 (where index 0 is the rightmost bit).

DECODERS

Here is the truth table for the example decoder depicted in the previous slide.
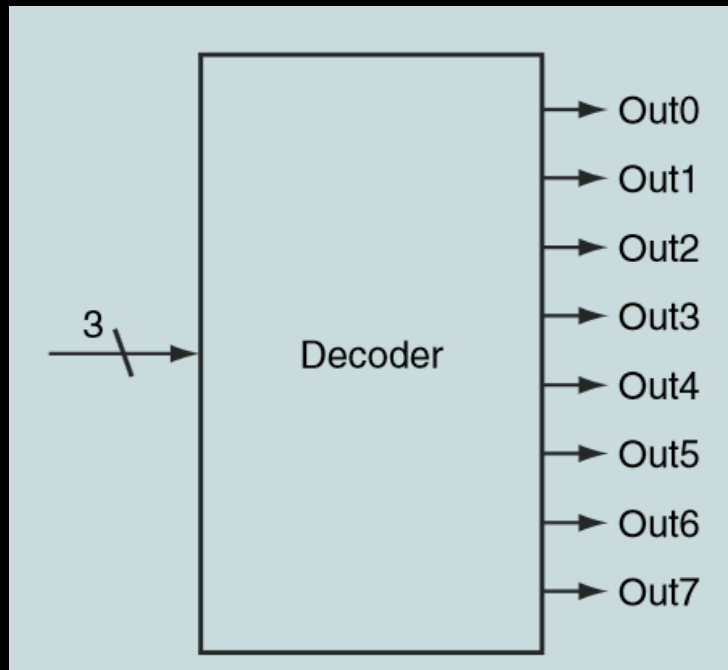
| In0	In1	In2	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

MULTIPLEXORS

Multiplexors, or *selectors*, are logic functions whose output value is one of its input values, determined by a selector value.
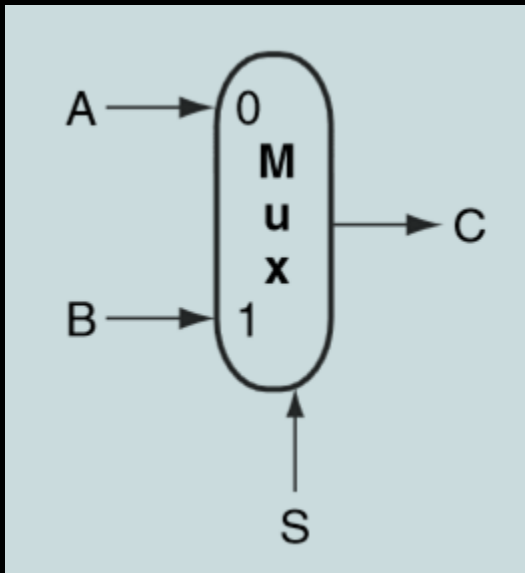
The multiplexor on the left has two input values, A and B. There is an additional input value S, the *selector (or control) value*. The selector value determines which of the input values, A or B, will be used as the output value.

MULTIPLEXORS

As stated before, all logic functions can be implemented using only AND, OR, and NOT. Below is the gate implementation of our example multiplexor.
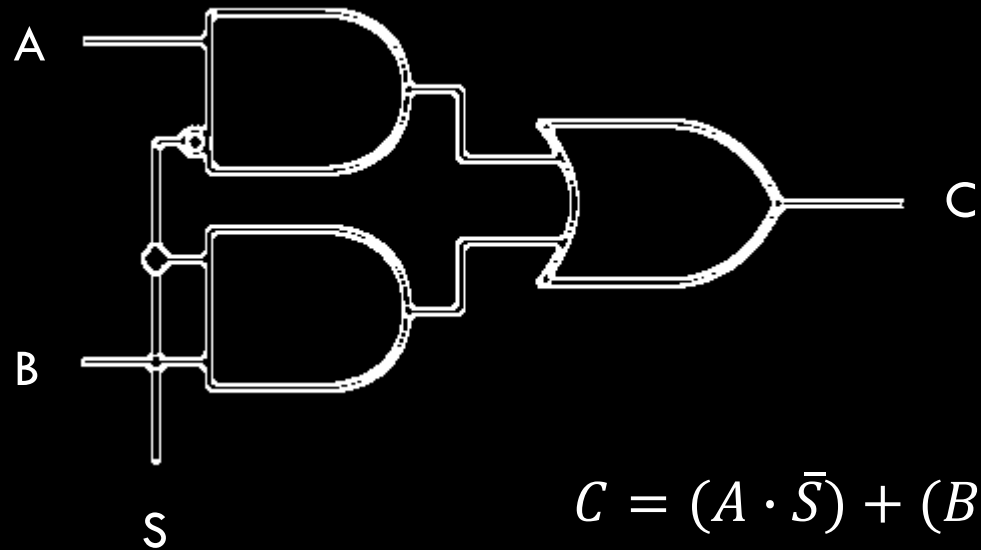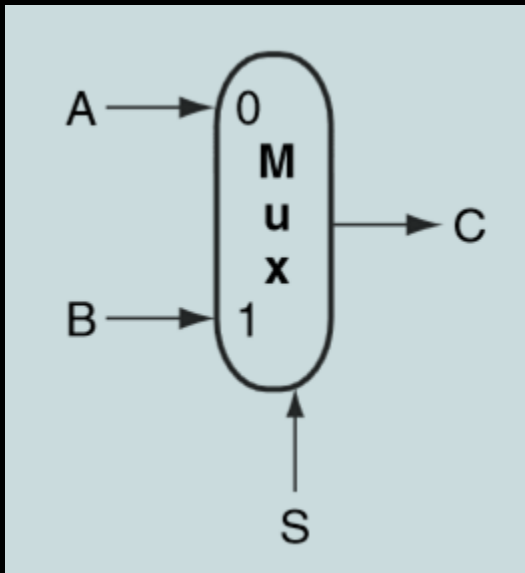
$$C = (A \cdot \bar{S}) + (B \cdot S)$$

MULTIPLEXORS

Clearly, with only two data inputs, our selector can uniquely identify the selected input value using only a single selector input. We can select A if S is false (0) and B if S is true (1). But what if we want more than two data inputs?

To uniquely identify each of n data input values, we'll need $\lceil \log_2 n \rceil$ selector input values.

It's easy to convince ourselves of this. Using n bits, we can represent the decimal range 0 through $(2^n - 1)$. Therefore, $\log_2 n$ bits can be used to represent the range 0 through $(2^{\log_2 n} - 1)$, or 0 through $(n - 1)$.

MULTIPLEXORS

So, to implement a multiplexor with n data inputs and $\lceil \log_2 n \rceil$ selector inputs, we can implement the following.

- A decoder that generates n signals, each indicating a different input value.

- An array of n AND gates, each combining one of the inputs with a signal from the decoder.

- A large OR gate that takes an input all of the outputs of the AND gates.

MULTIPLEXORS

As an example, let's say we want to implement a multiplexor which accepts 4 input bits. We will need 2 selector bits. The selector bits can take on the values 00, 01, 10, 11. Let's say our selector-bits decoder uses the following truth table.

S0	S1	Out3	Out2	Out1	Out0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

We associate Out0 with data input A, Out1 with data input B, and so on. So our multiplexor can be implemented with the following logic equation:

$$(A \cdot Out0) + (B \cdot Out1) + (C \cdot Out2) + (D \cdot Out3)$$

If the selector bits are 10, the decoder will give us the output bits 0100. Only C will be logically multiplied by 1, zeroing out the other contributions.

TWO-LEVEL LOGIC

We already know that we can implement any logic function using only AND, OR, and NOT gates.

Furthermore, we can write any logic function in a standard form which has the following features:

- Every input is either a true or complemented variable (i.e. A or $\bar{A}$).

- There are only two levels of gates – one being AND and the other being OR.

- Possibly a negation on the final output.

This canonical form is known as *two-level representation*.

TWO-LEVEL LOGIC

There are two alternative forms of two-level representation.

- *Sum of Products*
  - A logical sum (OR) is taken over a collection of logical products (AND).
  - Example: $(A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) + (B \cdot C \cdot \bar{A})$

- *Product of Sums*
  - A logical product (AND) is taken over a collection of logical sums (OR).
  - Example: $\overline{(\bar{A} + \bar{B} + C) \cdot (\bar{A} + \bar{C} + B) \cdot (\bar{B} + \bar{C} + A)}$

TWO-LEVEL LOGIC

What is the advantage of using this canonical form of two-level representation? Take for example the logic equation we found for E a few slides ago.

$$E = ((A \cdot B) + (B \cdot C) + (C \cdot A)) \cdot (\overline{A \cdot B \cdot C})$$

This equation has three levels of logic. Think of it this way: we must first perform $(A \cdot B)$, $(B \cdot C)$, and $(C \cdot A)$. The results of these are logically summed, after which the result is logically multiplied with another Boolean expression. So we have three steps.

The canonical form below has only two steps, but performs the same logical function:

$$E = (A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) + (B \cdot C \cdot \bar{A})$$

TWO-LEVEL LOGIC

It's a bit harder to see why every logical function can be represented in the canonical forms so let's look at an example.

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Here, we have three input values A, B, and C and output value D. Let's try to construct the sum-of-products representation of D.

TWO-LEVEL LOGIC

Sum-of-products representation of D:

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

⬅ T1 (row 0 0 1 1)
⬅ T2 (row 0 1 0 1)
⬅ T4 (row 1 0 0 1)
⬅ T7 (row 1 1 1 1)

First, we note that there are only four input combinations which result in a value of true for D. These combinations are 001, 010, 100, and 111. We will just refer to them as T1, T2, T4, and T7. So, we can at least say the following:

$$D = T1 + T2 + T4 + T7$$

In other words, if have any of those sequences as input, then D must be true.

TWO-LEVEL LOGIC

Sum-of-products representation of D:

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

⬅ T1

⬅ T2

⬅ T4

⬅ T7

Now, we can express each of our sequences in terms of the input values. Take T1 for example.

$$T1 = \bar{A} \cdot \bar{B} \cdot C$$

In other words, T1 is only true if A is false *and* B is false *and* C is true. For the other sequences, we have the following:

$$T2 = \bar{A} \cdot B \cdot \bar{C}$$

$$T4 = A \cdot \bar{B} \cdot \bar{C}$$

$$T7 = A \cdot B \cdot C$$

TWO-LEVEL LOGIC

Sum-of-products representation of D:

A	B	C	D
0	0	0	0
0	0	1	1	⬅ T1
0	1	0	1	⬅ T2
0	1	1	0
1	0	0	1	⬅ T4
1	0	1	0
1	1	0	0
1	1	1	1	⬅ T7

Finally, we can replace our T terms to get the following equation for D:

$$D = (\bar{A} \cdot \bar{B} \cdot C) + (\bar{A} \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot B \cdot C)$$

This is our canonical sum-of-products representation of D.

PROGRAMMABLE LOGIC ARRAYS

The sum-of-products representation is implemented by the *programmable logic array (PLA)*. A PLA is composed of:

- A set of inputs and corresponding input complements.

- An array of AND gates that implement the first level of logic and form a set of *product terms*, or *minterms*.

- An array of OR gates, each of which forms a logical sum of any number of minterms.

Note that the contents of a PLA are fixed when the PLA is constructed but an equivalent structure called a PAL can be programmed electronically.

PROGRAMMABLE LOGIC ARRAYS

A PLA can directly implement the truth table of a set of logic functions with multiple inputs and outputs. Let's look at an example using the truth table from earlier.
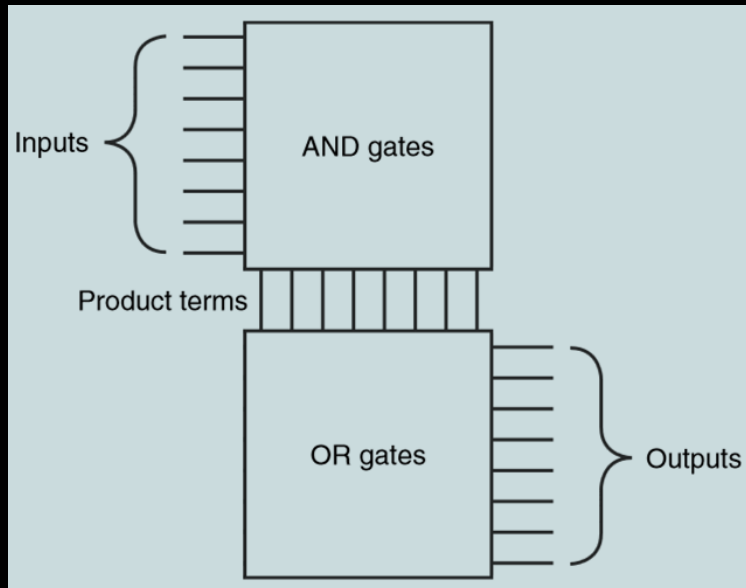
A	B	C	D	E	F
0	0	0	0	0	0
1	0	0	1	0	0
0	1	0	1	0	0
0	0	1	1	0	0
1	1	0	1	1	0
1	0	1	1	1	0
0	1	1	1	1	0
1	1	1	1	0	1

A, B, and C are input values. D, E, and F are output values.

PROGRAMMABLE LOGIC ARRAYS

A	B	C	D	E	F	
0	0	0	0	0	0	
1	0	0	1	0	0	← T1
0	1	0	1	0	0	← T2
0	0	1	1	0	0	← T3
1	1	0	1	1	0	← T4
1	0	1	1	1	0	← T5
0	1	1	1	1	0	← T6
1	1	1	1	0	1	← T7

To construct the PLA, we can perform a process similar to constructing the sum-of-products representation of a logical equation.

First we note that there are only seven sequences which result in a truth value for any of the output values D, E, or F.

PROGRAMMABLE LOGIC ARRAYS

A	B	C	D	E	F	
0	0	0	0	0	0	
1	0	0	1	0	0	← T1
0	1	0	1	0	0	← T2
0	0	1	1	0	0	← T3
1	1	0	1	1	0	← T4
1	0	1	1	1	0	← T5
0	1	1	1	1	0	← T6
1	1	1	1	0	1	← T7

The logic functions for D, E, and F can be expressed in terms of these sequences.

$$D = T1 + T2 + T3 + T4 + T5 + T6 + T7$$

$$E = T4 + T5 + T6$$

$$F = T7$$

PROGRAMMABLE LOGIC ARRAYS

A	B	C	D	E	F
0	0	0	0	0	0
1	0	0	1	0	0
0	1	0	1	0	0
0	0	1	1	0	0
1	1	0	1	1	0
1	0	1	1	1	0
0	1	1	1	1	0
1	1	1	1	0	1

⬅ T1
⬅ T2
⬅ T3
⬅ T4
⬅ T5
⬅ T6
⬅ T7

The logic functions for D, E, and F can be expressed in terms of these sequences.

$$D = T1 + T2 + T3 + T4 + T5 + T6 + T7$$

$$E = T4 + T5 + T6$$

$$F = T7$$

And each sequence can be expressed in terms of A, B, and C. For example,

$$T1 = A \cdot \bar{B} \cdot \bar{C}$$

PROGRAMMABLE LOGIC ARRAYS

A	B	C	D	E	F
0	0	0	0	0	0
1	0	0	1	0	0
0	1	0	1	0	0
0	0	1	1	0	0
1	1	0	1	1	0
1	0	1	1	1	0
0	1	1	1	1	0
1	1	1	1	0	1

⟸ T1
⟸ T2
⟸ T3
⟸ T4
⟸ T5
⟸ T6
⟸ T7

So, our products (or minterms) are:

$$T1 = A \cdot \bar{B} \cdot \bar{C}$$
$$T2 = \bar{A} \cdot B \cdot \bar{C}$$
$$T3 = \bar{A} \cdot \bar{B} \cdot C$$
$$T4 = A \cdot B \cdot \bar{C}$$
$$T5 = A \cdot \bar{B} \cdot C$$
$$T6 = \bar{A} \cdot B \cdot C$$
$$T7 = A \cdot B \cdot C$$

And our sums of products are:

$$D = T1 + T2 + T3 + T4 + T5 + T6 + T7$$
$$E = T4 + T5 + T6$$
$$F = T7$$

PROGRAMMABLE LOGIC ARRAYS
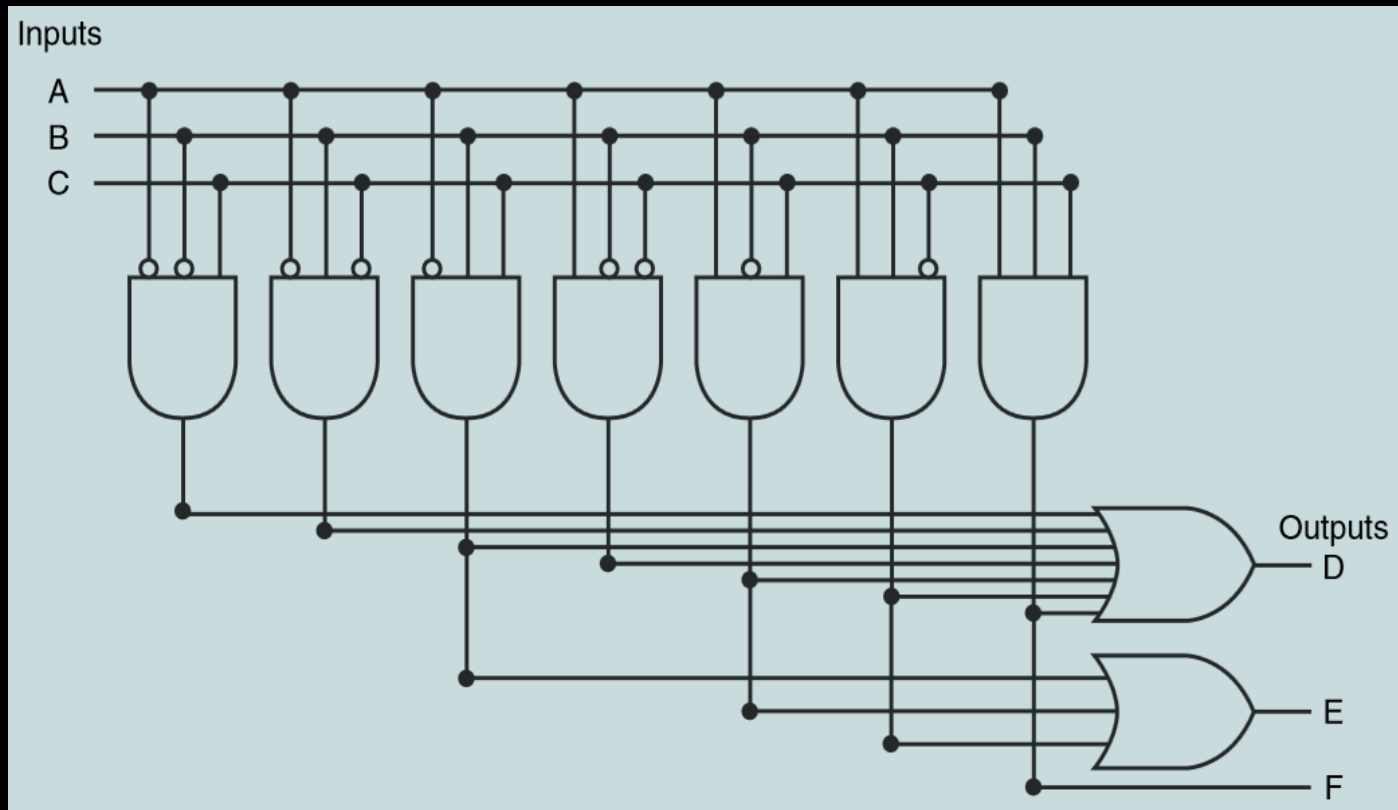
So, our products (or minterms) are:

$$T1 = A \cdot \bar{B} \cdot \bar{C}$$
$$T2 = \bar{A} \cdot B \cdot \bar{C}$$
$$T3 = \bar{A} \cdot \bar{B} \cdot C$$
$$T4 = A \cdot B \cdot \bar{C}$$
$$T5 = A \cdot \bar{B} \cdot C$$
$$T6 = \bar{A} \cdot B \cdot C$$
$$T7 = A \cdot B \cdot C$$

And our sums of products are:

$$D = T1 + T2 + T3 + T4 + T5 + T6 + T7$$
$$E = T4 + T5 + T6$$
$$F = T7$$

ARITHMETIC LOGIC UNIT

The Arithmetic Logic Unit (ALU) is the central component of the computing process – it performs all of the arithmetic and logical operations.

We can construct an ALU using only the AND, OR, NOT, and multiplexor logic blocks.

Since the MIPS word is 32 bits, our ALU needs to handle 32 bit inputs but we can start by creating a 1 bit ALU and then extend our ALU for 32 bits.

ARITHMETIC LOGIC UNIT

We start with implementing an ALU that performs AND and OR operations. Our input values are a and b. Our circuit performs $a \cdot b$ and $a + b$. The actual result is selected using a multiplexor where the selector value 0 indicates an AND operation and the selector value 1 indicates an OR operation.
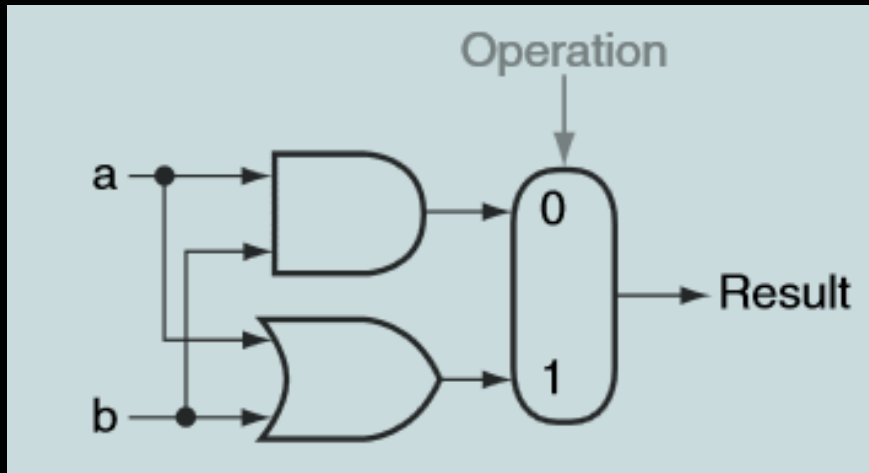
If a has the value 1 and b has the value 0, then a selector value of 1 will cause the result to be 1 while a selector value of 0 will cause the result to be 0.

ARITHMETIC LOGIC UNIT

Now we need to add addition to our ALU. We will represent the *adder* as a *black box* which hides the implementation details of addition except to say that our adder must accept two inputs for the operands and have one output for the result.

We must additionally include a *CarryIn* input and a *CarryOut* output. Why do we need these? Consider the example below.
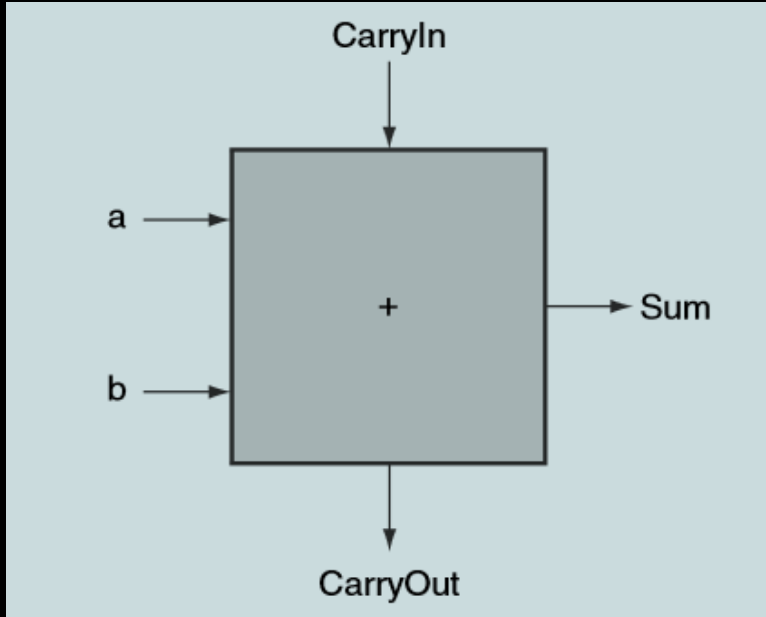
$$1\ 0\ 1\ 0$$
$$\underline{+\quad\ \ 1\ 1}$$

ARITHMETIC LOGIC UNIT

Now we need to add addition to our ALU. We will represent the *adder* as a *black box* which hides the implementation details of addition except to say that our adder must accept two inputs for the operands and have one output for the result.

We must additionally include a *CarryIn* input and a *CarryOut* output. Why do we need these? Consider the example below.

```
   1  0  1  0
+        1  1
_____
            1
```

ARITHMETIC LOGIC UNIT

Now we need to add addition to our ALU. We will represent the *adder* as a *black box* which hides the implementation details of addition except to say that our adder must accept two inputs for the operands and have one output for the result.

We must additionally include a *CarryIn* input and a *CarryOut* output. Why do we need these? Consider the example below.

```
    ①  ←——— Carry!
  1 0 1 0
+     1 1
—————————
      0 1
```

Note that our adder here only works with single bit operands, so we must be able to pass the carry around when necessary.

ARITHMETIC LOGIC UNIT

Now we need to add addition to our ALU. We will represent the *adder* as a *black box* which hides the implementation details of addition except to say that our adder must accept two inputs for the operands and have one output for the result.

We must additionally include a *CarryIn* input and a *CarryOut* output. Why do we need these? Consider the example below.

```
        1
    1 0 1 0
+       1 1
  _____
    1 1 0 1
```

ARITHMETIC LOGIC UNIT

Given the input bits a, b, and CarryIn, we can actually construct a truth table for the bits CarryOut and Sum.

a	b	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
1	0	0	0	1
1	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	1	0	1	0
1	1	1	1	1

Using our procedure for constructing sum-of-product canonical forms, we can easily determine that CarryOut is given by the following logical equation:

$$CarryOut = (a \cdot CarryIn) + (b \cdot CarryIn) + (a \cdot b)$$

Question: Why can we leave out the last minterm?

ARITHMETIC LOGIC UNIT

Alright, so we have an equation for the CarryOut. Therefore, our adder at least has the following hardware:
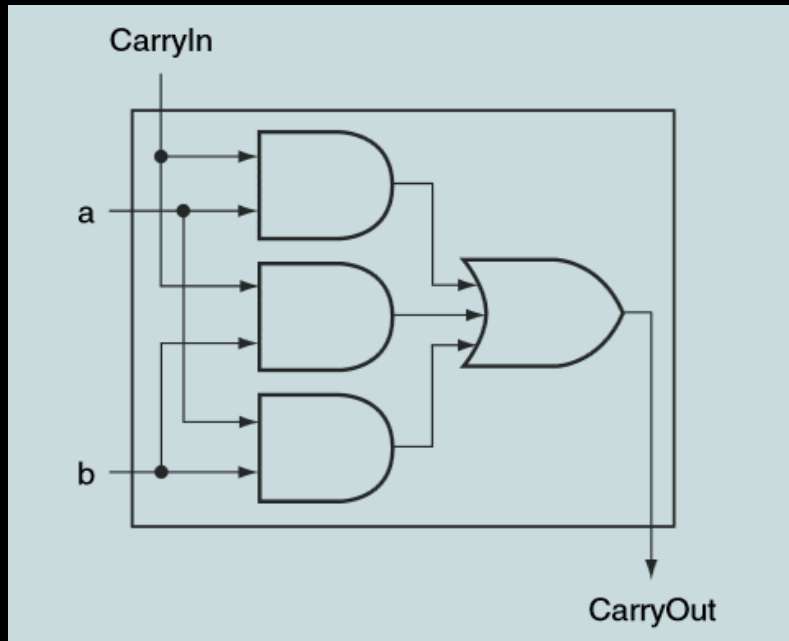
$$CarryOut = (a \cdot CarryIn) + (b \cdot CarryIn) + (a \cdot b)$$

Note that the summation is left out of this diagram. If we look back at our truth table, we can obtain the following equation for the sum (try it out!). Try to design the hardware to implement this logic equation for Sum.

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

ARITHMETIC LOGIC UNIT
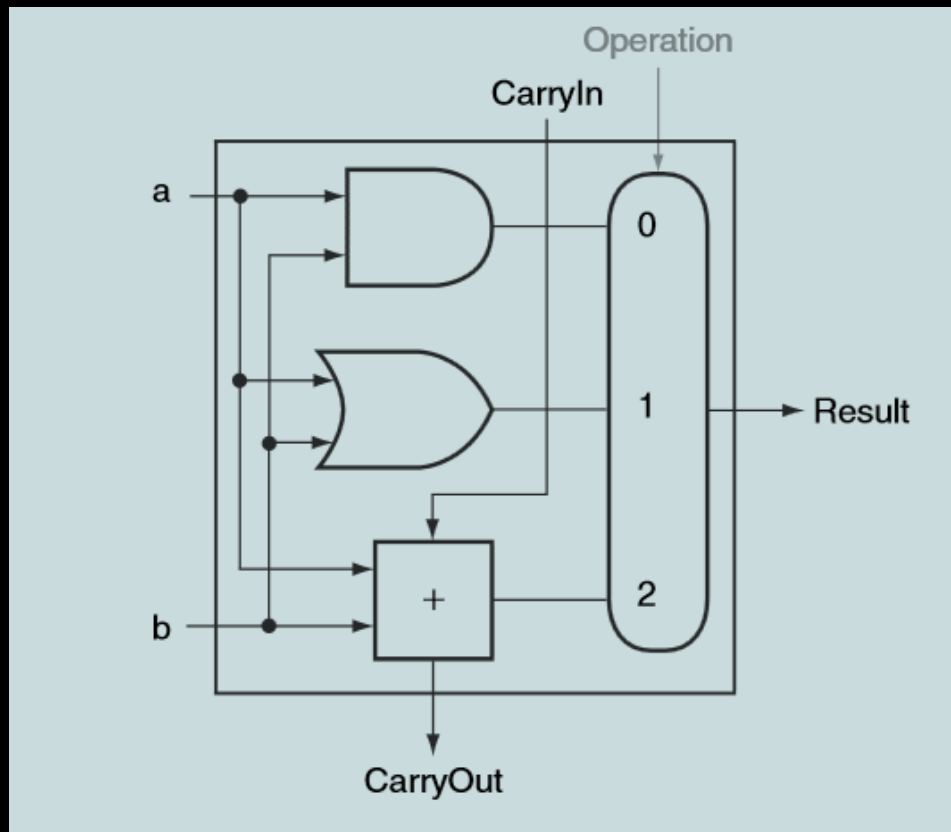
Once we've implemented the CarryOut and Sum logic functions, then we can combine our adder (shown here as a black box again) with the logical operations to create a simple 1-bit ALU.

In reality, ALUs tend to have more features – these are simply added as another selector value to the multiplexor.

ARITHMETIC LOGIC UNIT

Our 1-bit ALU can now perform AND, OR, and addition. To add subtraction to the mix, we need only to add the ability to negate the second operand b.
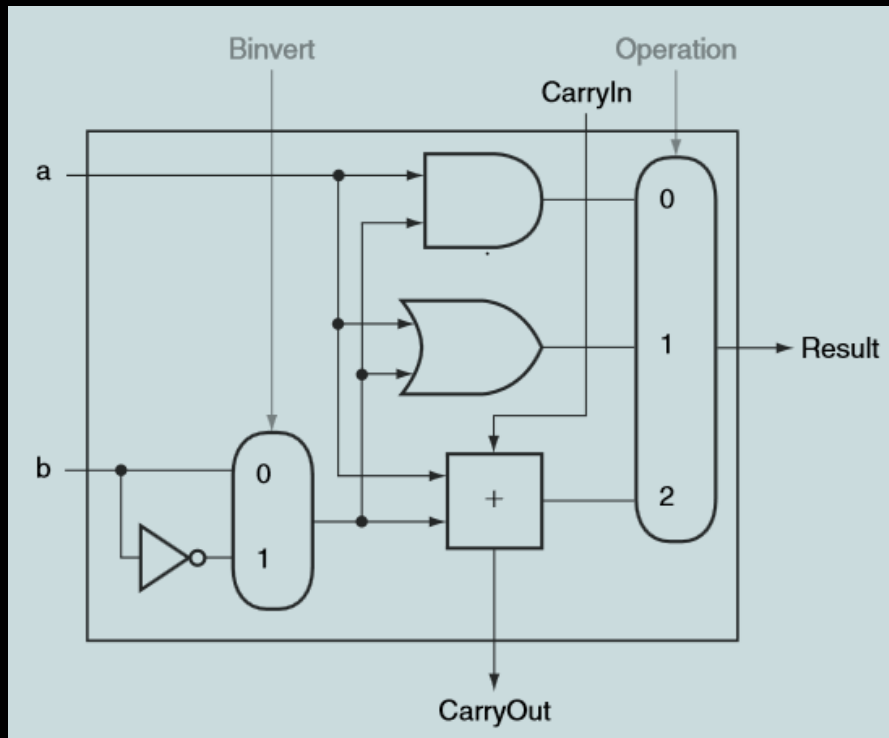
Recall the rules for negating a two's complement number:
- Invert each individual bit.
- Add 1.

Now, we can subtract by negating $b0, b1, b2, \dots b31$ and setting the first CarryIn value to 1.

ARITHMETIC LOGIC UNIT

Now we need to add the ability to negate a.

The reason for this is so that we can implement NOR. DeMorgan's laws tell us that $\overline{(a + b)} = \bar{a} \cdot \bar{b}$.

Now, we almost have a MIPS-compliant 1-bit ALU.
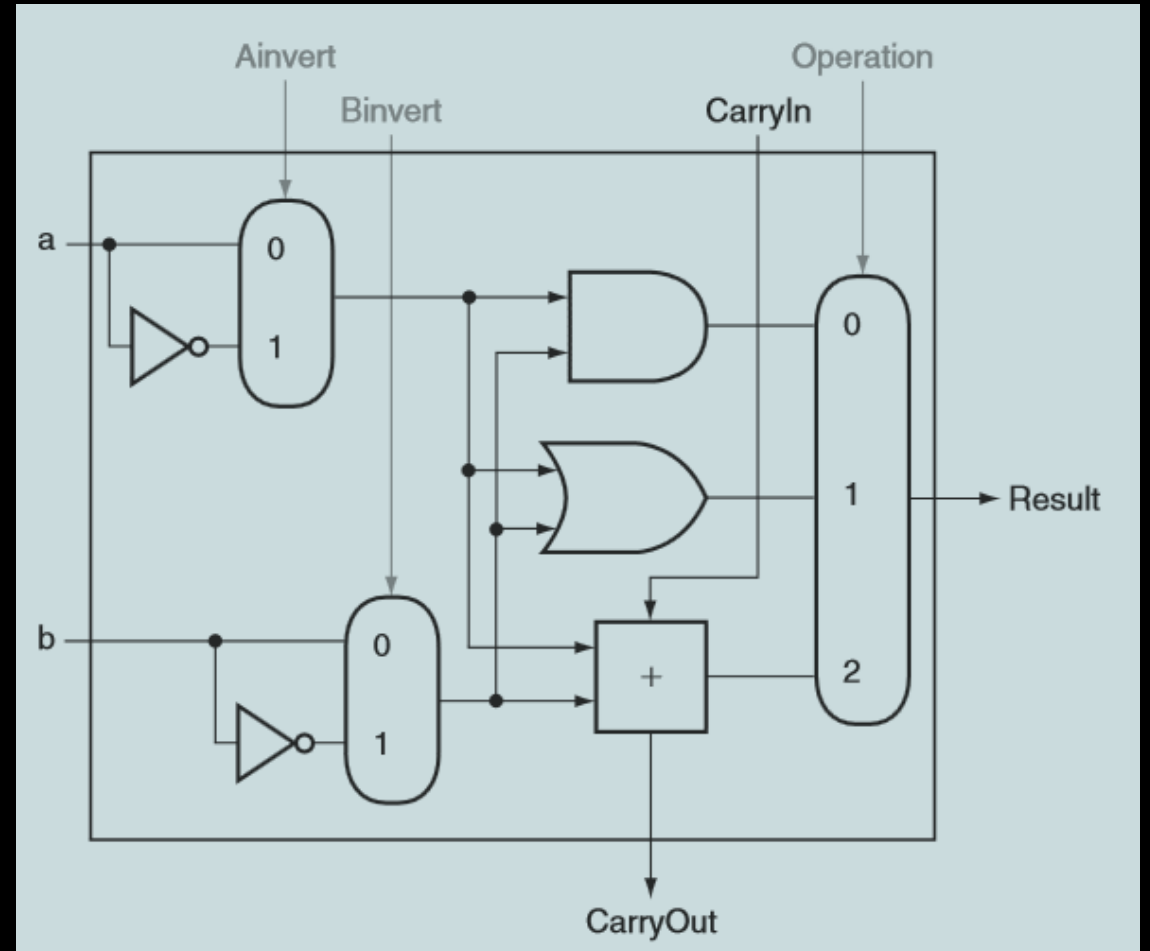
ARITHMETIC LOGIC UNIT

We just need to add support for the `slt` operation (and overflow detection).

We do this by adding a new input value called Less, which is just for `slt` instructions. We also add a new output to the ALU called Set.

<u>Here's how it works</u>: Less always has the value 0. If the selector value to the multiplexor is 3, then the Result will be 0 for every bit. The only exception is for the first bit, whose Less input will take the Set value of the 32nd bit (1 if a-b is negative and 0 if a-b is positive).
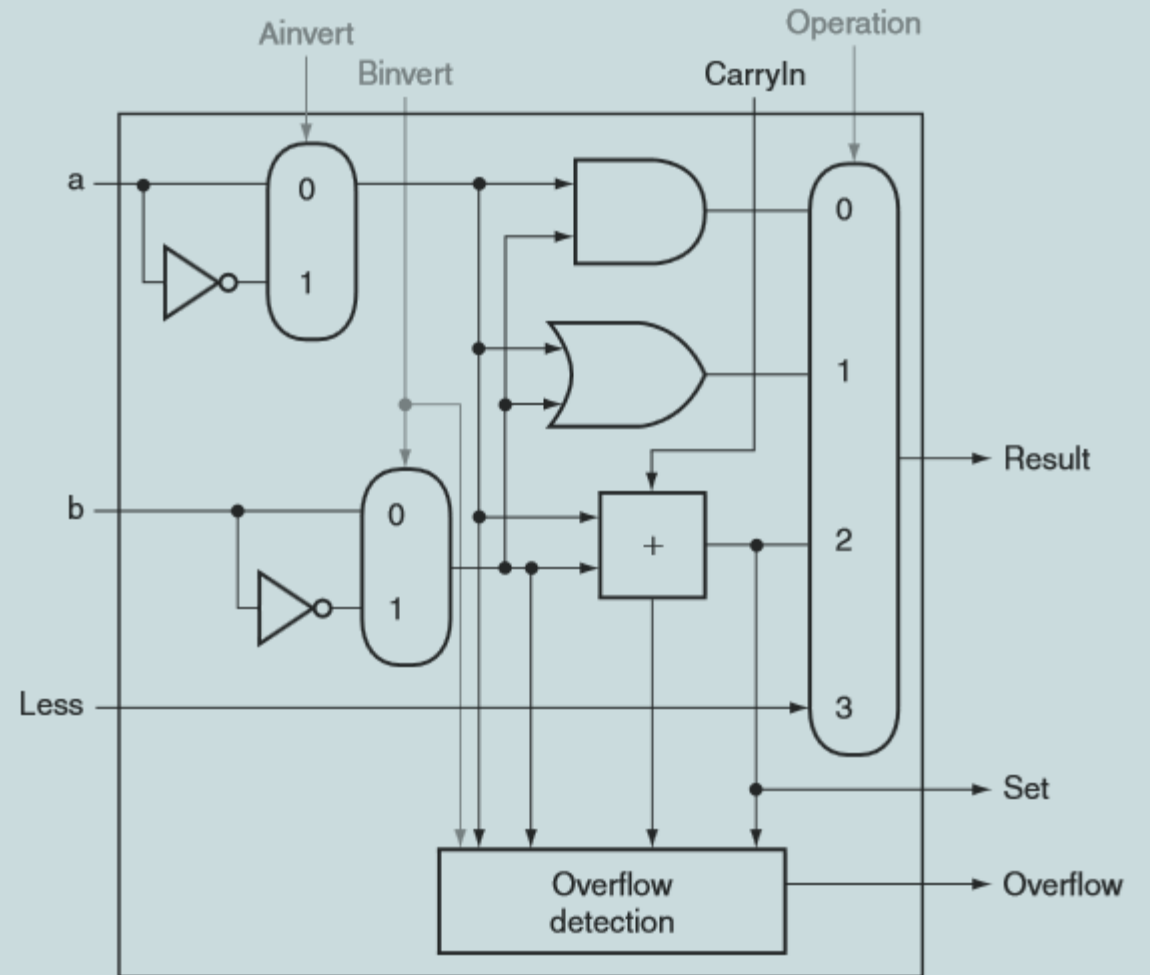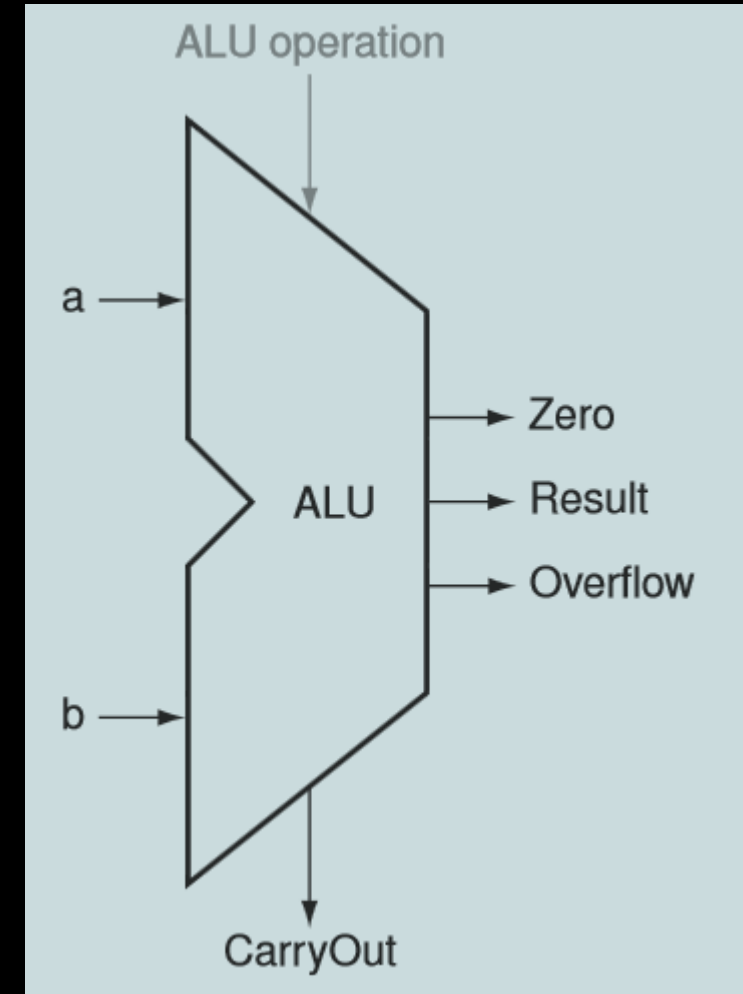
ARITHMETIC LOGIC UNIT

Now that we've implemented a 1-bit ALU, we can simply combine 32 1-bit ALUs to create a 32-bit ALU. Easy!
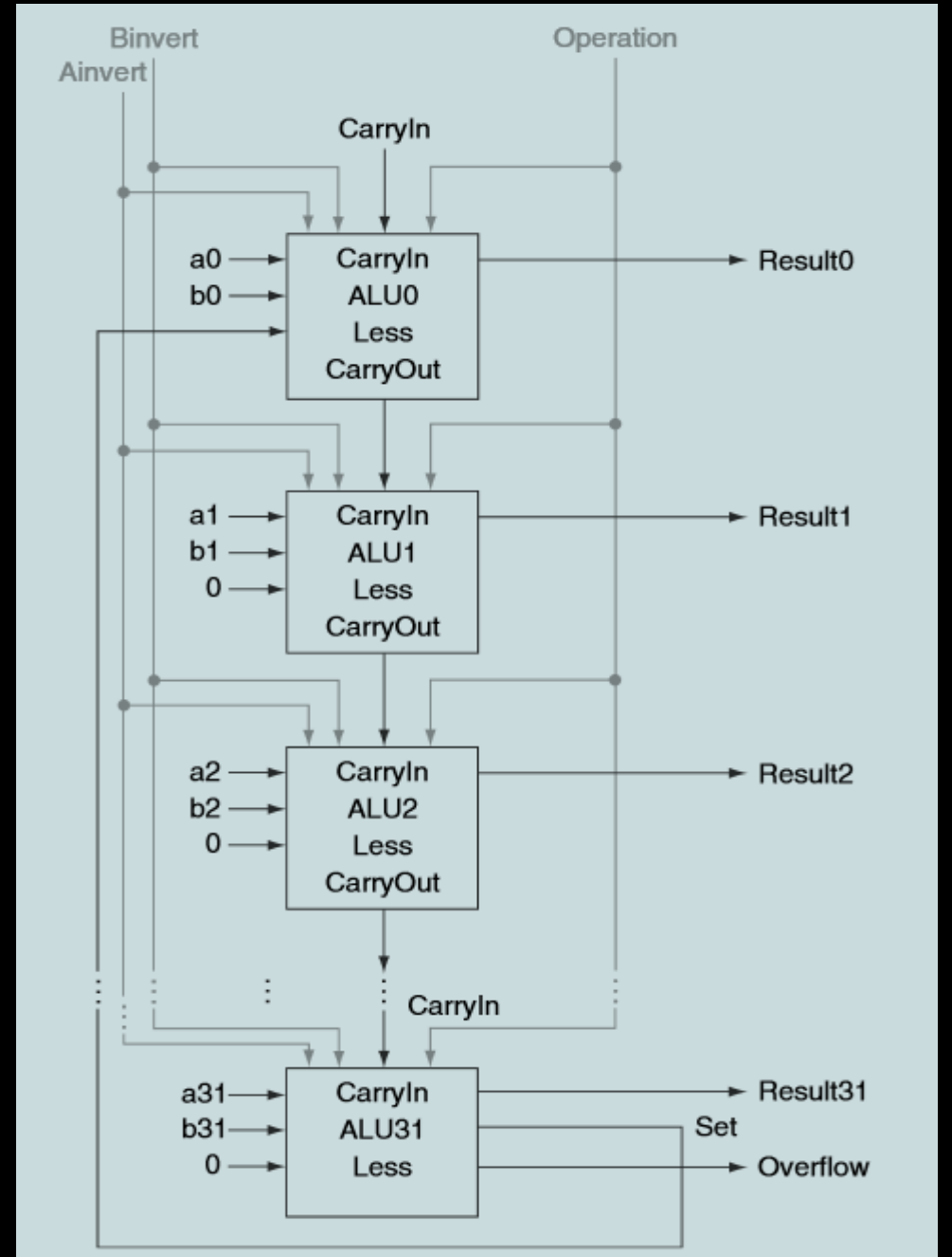
ARITHMETIC LOGIC UNIT

To hide the implementation details, we can use the universal ALU representation instead.

CLOCKS

Recall that sequential logic involves the idea of an internal state which affects the output of a logic block. In sequential logic, there is also the notion of a *clock,* which is used to decide when an element that contains a state should be updated.
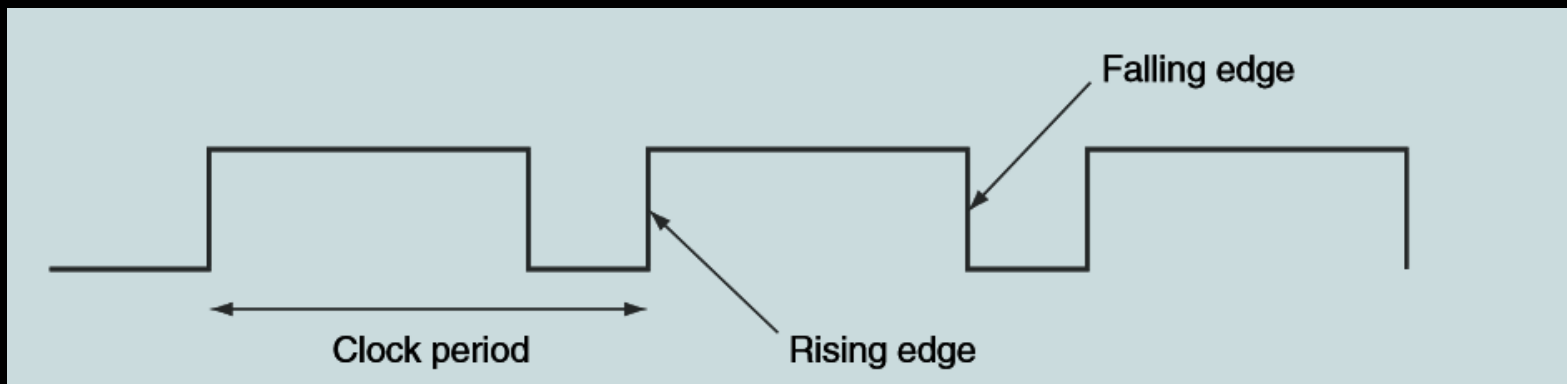
Clocks are simply a free-running signal with a fixed *cycle time* (clock period). The clock period is divided into two portions: high and low voltage.
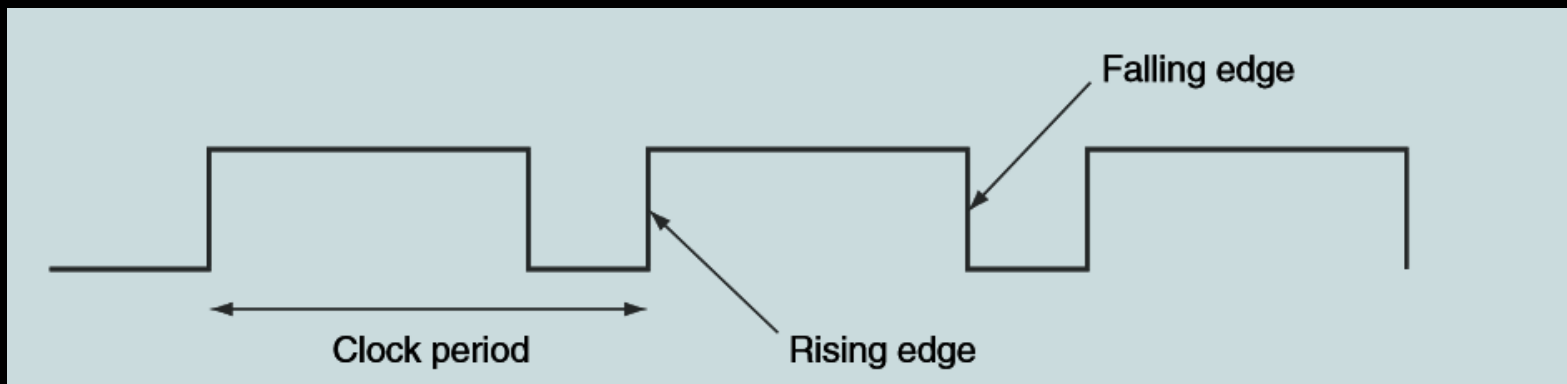
CLOCKS

Edge-triggered clocking refers to the scheme in which all state changes occur on some particular clock edge (either rising or falling).

There are, however, other *clocking methodologies* that may be implemented.

STATE ELEMENTS

State elements are memory elements with at least two inputs and one output.

The inputs are the data value to be written to the state element and the clock signal, which indicates when the data value should be written.

The output is the data value that was written on the previous cycle.

Some state elements are only written when there is an explicit write signal, which can only occur on the active clock edge.

SYNCHRONOUS SYSTEM

A clocked system is also known as a *synchronous system*.

Below is a diagram representing the relationship between state elements and logic blocks in synchronous, sequential logic design.
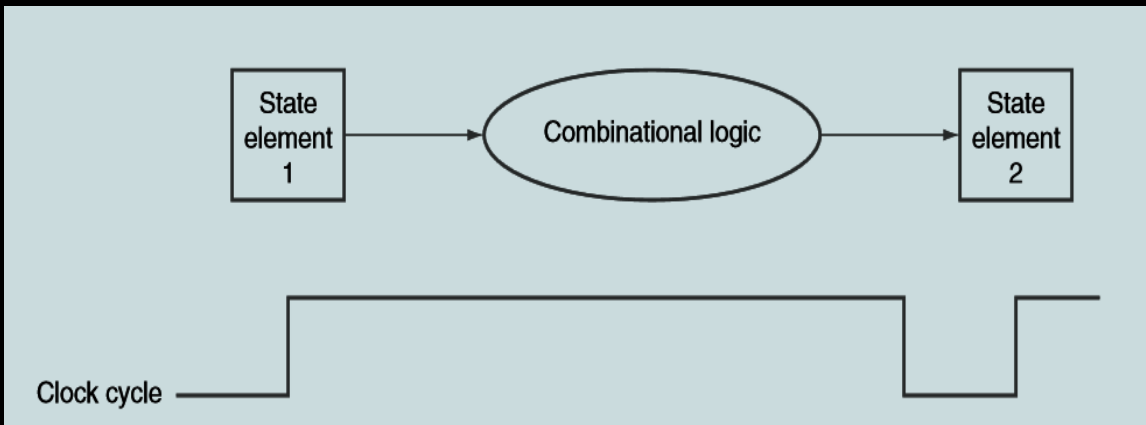
Why not just shorten the clock cycle time?

There is a lower bound on the length of the clock period, which must be long enough for all state input elements to be "valid" before they are written. A signal is considered "valid" if it is stable.

SYNCHRONOUS SYSTEMS

Edge-triggered methodology allows for state elements to be used as input as well as output. The previous diagram can be condensed into the following, which uses only one state element for the combinational logic block.