Licence
Maîtrise
d'Informatique

AMp : Computer Architecture basics

Robert Strandh

12 janvier 2000

Table des matières

Introduction

This document was obtained by translating the collection of web pages on computer architecture into latex and encapsuled postscript. The goal was to have a prtty printing of these page for the exams.

The first part is an introduction to digital circuits. We recommend you read the pages in this order :

1. Gates. Here, we introduce the fundamental building blocks of digital computers.
2. Truth tables. We discuss truth tables as a way both of describing an existing circuit and of specifying a circuit to be built.
3. Combinatorial circuits. We introduce circuits whose output values depend only on a combination of the input values.
4. Logic expressions. We explain how to describe circuits as algebraic formulae and how to manipulate those formulae with algebraic laws.
5. Multiplexer. A particular combinatorial circuit so commonly used that we discuss it separately.
6. Demultiplexer. A particular combinatorial circuit so commonly used that we discuss it separately.
7. Decoder. A particular combinatorial circuit so commonly used that we discuss it separately.
8. Binary arithmetic. In this section, we introduce the fundamentals of binary arithmetic and representation of numbers.
9. Circuits for binary arithmetic. We show different circuits for binary arithmetic and explain trade-offs between speed and number of gates.
10. Latches and flip-flops. Flip-flops are the basic elements of sequential circuits, the way gates are the basic elements of combinatorial circuits.
11. State table. We discuss state tables as a way both of describing an existing sequential circuit and as a way of specifying a sequential circuit to be built.
12. Sequential circuits. We introduce circuits whose output values depend not only on the inputs, but also on previous input and output values.
13. Registers. A register is a particularly simple sequential circuit that can be instructed to store its input values indefinitely.
14. Counters. A counter is a another particularly simple sequential circuit that normally increments its stored value for each clock pulse.
15. Binary multiplication. We show how to build a circuit for binary multiplication.

16. Tri-state logic circuits. Tri-state logic circuits represent a pragmatic solution to some problems of circuit complexity.
17. Buses. With tri-state logic, we can use a bus to transport data.
18. Memories. While memories are clearly sequential circuits, they have a special structure that makes it interesting to look at them separately.
19. Read-only memories. A read-only memory is nothing more than a combinatorial circuit, but often built as a memory.

Next, we design a number of specific combinatorial and sequential circuits, which will be the building blocks of a simple computer :

1. Counter with clear. This sequential circuit is nominally an ordinary counter, but an additional input allows us to clear the contents of the counter.
2. Counter register. This sequential circuit is a combination of a counter and a register.
3. Counter register with clear. This sequential circuit is a combination of a counter with clear and a register. Counter register with clear and explicit increment. This sequential circuit is similar to a counter register with clear, except that it does not increment unless explicitly instructed to.
4. Micro memory. This is a combinatorial circuit sometimes in the form of a PROM.
5. Instruction decoder. This is a combinatorial circuit sometimes in the form of a PROM.
6. Arithmetic and logic unit. This is a combinatorial circuit for doing arithmetic and logic operations. The operation can be selected by the value of inputs.
7. The first computer. We present our first simple computer, allowing us to make some very simple computations.

Now, we gradually complicate our simple computer :

1. Conditional jumps. So far, our computer could only do unconditional jumps. Now, we show how to introduce conditional jumps as well.
2. Interrupts. Our computer must know how to react to external events from timers and peripherals. The mechanism for interrupts makes that possible.

Première partie .

Introduction to Digital circuits

1. Gates

1.1. Introduction

The basic building blocks of a computer are called *logical gates* or just *gates*.

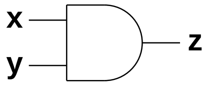
Gates are basic circuits that have at least one (and usually more) *input* and exactly one output. Input and output values are the logical values *true* and *false*. In computer architecture it is common to use `0` for false and `1` for true. Gates have no *memory*. The value of the output depends only on the current value of the inputs. This fact makes it possible to use a *truth table* to fully describe the behavior of a gate.

We usually consider three basic kinds of gates, *and*-gates, *or*-gates, and *not*-gates (or *inverters*).

1.2. Basic gates

1.2.1. The *and*-gate

An *and*-gate has an arbitrary number of inputs. The output value is `1` if and only if *all* of the inputs are `1`. Otherwise the output value is `0`. The name has been chosen because the output is `1` if and only if the first input *and* the second input, *and*, ..., *and* the nth input are all `1`.

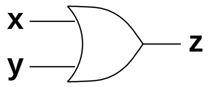
It is often useful to draw *diagrams* of gates and their interconnections. In such diagrams, the *and*-gate is drawn like this : 

The truth table for an *and*-gate with two inputs looks like this :

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

1.2.2. The *or*-gate

Like the *and*-gate, the *or*-gate can have an arbitrary number of inputs. The output value is `1` if and only of *at least one* of the input values are `1`. Otherwise the output is `0`. In other words, the output value is `0` only if all inputs are `0`. The name has been chosen because the output is `1` if and only if the first input *or* the second input, *or*, ..., *or* the nth input is `1`.

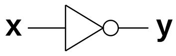
In circuit diagrams, we draw the *or*-gate like this : 

The truth table for an *or*-gate with two inputs looks like this :

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

1.2.3. The *inverter*

An *inverter* has exactly one input and one output. The value of the output is *1* if and only if the input is *0*. Otherwise, the output is *0*. In other words, the value of the output is the exact opposite of the value of the input.

In circuit diagrams, we draw the *inverter* like this : 

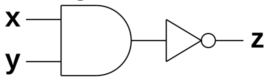
The truth table for an *inverter* looks like this :

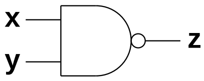
x	y
0	1
1	0

1.3. Combined gates

Sometimes, it is practical to combine functions of the basic gates into more complex gates, for instance in order to save space in circuit diagrams. In this section, we show some such combined gates together with their truth tables.

1.3.1. The *nand*-gate

The *nand*-gate is an *and*-gate with an inverter on the output. So instead of drawing several gates like this : 

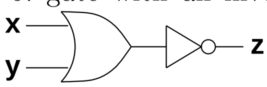
We draw a single *and*-gate with a little ring on the output like this : 

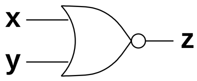
The *nand*-gate, like the *and*-gate can take an arbitrary number of inputs.

The truth table for the *nand*-gate is like the one for the *and*-gate, except that all output values have been inverted :

x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

1.3.2. The *nor*-gate

The *nor*-gate is an *or*-gate with an inverter on the output. So instead of drawing several gates like this : 

We draw a single *or*-gate with a little ring on the output like this : 

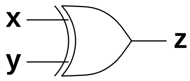
The *nor*-gate, like the *or*-gate can take an arbitrary number of inputs.

The truth table for the *nor*-gate is like the one for the *or*-gate, except that all output values have been inverted :

x	y	z
0	0	1
0	1	0
1	0	0
1	1	0

1.3.3. The *exclusive-or*-gate

The *exclusive-or*-gate is similar to an *or*-gate. It can have an arbitrary number of inputs, and its output value is `1` if and only if *exactly one input* is `1` (and thus the others `0`). Otherwise, the output is `0`.

We draw an *exclusive-or*-gate like this : 

The truth table for an *exclusive-or*-gate with two inputs looks like this :

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

1.4. So, how many different types of gates are there ?

A valid question at this point is how many different kinds of gates there are, and what they are called.

Let us limit ourselves to gates with n inputs. The truth tables for such gates have 2^n lines. Such a gate is completely defined by the output column in the truth table. The output column can be viewed as a string of 2^n binary digits. How many different strings of binary digits of length 2^n are there ? The answer is 2^{2^n}, since there are 2^k different strings of k binary digits, and if k=2^n, then there are 2^{2^n} such strings. In particular, if n=`2`, we can see that there are `16` different types of gates with `2` inputs.

Most of these gates do not have any names, and indeed, most of them are quite useless. For completeness, let us look at all `16` and study the functions they compute. Each entry in the following table is specified by the output string :

– `0000` A gate that ignores both its inputs and always gives `0` on the output. This gate does not require any circuits. Just let the inputs hang and connect the output to a `0`.
– `0001` This is the *and*-gate described above.
– `0010` This is like an *and*-gate with an *inverter* on the second input.
– `0011` This gate ignores its second input, and gives as output the value of its first input. It does not require any circuits. Just connect the output to the first input

and let the second input hang.

- 0100 This is like an *and*-gate with an *inverter* on the first input.
- 0101 This gate ignores its first input, and gives as output the value of its second input. It does not require any circuits. Just connect the output to the second input and let the first input hang.
- 0110 This is the *exclusive-or*-gate described above.
- 0111 This is the *or*-gate described above.
- 1000 This is the *nor*-gate described above.
- 1001 This is like an *exclusive-or*-gate with an inverter on its output.
- 1010 This gate can be built with an *inverter* on the second input, and with the first input hanging.
- 1011 This is like an *or*-gate with an inverter on its second input.
- 1100 This gate can be built with an *inverter* on the first input, and with the second input hanging.
- 1101 This is like an *or*-gate with an inverter on its first input.
- 1110 This is the *nand*-gate described above.
- 1111 This is a gate that ignores both its inputs and always gives a 1 as output. This gate does not require any circuits. Just let the inputs hang and connect the output to a 1.

As you can see, most of the gates possible, are quite useless.

1.5. Doing it all with only one kind of gate

As it turns out, it is possible to build any kind of gate using only *nand*-gates. To see this, first observe that an *inverter* is just a *nand*-gate with only one input. Second, an *and*-gate can be built as a *nand* with an inverter on its output. Last, an *or*-gate can be build with a *nand*-gate with an *inverter* on each input.

In some circuit technology, it is actually *easier* to build a *nand*-gate than any of the other gates. In that case, the *nand*-gate is considered the most primitive building block, and everything else is built from it.

Similarly, all gates can be realized from only *nor*-gates. Again an *inverter* is just a *nor*-gate with only one input. An *or*-gate is a *nor*-gate with an *inverter* on its output, and an *and*-gate is just a *nor*-gate with an *inverter* on each input.

1.6. Summary

Although gates can be arbitrarily complicated, from now on when we say *gates* we mean one of *inverter*, *and*, *or*, *nand*, *nor*, and sometimes *exclusive or*.

So why do we want to exclude other gates ? The main reason is that we want the number of gates of a circuit to reflect the cost of fabrication of that circuit. If we allow arbitrarily complicated gates, we can design an arbitrarily complicated combinatorial circuit with only one gate.

Our restriction above is not ideal, since in order for the number of gates to reflect the cost, we would have to take the particular technology used for fabrication into account.

Different circuits are easy to make in different technologies. But in order to remain somewhat independent of specific technology, we use this approximation.

2. Truth tables

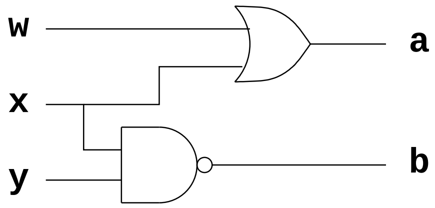
2.1. Introduction

To specify the exact way in which a combinatorial circuit works, we might use different methods, such as logical expressions, or as in this section, *truth tables*.

A truth table is a complete enumeration of all possible combinations of input values, each one with its associated output value.

2.2. Describing existing circuits

When used to describe an existing circuit, output values are (of course) either `0` or `1`. Suppose for instance that we wish to make a truth table for the following circuit :



All we need to do to establish a truth table for this circuit is to compute the output value for the circuit for each possible combination of input values. We obtain the following truth table :

w	x	y	a	b
0	0	0	0	1
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

2.3. Specifying circuits to build

When used as a *specification* for a circuit, a table may have some output values that are not specified, perhaps because the corresponding combination of input values can never occur in the particular application. We can indicate such unspecified output values with a dash `-`.

For instance, let us suppose we want a circuit of four inputs, interpreted as two nonnegative binary integers of two binary digits each, and two outputs, interpreted as the nonnegative binary integer giving the quotient between the two input numbers. Since division is not defined when the denominator is zero, we do not care what the output value is in this case. Of the sixteen entries in the truth table, four have a zero denominator. Here is the truth table :

x_1	x_0	y_1	y_0	z_1	z_0
0	0	0	0	-	-
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	-	-
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	-	-
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	-	-
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	1	0	1

Unspecified output values like this can greatly decrease the number of circuits necessary to build the circuit. The reason is simple : when we are free to choose the output value in a particular situation, we choose the one that gives the fewest total number of gates.

For the purpose of this course, we don't care about minimality. It might be conceptually advantageous still to indicate outputs with arbitrary values.

3. Combinatorial Circuits

3.1. Prerequisites

Read about gates first.

3.2. Introduction

A *combinatorial circuit* is a generalized gate. In general such a circuit has m inputs and n outputs.

Such a circuit can always be constructed as n separate combinatorial circuits, each with exactly one output. For that reason, some texts only discuss combinatorial circuits with exactly one output. In reality, however, some important *sharing of intermediate signals* may take place if the entire n-output circuit is constructed at once. Such sharing can significantly reduce the number of gates required to build the circuit.

When we build a combinatorial circuit from some kind of specification, we always try to make it *as good as possible*. The only problem is that the definition of "as good as possible" may vary greatly. In some applications, we simply want to minimize the number of gates (or the number of transistors, really). In other, we might be interested in as short a *delay* (the time it takes a signal to traverse the circuit) as possible, or in as low *power consumption* as possible. In general, a mixture of such criteria must be applied.

Circuit minimization is a difficult problem from complexity point of view. Computer programs that try to optimize circuit design apply a number of *heuristics* to improve speed. In this course, we are not concerned with optimality. We are therefore only going to discuss a simple method that works for all possible combinatorial circuits (but that can waste large numbers of gates).

A separate single-output circuit is built for each output of the combinatorial circuit.

Our simple method starts with the truth table (or rather *one of the acceptable truth tables*, in case we have a choice). Our circuit is going to be a two-layer circuit. The first layer of the circuit will have at most 2^n *nand*-gates, each with n inputs (where n is the number of inputs of the combinatorial circuit). The second layer will have a single *nand*-gate with as many inputs as there are gates in the first layer. For each line of the truth table with an output value of `1`, we put down a *nand*-gate with n inputs. For each input entry in the table with a `1` in it, we connect an input of the *nand*-gate to the corresponding input. For each entry in the table with a `0` in it, we connect an input of the *nand*-gate to the corresponding input *inverted*.

The output of each *nand*-gate of the fist layer is then connected to an input of the giant *nand*-gate of the second layer. And that's it.

As an example of our general method, consider the following truth table (where a `-` indicates that we don't care what value is chosen) :

x	y	z	a	b
0	0	0	-	0
0	0	1	1	1
0	1	0	1	-
0	1	1	0	0
1	0	0	0	1
1	0	1	0	-
1	1	0	-	-
1	1	1	1	0

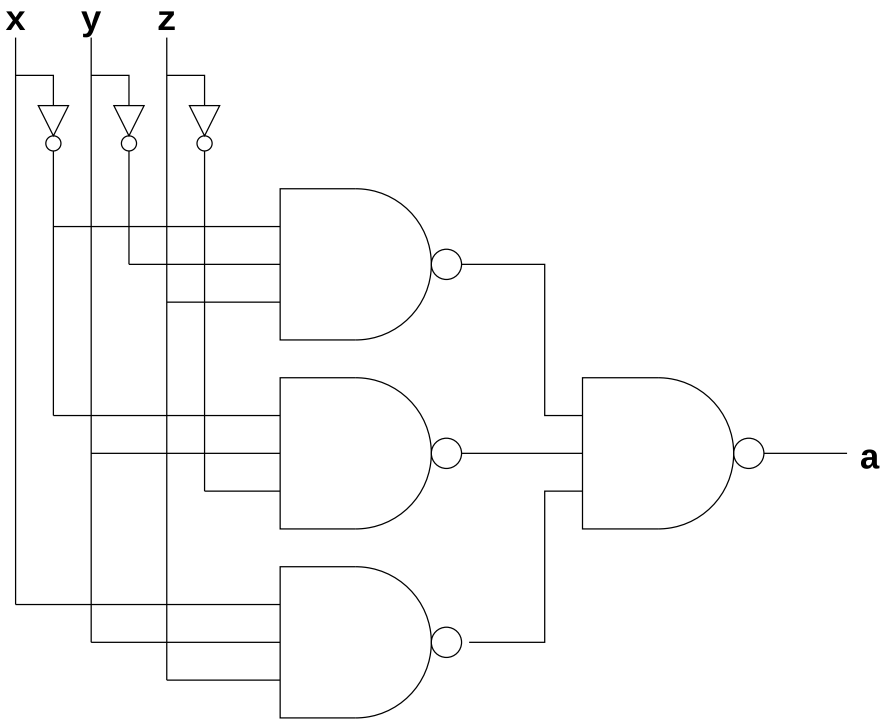
The first step is to arbitrarily choose values for the undefined outputs. With out simple method, the best solution is to choose a `0` for each such undefined output. We get this table :

x	y	z	a	b
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	0

Now, we have to build two separate single-output circuits, one for the `a` column and one for the `b` column.

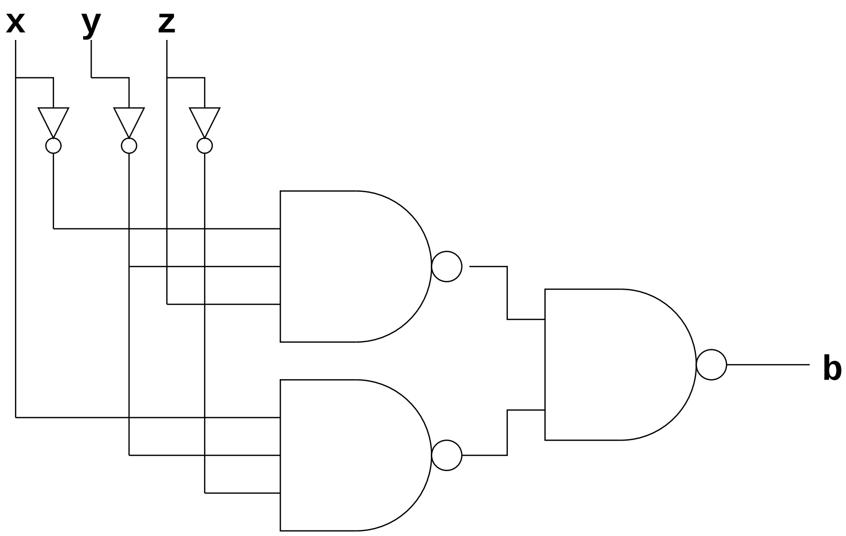
For the first column, we get three 3-input *nand*-gates in the first layer, and a 3-input *nand*-gate in the second layer. We get three *nand*-gates since there are three rows in the `a` column with a value of `1`. Each one has 3-inputs since there are three inputs, `x`, `y`, and `z` of the circuit. We get a 3-input *nand*-gate in the second layer since there are three *nand*-gates in the first layer.

Here is the complete circuit for the first column :

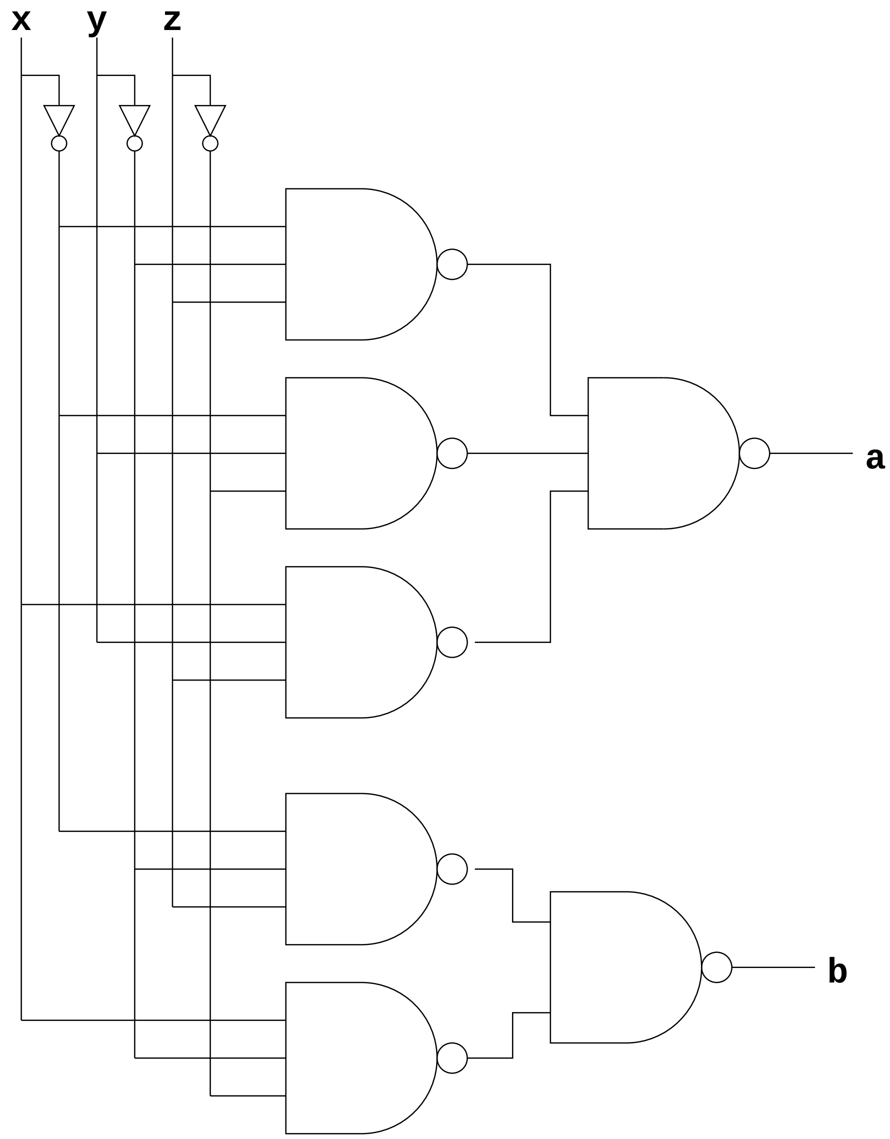


For the second column, we get two 3-input *nand*-gates in the first layer, and a 2-input *nand*-gate in the second layer. We get two *nand*-gates since there are two rows in the `b` column with a value of `1`. Each one has 3-inputs since again there are three inputs, `x`, `y`, and `z` of the circuit. We get a 2-input *nand*-gate in the second layer since there are two *nand*-gates in the first layer.

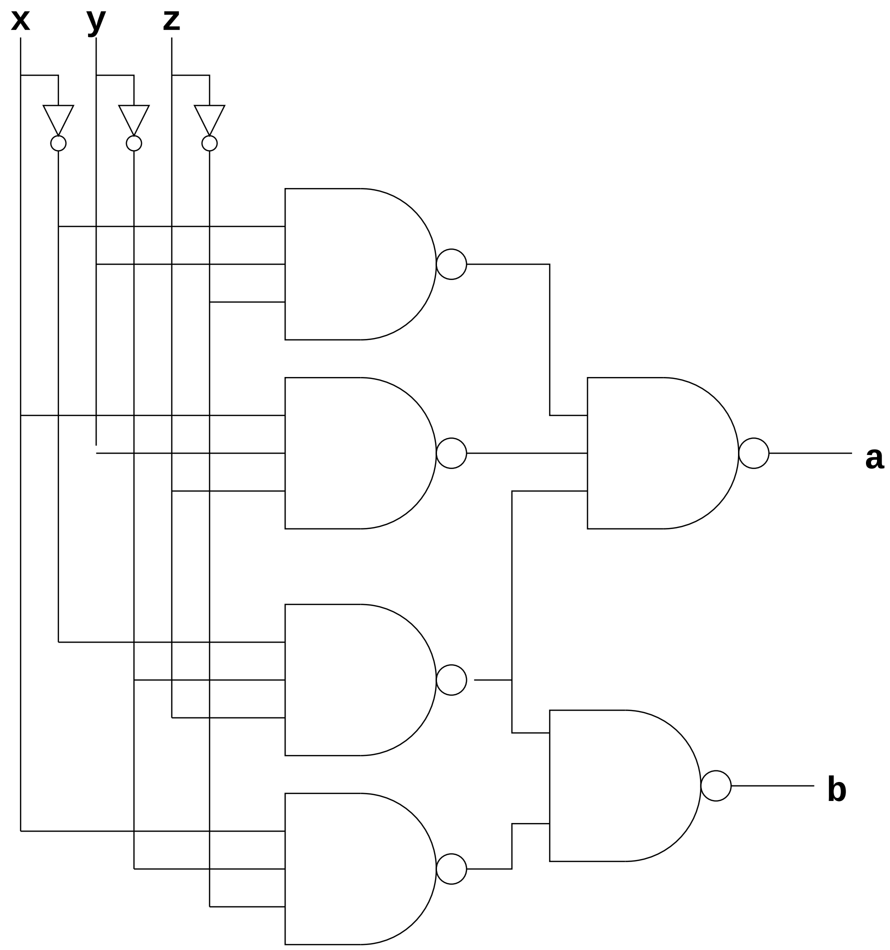
Here is the complete circuit for the second column :



Now, all we have to do is to combine the two circuits into a single one :



While this circuit works, it is not the one with the fewest number of gates. In fact, since both output columns have a `1` in the row correspoding to the inputs `0 0 1`, it is clear that the gate for that row can be shared between the two subcircuits :



In some cases, even smaller circuits can be obtained, if one is willing to accept more layers (and thus a higher circuit delay).

4. Logic expressions

4.1. Introduction

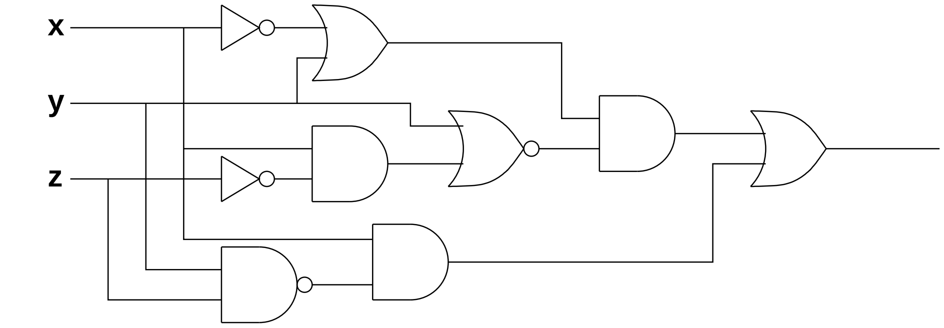
To define what a combinatorial circuit does, we can use a *logic expression* or an *expression* for short. Such an expression uses the two constants `0` and `1`, variables such as x, y, and z (sometimes with suffixes) as names of inputs and outputs, and the operators `+` (which stands for *or*), `*` (which stands for *and*, and is usually replaced as usual with juxtaposition), and a horizontal bar (which stands for *not*). As usual, multiplication is considered to have higher priority than addition. Parentheses are used to modify the priority.

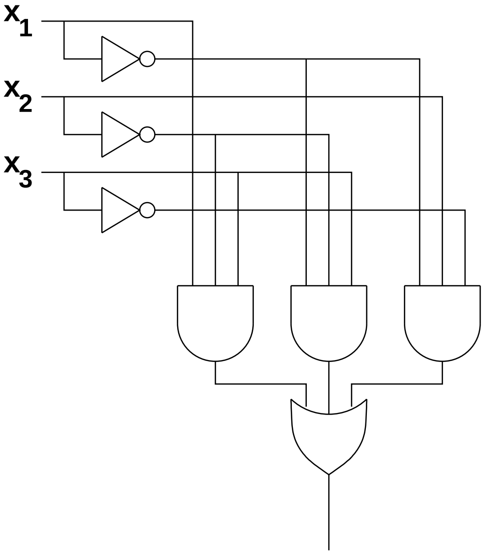
Examples :

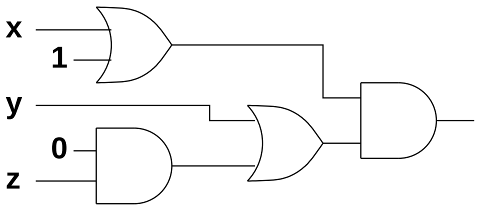
$$(\bar{x}+y)(\overline{\ y+x\bar{z}\ })+x\bar{y}\bar{z}$$
$$x_1\bar{x_2}x_3+\bar{x_1}\bar{x_2}x_3+\bar{x_1}x_2\bar{x_3}$$
$$(x+1)(y+0z)$$

4.2. Correspondence between expressions and combinatorial circuits

Each expression has a trivial corresponding combinatorial circuit. For instance, the expressions in the example above, correspond to the following circuits :







4.3. Power of logic expressions

A valid question is : *can logic expressions describe all possible combinatorial circuits ?*. The answer is *yes* and here is why :

You can trivially convert the truth table for an arbitrary circuit into an expression. The expression will be in the form of a sum of products of variables and there inverses. Each row with output value of `1` of the truth table corresponds to one term in the sum. In such a term, a variable having a `1` in the truth table will be uninverted, and a variable having a `0` in the truth table will be inverted.

Take the following truth table for example :

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The corresponding expression is : $\bar{x}\bar{y}z + x\bar{y}\bar{z} + xyz$

Since you can describe any combinatorial circuit with a truth table, and you can describe any truth table with an expression, you can describe any combinatorial circuit with an expression.

4.4. Simplicity of logic expressions

There are many logic expressions (and therefore many circuits) that correspond to a certain truth table, and therefore to a certain function computed. For instance, the following two expressions compute the same function :

$$x(y + z) \quad xy + xz$$

The left one requires two gates, one *and*-gate and one *or*-gate. The second expression requires two *and*-gates and one *or*-gate. It seems obvious that the first one is preferable to the second one. However, this is not always the case. It is not always true that the number of gates is the only way, nor even the best way, to determine simplicity.

We have, for instance, assumed that gates are ideal. In reality, the signal takes some time to propagate through a gate. We call this time the gate *delay*. We might be interested in circuits that minimize the total gate delay, in other words, circuits that make the signal traverse the fewest possible gates from input to output. Such circuits are not necessarily the same ones that require the smallest number of gates.

5. Multiplexer

A multiplexer is a combinatorial circuit that is given a certain number (usually a power of two) *data inputs*, let us say 2^n, and `n` *address inputs* used as a binary number to select one of the data inputs. The multiplexer has a single output, which has the same value as the selected data input.

In other words, the multiplexer works like the input selector of a home music system. Only one input is selected at a time, and the selected input is transmitted to the single output. While on the music system, the selection of the input is made manually, the multiplexer chooses its input based on a binary number, the address input.

The truth table for a multiplexer is huge for all but the smallest values of `n`. We therefore use an abbreviated version of the truth table in which some inputs are replaced by '`-`' to indicate that the input value does not matter.

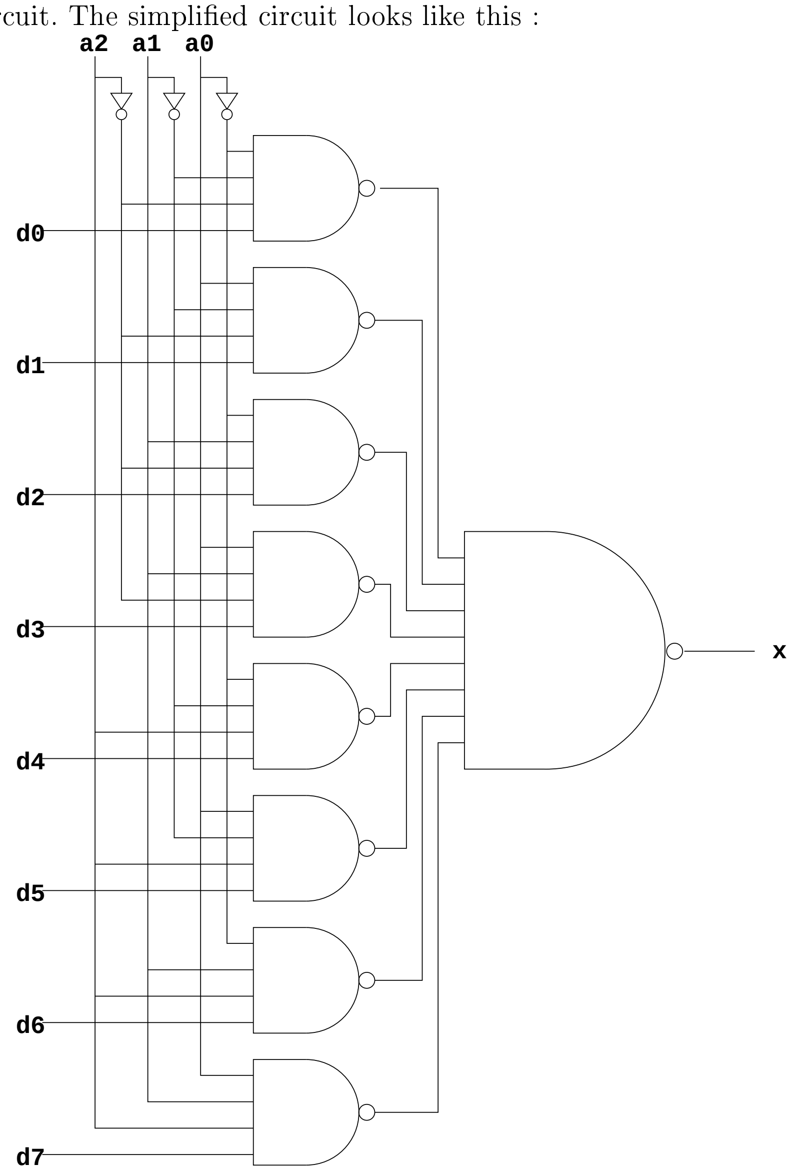
Here is such an abbreviated truth table for `n = 3`. The full truth table would have $2^{(3+2^3)}$ `= 2048` rows.

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	x
0	0	0	-	-	-	-	-	-	-	0	0
0	0	0	-	-	-	-	-	-	-	1	1
0	0	1	-	-	-	-	-	-	0	-	0
0	0	1	-	-	-	-	-	-	1	-	1
0	1	0	-	-	-	-	-	0	-	-	0
0	1	0	-	-	-	-	-	1	-	-	1
0	1	1	-	-	-	-	0	-	-	-	0
0	1	1	-	-	-	-	1	-	-	-	1
1	0	0	-	-	-	0	-	-	-	-	0
1	0	0	-	-	-	1	-	-	-	-	1
1	0	1	-	-	0	-	-	-	-	-	0
1	0	1	-	-	1	-	-	-	-	-	1
1	1	0	-	0	-	-	-	-	-	-	0
1	1	0	-	1	-	-	-	-	-	-	1
1	1	1	0	-	-	-	-	-	-	-	0
1	1	1	1	-	-	-	-	-	-	-	1

We can abbreviate this table even more by using a letter to indicate the value of the selected input, like this :

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	x
0	0	0	-	-	-	-	-	-	-	c	c
0	0	1	-	-	-	-	-	-	c	-	c
0	1	0	-	-	-	-	-	c	-	-	c
0	1	1	-	-	-	-	c	-	-	-	c
1	0	0	-	-	-	c	-	-	-	-	c
1	0	1	-	-	c	-	-	-	-	-	c
1	1	0	-	c	-	-	-	-	-	-	c
1	1	1	c	-	-	-	-	-	-	-	c

The same way we can simplify the truth table for the multiplexer, we can also simplify the corresponding circuit. Indeed, our simple design method would yield a very large circuit. The simplified circuit looks like this :



6. Demultiplexer

6.1. Pre-requisites

Read about the multiplexer first.

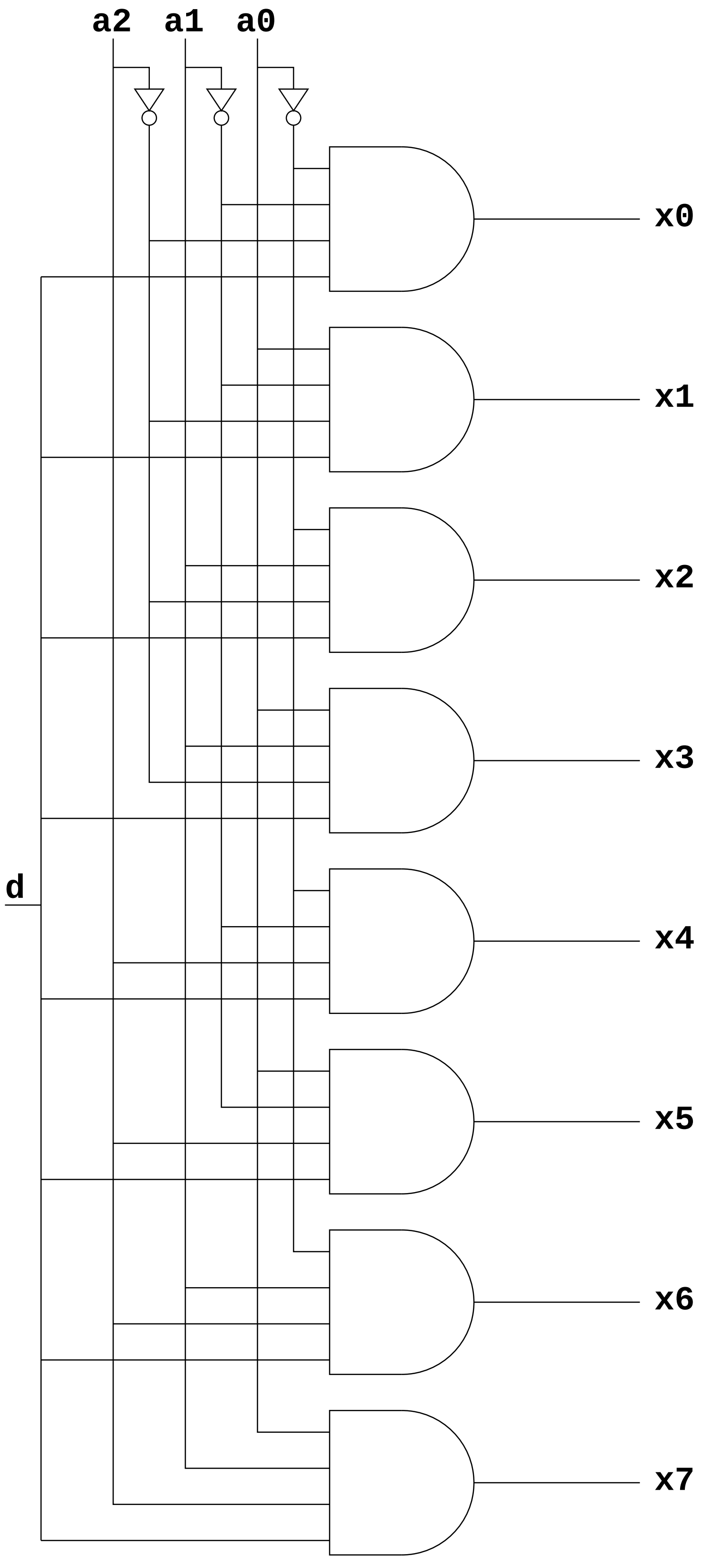
6.2. Demultiplexer

The demultiplexer is the inverse of the multiplexer, in that it takes a single data input and `n` address inputs. It has 2^n outputs. The address input determine which data output is going to have the same value as the data input. The other data outputs will have the value `0`.

Here is an abbreviated truth table for the demultiplexer. We could have given the full table since it has only `16` rows, but we will use the same convention as for the multiplexer where we abbreviated the values of the data inputs.

a_2	a_1	a_0	d	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	c	0	0	0	0	0	0	0	c
0	0	1	c	0	0	0	0	0	0	c	0
0	1	0	c	0	0	0	0	0	c	0	0
0	1	1	c	0	0	0	0	c	0	0	0
1	0	0	c	0	0	0	c	0	0	0	0
1	0	1	c	0	0	c	0	0	0	0	0
1	1	0	c	0	c	0	0	0	0	0	0
1	1	1	c	c	0	0	0	0	0	0	0

Here is one possible circuit diagram for the demultiplexer :


a2
a1
a0
d
x0
x1
x2
x3
x4
x5
x6
x7

7. Decoder

7.1. Pre-requisites

Read about the multiplexer and the demultiplexer first.

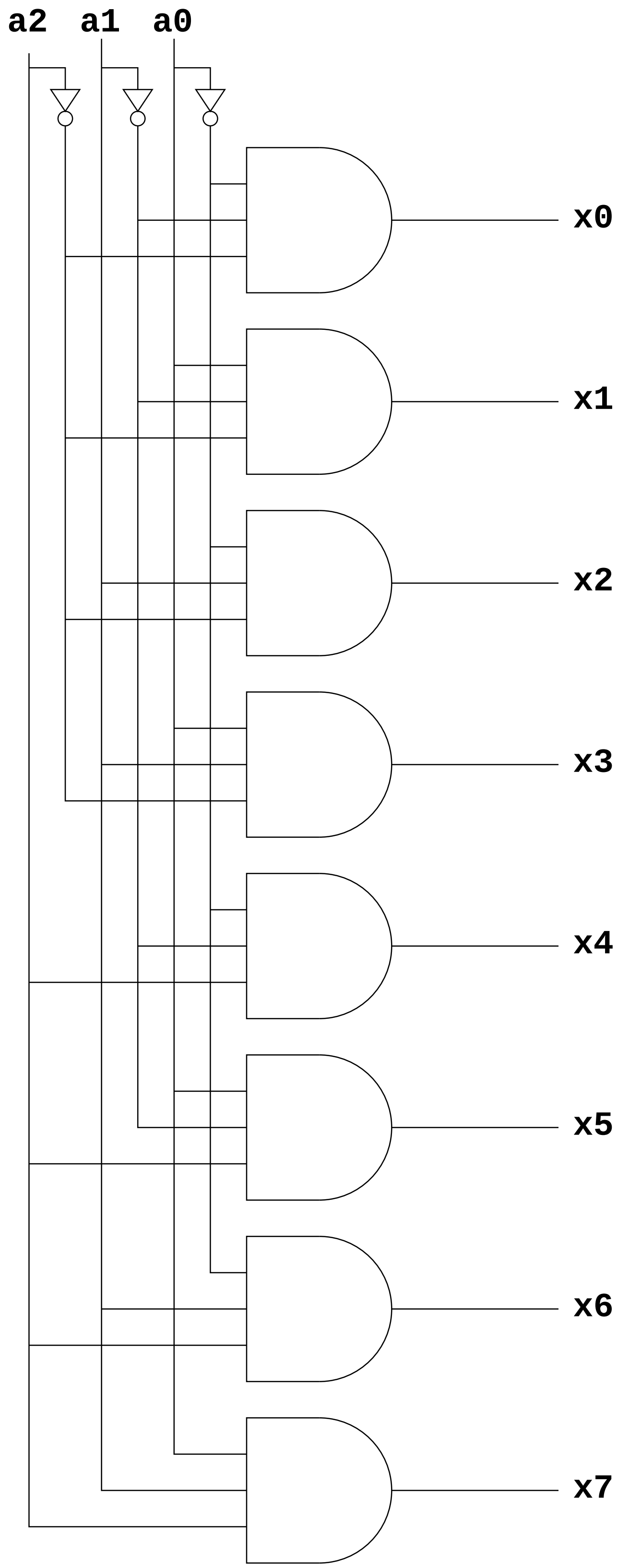
7.2. Decoder

In both the multiplexer and the demultiplexer, part of the circuits *decode* the address inputs, i.e. it translates a binary number of `n` digits to 2^n outputs, one of which (the one that corresponds to the value of the binary number) is `1` and the others of which are `0`.

It is sometimes advantageous to separate this function from the rest of the circuit, since it is useful in many other applications. Thus, we obtain a new combinatorial circuit that we call the *decoder*. It has the following truth table (for `n = 3`) :

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
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

Here is the circuit diagram for the decoder :


a2 a1 a0
x0
x1
x2
x3
x4
x5
x6
x7

8. Binary arithmetic

8.1. Introduction

An important part of the use of logic circuits is for computing various mathematical operations such as addition, multiplication, trigonometric operations, etc.

We must therefore have a way of representing numbers as binary data.

8.2. Nonnegative integers

The easiest numbers to represent are the nonnegative integers. To see how this can be done, recall how we represent number in the decimal system. A number such as `2034` is interpreted as :

```
2*103 + 0*102 + 3*101 + 4*100
```

But there is nothing special with the base `10`, so we can just as well use base `2`. In base `2`, each digit value is either `0` or `1`, which we can represent for instance by `false` and `true`, respectively.

In fact, we have already hinted at this possibility, since we usually write `0`, and `1` instead of `false` and `true`.

All the normal algorithms for decimal arithmetic have versions for binary arithmetic, except that they are usually simpler.

For adding two numbers, it suffices to notice that there is a carry of `1` whenever we add `0` and `1`, or `1` and `1` :

$$
\begin{array}{cccccc}
 & 1 & & 1 & 1 & \\
 & - & & - & - & \\
 & & 1 & 0 & 1 & 1 \\
+ & & 1 & 0 & 0 & 1 \\
\hline
 & 1 & 0 & 1 & 0 & 0
\end{array}
$$

Subtraction is no harder :

$$
\begin{array}{ccccc}
 & & 10 & 10 & \\
 & & - & - & \\
 & 1 & 0 & 0 & 1 \\
- & & 1 & 1 & 0 \\
\hline
 & 0 & 0 & 1 & 1
\end{array}
$$

For multiplying two number, the algorithm is simpler, since we only multiply by `0` (which gives all zeros), or `1` (which gives the original number) :

$$
\begin{array}{ccccccc}
 & & & 1 & 1 & 0 & 1 \\
 & * & & & 1 & 0 & 1 \\
\hline
 & & 1 & 1 & 0 & 1 & \\
 & & 0 & 0 & 0 & 0 & \\
 & 1 & 1 & 0 & 1 & & \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

Finally, division is just repeated subtraction as in decimal arithmetic :

$$
\begin{array}{ccc|cccc}
 & & & 0 & 1 & 1 & 0 \\
\cline{4-7}
1 & 0 & & 1 & 1 & 0 & 1 \\
\cline{1-2}
 & & & 1 & 0 & & \\
\cline{4-5}
 & & & & 1 & 0 & \\
 & & & & 1 & 0 & \\
\cline{5-6}
 & & & & & 0 & 1
\end{array}
$$

8.3. Negative integers

Things are easy as long as we stick to nonnegative integers. They become more complicated when we want to represent negative integers as well.

It may seem that we can do just what we do in decimal representation, i.e., indicate with a special sign whether the number is negative or not. In binary arithmetic, we could simply reserve one bit to determine the sign. In the circuitry for addition, we would have one circuit for adding two number, and another for subtracting two numbers. The combination of signs of the two inputs would determine which circuit to use on the absolute values, as well as the sign of the output.

While this method works, it turns out that there is one that is much easier to deal with by electronic circuits. This method is called the "two's complement" method. It turns out that with this method, we do not need a special circuit for subtracting two numbers.

In order to explain this method, we first show how it would work in decimal arithmetic with infinite precision. Then we show how it works with binary arithmetic, and finally how it works with finite precision.

8.3.1. Infinite-precision ten's complement

Imagine the odometer of an automobile. It has a certain number of wheels, each with the ten digits on it. When one wheel goes from `9` to `0`, the wheel immediately to the left of it, advances by one position. If *that* wheel already showed `9`, it too goes to `0` and advances the wheel to *its* left, etc. Suppose we run the car backwards. Then the reverse happens, i.e. when a wheel goes from `0` to `9`, the wheel to its left decreases by one.

Now suppose we have an odometer with an *infinite number of wheels*. We are going to use this infinite odometer to represent all the integers.

When all the wheels are `0`, we interpret the value as the integer `0`.

A positive integer n is represented by an odometer position obtained by advancing the rightmost wheel n positions from `0`. Notice that for each such positive number, there will be an infinite number of wheels with the value `0` to the left.

A negative integer n is represented by an odometer position obtained by decreasing the rightmost wheel n positions from `0`. Notice that for each such positive number, there will be an infinite number of wheels with the value `9` to the left.

In fact, we don't need an *infinite* number of wheels. For each number only a finite number of wheels is needed. We simply assume that the leftmost wheel (which will be either `0` or `9`) is duplicated an infinite number of times to the left.

While for each number we only need a finite number of wheels, the number of wheels is *unbounded*, i.e., we cannot use a particular finite number of wheels to represent *all* the numbers. The difference is subtle but important (but perhaps not that important for this particular course). If we need an infinite number of wheels, then there is no hope of ever using this representation in a program, since that would require an infinite-size memory. If we only need an unbounded number of wheels, we may run out of memory, but we can represent a lot of numbers (each of finite size) in a useful way. Since any program that runs in finite time only uses a finite number of numbers, with a large enough memory, we might be able to run our program.

Now suppose we have an addition circuit that can handle nonzero integers with an infinite number of digits. In other words, when given a number starting with an infinite number of `9`s, it will interpret this as an infinitely large positive number, whereas our interpretation of it will be a negative number. Let us say, we give this circuit the two numbers `...9998` (which we interpret as `-2`) and `...0005` (which we interpret as `+5`). It will add the two numbers. First it adds `8` and `5` which gives `3` and a carry of `1`. Next, it adds `9` and the carry `1`, giving `0` and a carry of `1`. For all remaining (infinitely many) positions, the value will be `0` with a carry of `1`, so the final result is `...0003`. This result is the correct one, even with our interpretation of negative numbers. You may argue that the carry must end up somewhere, and it does, but in infinity. In some ways, we are doing arithmetic modulo infinity.

Some implementations of some programming languages with arbitrary precision integer arithmetic (*Lisp* for instance) use exactly this representation of negative integers.

Let us finish this section by giving a simple method for computing the absolute value of a negative integer in our representation. It suffices to take each individual digit, replace it by `9` minus its original value, and then at the end, add `1` to the number obtained. So for instance, the number `...9998` becomes `1` plus `...0001` which is `...0002`. This method works both ways, i.e. you can also use it to negate a positive number.

8.3.2. Finite-precision ten's complement

What we have said in the previous section works almost as well with a fixed bounded number of odometer wheels. The only problem is that we have to deal with *overflow* and *underflow*.

Suppose we have only a fixed number of wheels, say `3`. In this case, we shall use the convention that if the leftmost wheel shows a digit between `0` and `4` inclusive, then we have a positive number, equal to its representation. When instead the leftmost wheel shows a digit between `5` and `9` inclusive, we have a negative number, whose absolute value can be computed with the method that we have in the previous section.

We now assume that we have a circuit that can add *positive* three-digit numbers, and we shall see how we can use it to add negative numbers in our representation.

Suppose again we want to add `-2` and `+5`. The representations for these numbers with three wheels are `998` and `005` respectively. Our addition circuit will attempt to add the two positive numbers `998` and `005`, which gives `1003`. But since the addition circuit only has three digits, it will truncate the result to `003`, which is the right answer for our interpretation.

A valid question at this point is in which situation our finite addition circuit will not work. The answer is somewhat complicated. It is clear that it always gives the correct result when a positive and a negative number are added. It is incorrect in two situations. The first situation is when two positive numbers are added, and the result comes out looking like a negative number, i.e, with a first digit somewhere between `5` and `9`. You should convince yourself that no addition of two positive numbers can yield an overflow and still look like a positive number. The second situation is when two negative numbers are added and the result comes out looking like a nonnegative number, i.e, with a first digit somewhere between `0` and `4`. Again, you should convince yourself that no addition of two negative numbers can yield an underflow and still look like a negative number.

We now have a circuit for addition of integers (positive or negative) in our representation. We simply use a circuit for addition of only positive numbers, plus some circuits that check :

– If both numbers are positive and the result is negative, then report overflow.
– If both numbers are negative and the result is positive, then report underflow.

8.3.3. Finite-precision two's complement

So far, we have studied the representation of negative numbers using ten's complement. In a computer, we prefer using base two rather than base ten. Luckily, the exact method described in the previous section works just as well for base two. For an n-bit adder (n is usually 32 or 64), we can represent positive numbers with a leftmost digit of `0`, which gives values between `0` and $2^{(n-1)} - 1$, and negative numbers with a leftmost digit of `1`, which gives values between $-2^{(n-1)}$ and `-1`.

The exact same rule for overflow and underflow detection works. If, when adding two positive numbers, we get a result that looks negative (i.e. with its leftmost bit `1`), then we have an overflow. Similarly, if, when adding two negative numbers, we get a result that looks positive (i.e. with its leftmost bit `0`), then we have an underflow.

8.4. Rational numbers

Integers are useful, but sometimes we need to compute with numbers that are not integer.

An obvious idea is to use *rational* numbers. Many algorithms, such as the simplex algorithm for linear optimization, use only rational arithmetic whenever the input is rational.

There is no particular difficulty in representing rational numbers in a computer. It suffices to have a pair of integers, one for the numerator and one for the denominator.

To implement arithmetic on rational numbers, we can use some additional restrictions on our representation. We may, for instance, decide that :

- positive rational numbers are always represented as two positive integers (the other possibility is as two negative numbers),
- negative rational numbers are always represented with a negative numerator and a positive denominator (the other possibility is with a positive numerator and a negative denominator),
- the numerator and the denominator are always relative prime (they have no common factors).

Such a set of rules makes sure that our representation is *canonical*, i.e., that the representation for a value is unique, even though, a priori, many representations would work.

Circuits for implementing rational arithmetic would have to take such rules into account. In particular, the last rule would imply dividing the two integers resulting from every arithmetic operation with their largest common factor to obtain the canonical representation.

Rational numbers and rational arithmetic is not very common in the hardware of a computer. The reason is probably that rational numbers don't behave very well with respect to the size of the representation. For rational numbers to be truly useful, their components, i.e., the numerator and the denominator, both need to be arbitrary-precision integers. As we have mentioned before, arbitrary precision anything does not go very well with fixed-size circuits inside the CPU of a computer.

Programming languages, on the other hand, sometimes use arbitrary-precision rational numbers. This is the case, in particular, with the language *Lisp*.

8.5. Floating-point numbers

Instead of using the obvious representation of rational numbers presented in the previous section, most computers use a different representation of a subset of the rational numbers. We call these numbers *floating-point* numbers.

Floating-point numbers use inexact arithmetic, and in return require only a fixed-size representation. For many computations (so-called scientific computations, as if other computations weren't scientific) such a representation has the great advantage that it is fast, while at the same time usually giving adequate precision.

There are some (sometimes spectacular) exceptions to the "adequate precision" statement in the previous paragraph, though. As a result, an entire discipline of applied mathematics, called numerical analysis, has been created for the purpose of analyzing how algorithms behave with respect to maintaining adequate precision, and of inventing new algorithms with better properties in this respect.

The basic idea behind floating-point numbers is to represent a number as *mantissa* and an *exponent*, each with a fixed number of bits of precision. If we denote the mantissa with m and the exponent with e, then the number thus represented is $m * 2^e$.

Again, we have a problem that a number can have several representations. To obtain a canonical form, we simply add a rule that m must be greater than or equal to `1/2` and strictly less than `1`. If we write such a mantissa in binal (analogous to *decimal*) form, we always get a number that starts with `0.1`. This initial information therefore does not have to be represented, and we represent only the remaining "binals".

The reason floating-point representations work well for so-called scientific applications, is that we more often need to *multiply* or *divide* two numbers. Multiplication of two floating-point numbers is easy to obtain. It suffices to multiply the mantissas and add the exponents. The resulting mantissa might be smaller than `1/2`, in fact, it can be as small as `1/4`. In this case, the result needs to be canonicalized. We do this by shifting the mantissa left by one position and subtracting one from the exponent. Division is only slightly more complicated. Notice that the imprecision in the result of a multiplication or a division is only due to the imprecision in the original operands. No additional imprecision is introduced by the operation itself (except possibly 1 unit in the least significant digit). Floating-point addition and subtraction do not have this property.

To add two floating-point numbers, the one with the smallest exponent must first have its mantissa shifted right by n steps, where n is the difference of the exponents. If n is greater than the number of bits in the representation of the mantissa, the second number will be treated as `0` as far as addition is concerned. The situation is even worse for subtraction (or addition of one positive and one negative number). If the numbers have roughly the same absolute value, the result of the operation is roughly zero, and the resulting representation may have no correct significant digits.

The two's complement representation that we mentioned above is mostly useful for addition and subtraction. It only complicates things for multiplication and division. For multiplication and division, it is better to use a representation with sign + absolute value. Since multiplication and division is more common with floating-point numbers, and since they result in multiplication and division of the mantissa, it is more advantageous to have the mantissa represented as sign + absolute value. The exponents are added, so it is more common to use two's complement (or some related representation) for the exponent.

Usually, computers manipulate data in chunks of `8`, `16`, `32`, `64`, or `128` bits. It is therefore useful to fit a single floating-point number with both mantissa and exponent in such a chunk. In such a chunk, we need to have room for the sign (1 bit), the mantissa, and the exponent. While there are many different ways of dividing the remaining bits between the mantissa and the exponent, in practice most computers now use a norm called IEEE-???, which mandates the following formats :???

8.6. Real numbers

One sometimes hears variations on the phrase "computers can't represent real numbers exactly". This, of course is not true. Nothing prevents us from representing (say) the square-root of two as the number two and a bit indicating that the value is the square root of the representation. Some useful operations could be very fast this way. It is true, though that we cannot represent *all* real numbers exactly. In fact, we have a similar problem that we have with rational numbers, in that it is hard to pick a useful subset that we *can* represent exactly, other than the floating-point numbers.

For this reason, no widespread hardware contains built-in real numbers other than the usual approximations in the form of floating-point.

9. Circuits for binary arithmetic

9.1. Introduction

Binary arithmetic is a combinatorial problem. It may seem trivial to use the methods we have already seen for designing combinatorial circuits to obtain circuits for binary arithmetic.

However, there is a problem. It turns out that the normal way of creating such circuits would often use up way too many gates. We must search for different ways.

9.2. Binary integer addition

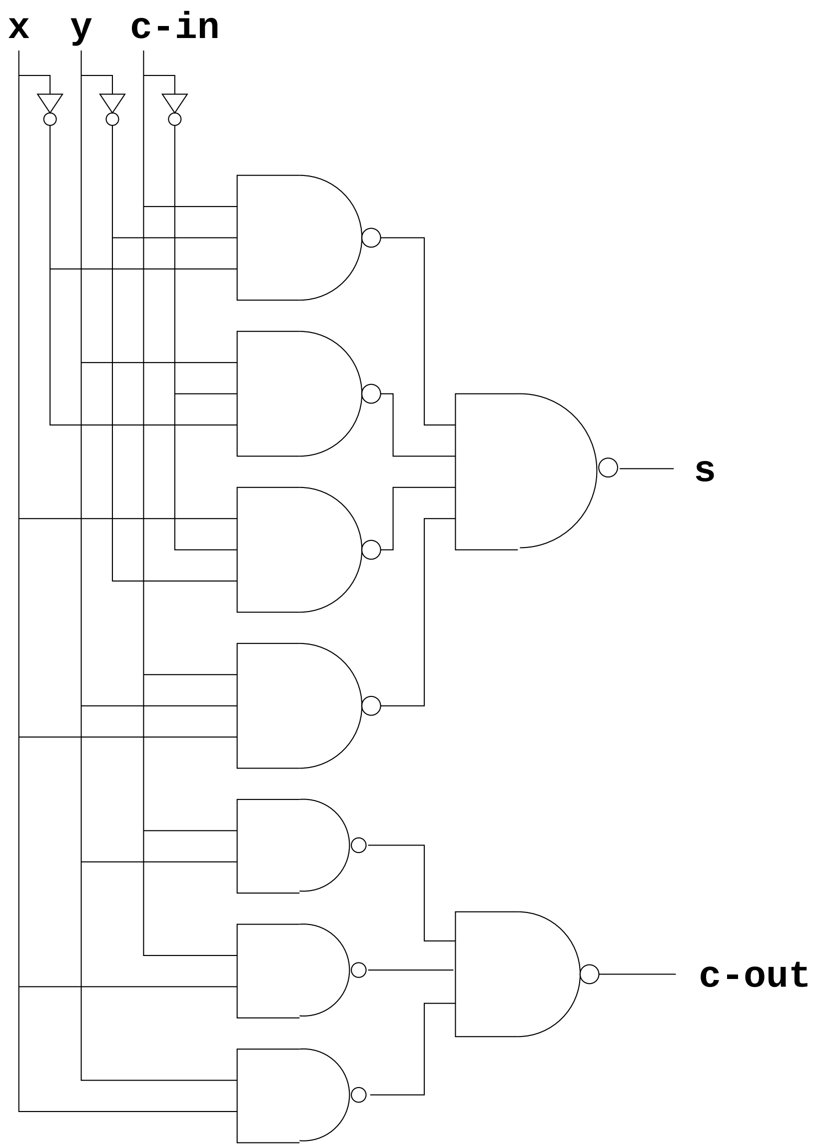
For binary integer addition, we can sacrifice our requirement on *circuit depth* that we previously had, in order to use fewer gates. The resulting circuit is of a type that we call *iterative combinatorial circuit*, in that it contains several copies of a simple element. For binary addition, that simple element is called a *full adder*.

A full adder is a combinatorial circuit (or actually two combinatorial circuits) of three inputs and two outputs. Its function is to add two binary digits plus a carry from the previous position, and give a two-bit result, the normal output and the carry to the next position. Here is the truth table for a full adder :

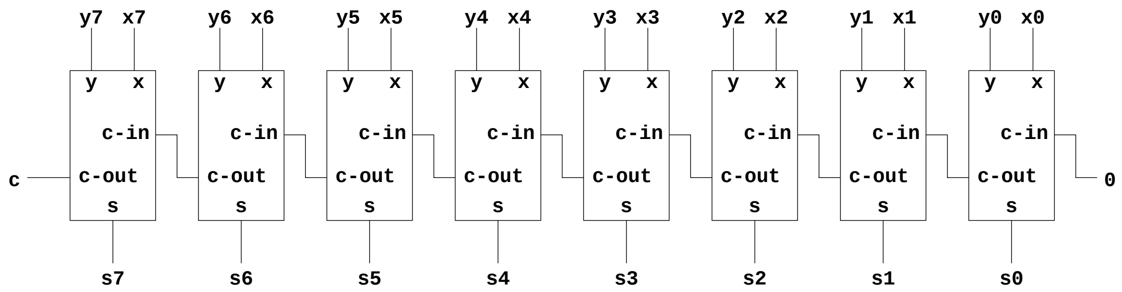
x	y	c-in	c-out	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Here, we have used variable names x and y for the inputs, *c-in* for the carry-in, s for the sum output and *c-out* for the carry-out.

A full adder can be trivially built using our ordinary design methods for combinatorial circuits. Here is the resulting circuit diagram :



The next step is to combine a series of such full adders into a circuit that can add (say) two 8-bit positive numbers. We do this by connecting the carry-out from one full adder to the carry-in of the full adder immediately to its left. The rightmost full adder takes a `0` on its carry-in.



Here, we have used subscript i for the i-th binary position.

As you can see, the depth of this circuit is no longer two, but considerably larger. In fact, the output and carry from position `7` is determined in part by the inputs of position `0`. The signal must traverse all the full adders, with a corresponding delay as a result.

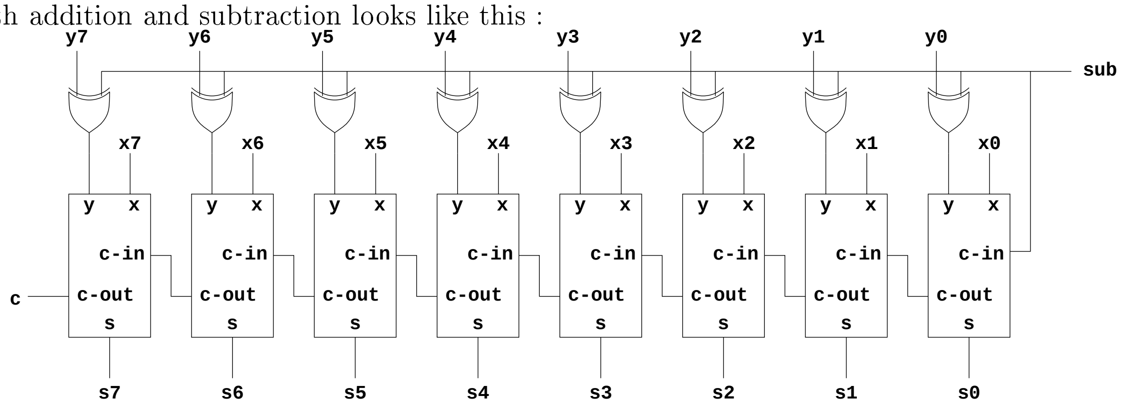
There are intermediate solutions between the two extreme ones we have seen so far (i.e. a combinatorial circuit for the entire (say) 32-bit adder, and an iterative combinatorial circuit whose elements are one-bit adders built as ordinary combinatorial circuits). We can for instance build an 8-bit adder as an ordinary two-level combinatorial circuit and build a 32-bit adder from four such 8-bit adders. An 8-bit adder can trivially be build from `65536` (2^{16}) *and*-gates, and a giant 65536-input *or*-gate.

Another intermediate solution consists of building so-called *carry-accelerator* circuits. [To be completed...]

9.3. Binary subtraction

Our binary adder can already handle negative numbers as indicated in the section on binary arithmetic. But we have not discussed how we can get it to handle subtraction.

To see how this can be done, notice that in order to compute the expression x - y, we can compute the expression x + -y instead. We know from the section on binary arithmetic how to negate a number by inverting all the bits and adding `1`. Thus, we can compute the expression as x + *inv(y)* + *1*. It suffices to invert all the inputs of the second operand before they reach the adder, but how do we add the `1`. That seems to require another adder just for that. Luckily, we have an unused carry-in signal to position `0` that we can use. Giving a `1` on this input in effect adds one to the result. The complete circuit with addition and subtraction looks like this :



9.4. Binary multiplication and division

Binary multiplication is even harder than binary addition. There is no good iterative combinatorial circuit available, so we have to use even heavier artillery. The solution is going to be to use a sequential circuit that computes one addition for every clock pulse. We will discuss this more in a later section since it needs mecanisms we have not discussed yet.

10. Latches and flip-flops

In the same way that gates are the building blocks of combinatorial circuits, *latches* and *flip-flops* are the building blocks of sequential circuits.

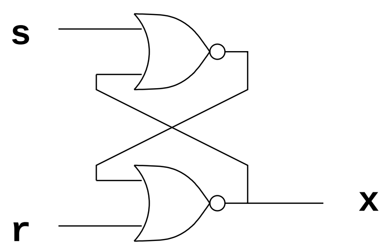
While gates had to be built directly from transistors, latches can be built from gates, and flip-flops can be built from latches. This fact will make it somewhat easier to understand latches and flip-flops.

Both latches and flip-flops are circuit elements whose output depends not only on the current inputs, but also on previous inputs and outputs. The difference between a latch and a flip-flop is that a latch does not have a *clock signal*, whereas a flip-flop always does.

10.1. Latches

How can we make a circuit out of gates that is not combinatorial ? The answer is *feed-back*, which means that we create *loops* in the circuit diagrams so that output values depend, indirectly, on themselves. If such feed-back is *positive* then the circuit tends to have stable states, and if it is *negative* the circuit will tend to oscillate.

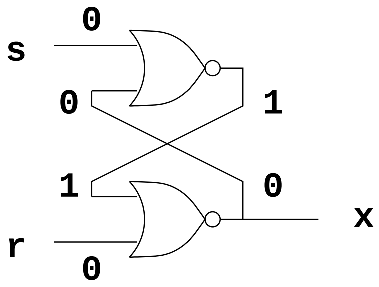
A latch has positive feedback. Here is an example of a simple latch :



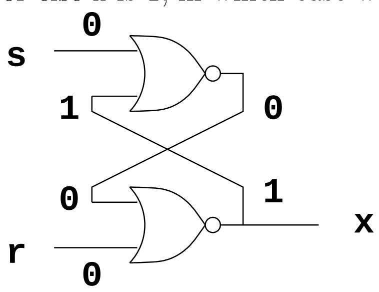
This latch is called *SR-latch*, which stands for *set* and *reset*.

It is not practical to use the methods that we have used to describe combinatorial circuits to describe the behavior of the SR-latch. Later, we will show a method for describing flip-flops and clocked sequential circuits. For now, we just rely on our intuition to describe how latches work.

The SR-latch is meant to have at most one of its inputs equal to `1` at any time. When both of its inputs are `0` it has two different stable states possible. Either `x` is `0`, in which case we have the following signal values :

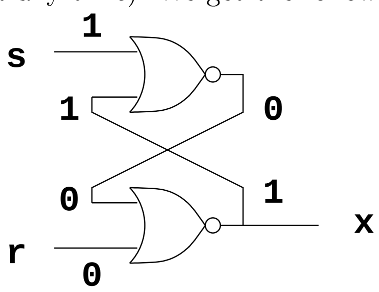


or else `x` is `1`, in which case we have the following signal values :



The actual value depends on the *history* of input values as we will show next.

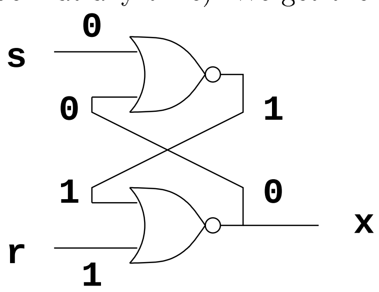
Now suppose that `s` is `1` (and therefore `r` is `0` since we allow at most one input to be `1` at any time). We get the following signal values :



The `1` on the `s` input makes sure the output of the upper *nor*-gate is `0`, and the two `0`s on the input of the lower *nor*-gate make sure the `x` output is `1`.

Now suppose the `s` input goes from `1` to `0`, while the `r` input remains at `0`. The second input of the upper *nor*-gate is `1`, so the transition from `1` to `0` of the `s` input, does not make any difference. The `x` output remains at `1`. In this case, if the `s` and `r` inputs are both `0`, there is only one possible stable state, the one that gives `x` the value `1`.

Conversely, suppose that `r` is `1` (and therefore `s` is `0` since we allow at most one input to be `1` at any time). We get the following signal values :

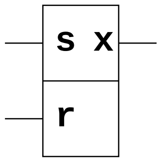


The `1` on the `r` input makes sure the `x` output is `0`, and the two `0`s on the input of the upper *nor*-gate make sure the output of the upper *nor*-gate is `0`.

Now suppose the `r` input goes from `1` to `0`, while the `s` input remains at `0`. The second input of the lower *nor*-gate is `1`, so the transition from `1` to `0` of the `r` input, does not make any difference. The output of the upper *nor*-gate remains at `1`. In this case, if the `s` and `r` inputs are both `0`, there is only one possible stable state, the one that gives `x` the value `0`.

From the discussion above, we conclude that the SR-latch is able to remember the last state of the inputs, in the sense that it remembers which of the two inputs, `s` or `r`, last had the value of `1`.

When we need to draw an SR-latch, we use the following symbol :

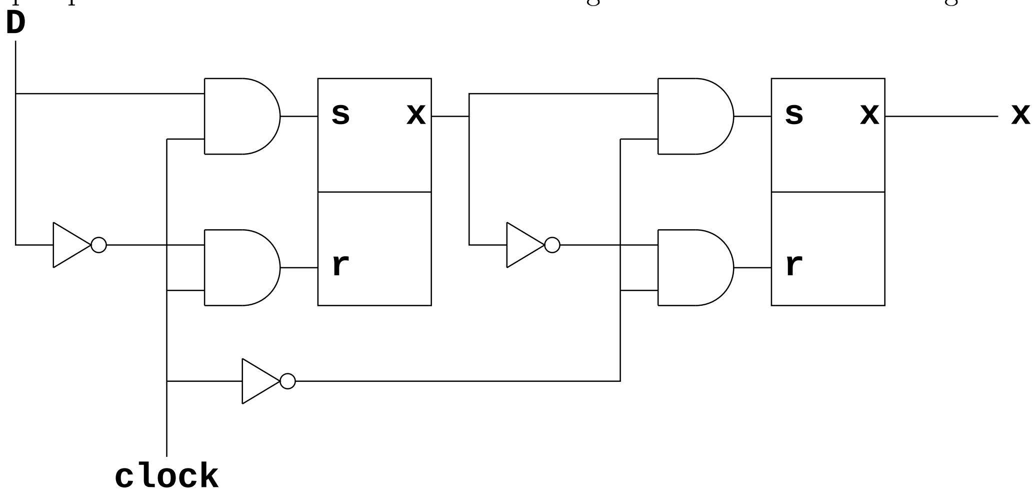


10.2. Flip-flops

Latches are *asynchronous*, which means that the output changes very soon after the input changes. Most computers today, on the other hand, are *synchronous*, which means that the outputs of all the sequential circuits change simultaneously to the rhythm of a global *clock signal*.

A *flip-flop* is a synchronous version of the latch. To complicate the situation even more, there are several fundamental types of flip-flops. Here, we shall only consider a type called *master-slave* flip-flop.

In addition to the fundamental types of flip-flops, there are minor variations depending on the number of inputs and how they control the state of the flip-flop. Here, we shall only consider a very simple type of flip-flop called a *D-flip-flop*. A master-slave D-flip-flop is built from two SR-latches and some gates. Here is the circuit diagram :



The leftmost SR-latch is called the *master* and the rightmost is called the *slave*.

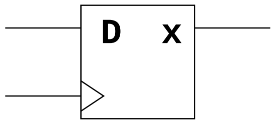
Let us first consider what happens when the clock signal is `1`. In this case, the two *and*-gates in front of the input of the master are *open*, i.e., they let the value of the `D`-input through to the `s` input of the master, and the inverse of the `D` input to the `r` input of the master. Thus, the value of the `D` input will go straight trough the master to the `x` output of the master. But the two *and*-gates of the slave re *closed*, i.e., their outputs are always `0`, so the slave keeps its old value.

When instead the clock signal is `0`, the reverse is true, i.e., the *and*-gates at the input of the master are *closed*, whereas the ones at the input of the slave are *open*. In this case, the flip-flop is completely insensitive to changes in input.

Now, let us consider what happens when the clock goes from `1` to `0`. For this to work, we have to assume that the input remains the same during a brief period from right before to right after the clock signal changes. The first thing that happens is that the *and*-gates at the input of the master *turn off*, i.e., they become insensitive to further changes in input. The value of the `x` output of the master is now the value of the `D` input right before the clock started changing. A brief moment later, the clock signal transition has traversed the *inverter* and reaches the *and*-gates of the slave. These gates *open*, allowing the `x` output of the master to be propagated to the `x` value of the slave. The `x` value of the slave, and therefore that of the entire flip-flop now contains the value of the `D` input right before the clock started changing. We can say that *the clock transition copied the input to the output* of the flip-flop. But at no point in time is there a direct path from input to output. The output changes only as a result of clock transitions from `1` to `0`.

Finally, let us see what happens when the clock goes from `0` to `1`. First, the *and*-gates of the master open, letting the value of the `D` input into the master. By the time the `D` value reaches the master, the clock signal transition reaches the *and*-gates of the slave, and turns them off before the possibly modified output of the master reaches the slave. Thus, the slave keeps its old value. From the outside, nothing seems to happen, since the output does not change. From now on, however, the master is open to changes in the input.

Here is the symbol we use for D-flip-flops :



The little triangle for the clock input indicates that this input is sensitive only to *transitions* as opposed to *levels* as described in the previous paragraph. Sometimes we do not draw the clock input at all when it is understood that it is there. Clock signals are boring since they are all just connected to each other. There is therefore little use to draw them all, and thereby clutter the diagram unnecessarily.

10.3. Summary

We have shown how to build a *D-flip-flop*. It copies its input to its output as a result of a *clock signal* transition from `1` to `0`. The value copied is the value the input has *immediately before the clock transition.* Most of the clock period, the D-flip-flop is insensitive to changes in the input.

We shall use this key characteristic of the D-flip-flop to build *synchronous sequential circuits.*

11. State table

11.1. Introduction

Where truth tables are used to describe combinatorial circuits, *state tables* are used to describe sequential circuits.

A state table looks similar to a truth table, except that in addition to the inputs, the left part of the table also contains the *values of the outputs*. The right part of the table does not contain the *values of the outputs*, but the *values of the outputs after the next clock transition*. To distinguish between the output values before and after the clock transition, we call them `x`, `y`, etc before the transition and `x'`, `y'`, etc after the transition.

Here is an example of a state table :

x	a	b	c	a'	b'	c'
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	1	0	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
0	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	1	0	1	0
1	0	1	0	0	1	1
1	0	1	1	1	0	0
1	1	0	0	1	0	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	0	0	0

As you can see, when the `x` input is `0`, the values of the outputs do not change as a result of a clock pulse. When the value of the `x` input is `1`, however, the values of the outputs (interpreted as a binary number) is one plus the values of the inputs (also interpreted as a binary number). This state table describes a counter with an extra input that lets you control whether it counts or not.

12. Sequential circuits

12.1. Pre-requisites

Read about latches and flip-flops first.

12.2. Introduction

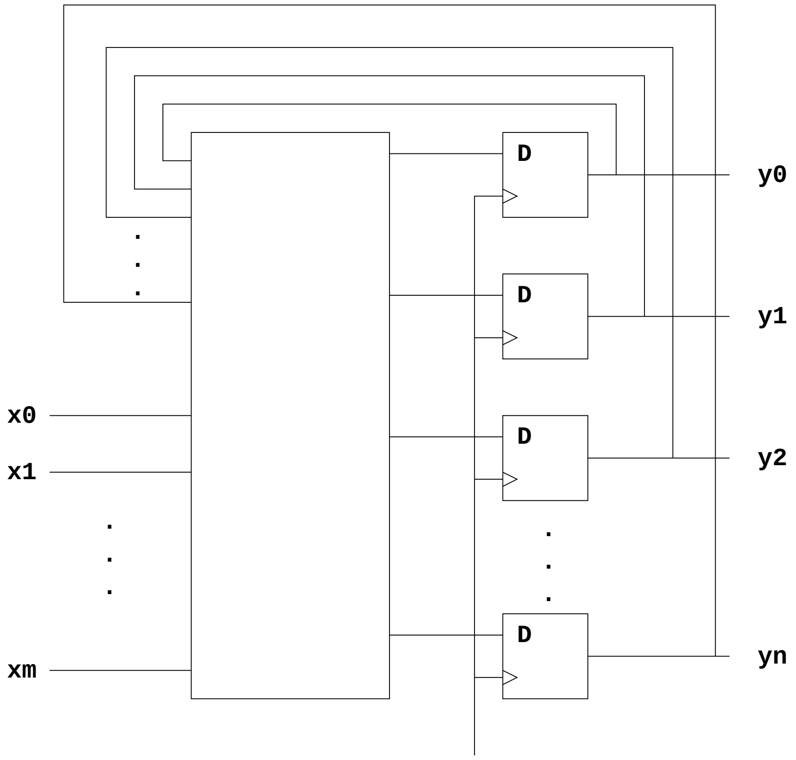
In the same way that combinatorial circuits are generalizations of gates, sequential circuits are generalizations of flip-flops.

In general, we define a *synchronous sequential circuit*, or just *sequential circuit* as a circuit with m inputs, n outputs, and a distinguished *clock input*. The description of the circuit is made with the help of a *state table*.

We have intentionally simplified our definition of sequential circuit. With our definition, the number of different states of the circuit is completely determined by the number of outputs. A more general definition separates the concept of output and the concept of *state*. Four our purposes, the additional generality is not needed, and we stall stick with our simple definition.

The same "goodness" criteria apply to the design of sequential circuits as to combinatorial circuits, i.e., number of transistors, speed, power consumption etc. Thus, as with combinatorial circuits, we are not going to discuss methods for obtaining optimal circuits, but only a very general method that in the worst case may waste a large number of tansistors. However, the purpose is not to turn the students into circuit designers, but to give them an idea of how sequential circuits work.

For a sequential circuit with m inputs and n outputs, our method uses n D-flip-flops (one for each output), and a combinatorial circuit with $m + n$ inputs and n outputs. Here is the general structure of the resulting circuit :

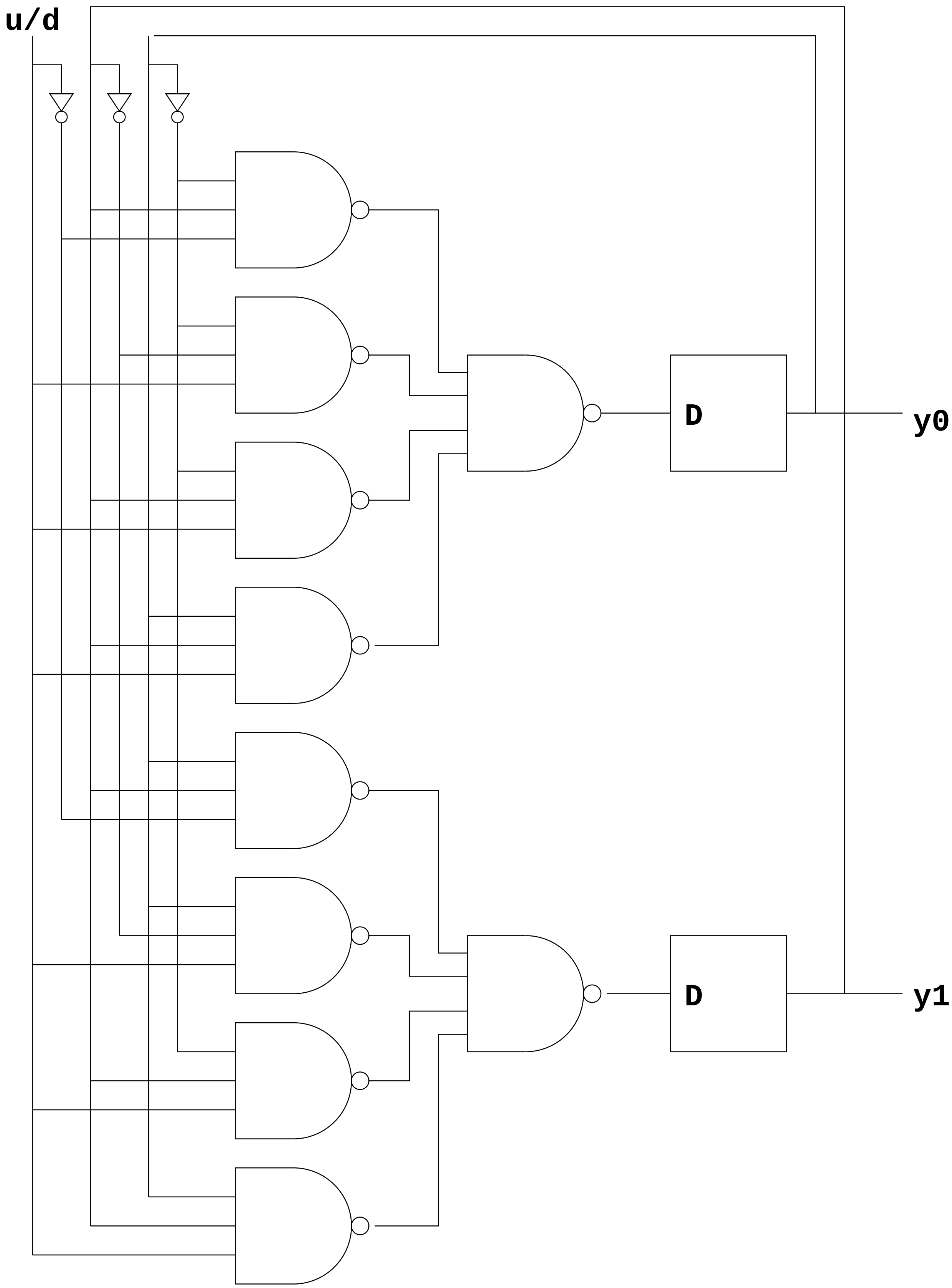


Since we are using D-flip-flops, the outputs of the circuit after the next clock pulse is exactly the same as the output of the combinatorial circuit. We can therefore use our general method for building combinatorial circuits, this time applied to the *state table* of the sequential circuit.

As an example, let us construct a 2-bit counter with an input indicating whether to count up or down (`0` means down and `1` means up). The counter should stop at `00` when counting down and at `11` when counting up. Here is the state table for such a counter :

u/d	y_1	y_0	y'_1	y'_0
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

With our general method, all we need to do is to build a sequential circuit from the truth table corresponding exactly to this state table. Here is the resulting circuit :


u/d
D
y0
D
y1

13. Register

13.1. Pre-requisites

Read about latches and flip-flops, and about sequential circuits first.

13.2. Registers

A *register* is a sequential circuit with $n + 1$ (not counting the clock) inputs and n output. To each of the outputs corresponds an input. The first n inputs will be called $\mathtt{x}_0$ trough $\mathtt{x}_{n-1}$ and the last input will be called `ld` (for load). The n outputs will be called $\mathtt{y}_0$ trough $\mathtt{y}_{n-1}$.

When the `ld` input is `0`, the outputs are uneffected by any clock transition. When the `ld` input is `1`, the `x` inputs are stored in the register at the next clock transition, making the `y` outputs into copies of the `x` inputs before the clock transition.

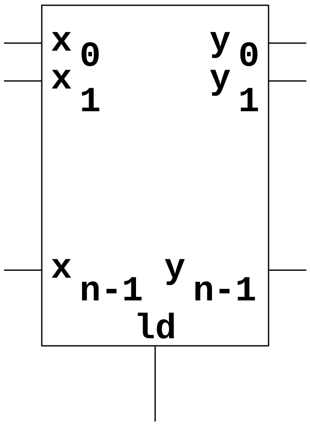
We can explain this behavior more formally with a state table. As an example, let us take a register with $n = 4$. The left side of the state table contains `9` columns, labeled x_0, x_1, x_2, x_3, `ld`, y_0, y_1, y_2, and y_3. This means that the state table has `512` rows. We will therefore abbreviate it. Here it is :

ld	x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0	y'_3	y'_2	y'_1	y'_0
0	–	–	–	–	c_3	c_2	c_1	c_0	c_3	c_2	c_1	c_0
1	c_3	c_2	c_1	c_0	–	–	–	–	c_3	c_2	c_1	c_0

As you can see, when `ld` is `0` (the top half of the table), the right side of the table is a copy of the values of the old outputs, independently of the inputs. When `ld` is `1`, the right side of the table is instead a copy of the values of the inputs, independently of the old values of the outputs.

Registers play an important role in computers. Some of them are visible to the programmer, and are used to hold variable values for later use. Some of them are hidden to the programmer, and are used to hold values that are internal to the central processing unit, but nevertheless important.

We use this symbol for registers (notice that we have omitted the clock signal, as it is understood) :



14. Counters

A *counter* is a sequential circuit with *0* inputs and n outputs. Thus, the value after the clock transition depends only on old values of the outputs. For a counter, the values of the outputs are interpreted as a sequence of *binary digits* (see the section on binary arithmetic).

We shall call the outputs o_0, o_1, $\ldots$, o_{n-1}. The value of the outputs for the counter after a clock transition is a binary number which is *one plus* the binary number of the outputs before the clock transition.

We can explain this behavior more formally with a state table. As an example, let us take a counter with $n = 4$. The left side of the state table contains `4` columns, labeled o_0, o_1, o_2, and o_3. This means that the state table has `16` rows. Here it is in full :

o_3	o_2	o_1	o_0	o'_3	o'_2	o'_1	o'_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	0	0

As you can see, the right hand side of the table is always one plus the value of the left hand side of the table, except for the last line, where the value is `0` for all the outputs. We say that the counter *wraps around.*

Counters (with some variations) play an important role in computers. Some of them are visible to the programmer, such as the program counter (PC). Some of them are hidden to the programmer, and are used to hold values that are internal to the central processing unit, but nevertheless important.

Important variations include :

- The ability to count up or down according to the value of an additional input
- The ability to count or not according the the value of an additional input
- The ability to clear the contents of the counter if some additional input is `1`

- The ability to act as a register as well, so that a predetermined value is loaded when some additional input is `1`
- The ability to count using a different representation of numbers from the normal (such as Gray-codes, 7-segment codes, etc)
- The ability to count with different increments that `1`

15. Binary multiplication

15.1. Pre-requisites

Read about sequential circuits first.

As we mentioned in the section on circuits for binary arithmetic, multiplication are (currently, at least) too complicated for a combinatorial circuit.

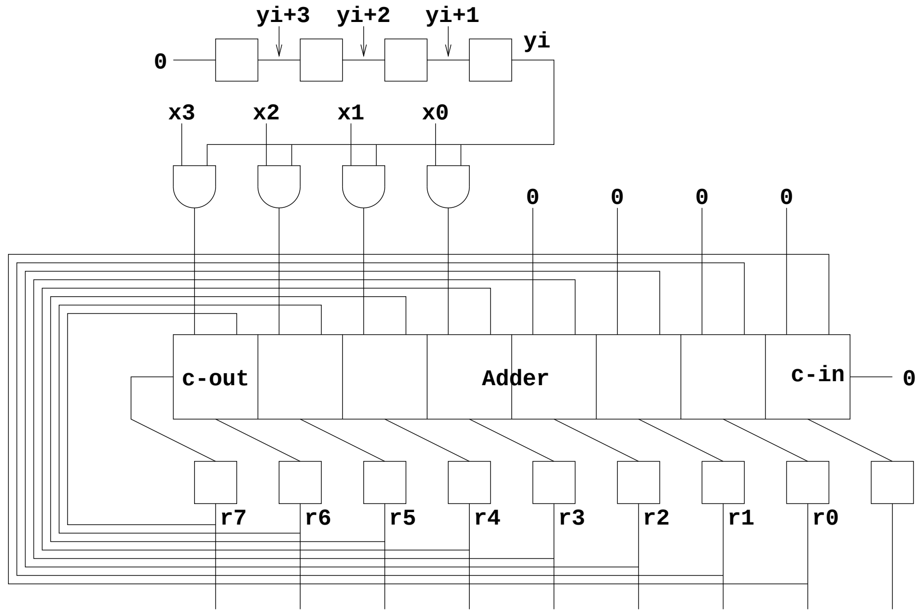
The solution to this problem is going to be to use a sequential circuit and to divide the work into several stages, one stage for each clock pulse.

The algorithm we are going to use is the same one that is used in ordinary binary multiplication, only we change the order of the operations slightly. In ordinary multiplication, we compute all the partial results of multiplying the first factor with each binary digit of the second factor. Then we add each partial result together to obtain the final result. In the modified verson, we keep an *accumulator* containing the sum of all the partial results computed so far. When we have computed the last partial result, we have also computed the final result.

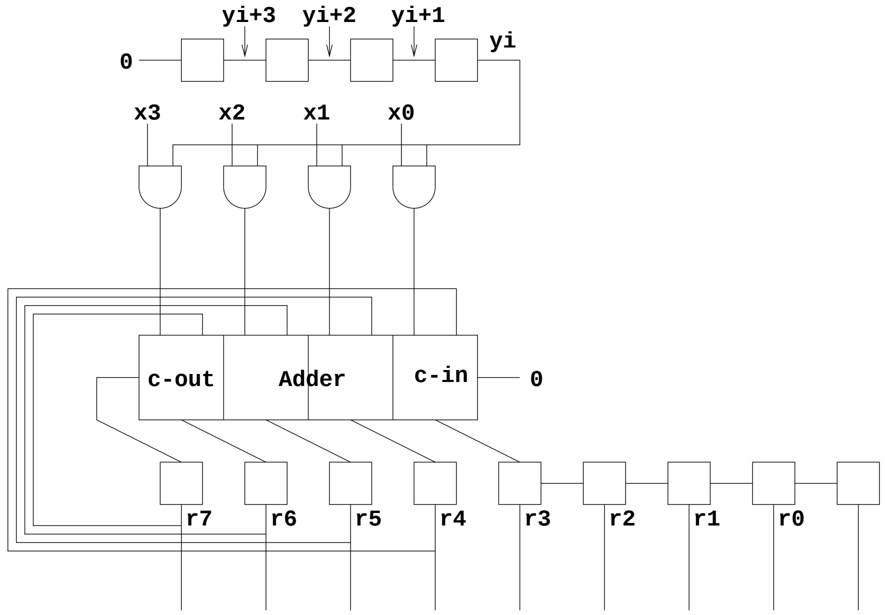
To see how this can be done, notice that the result of multiplying two n-digit positive binary numbers, the result may be as long as $2n$ digit long.

Let us denote the first factor by x, the second by y, and the result by r. We first write y as $\mathrm{y}_{n-1}2^{n-1}+\mathrm{y}_{n-2}2^{n-2}+...+\mathrm{y}_0 2^0$. At any point i in time, the result will contain x times $\mathrm{y}_{i-1}2^{n-1}+\mathrm{y}_{i-2}2^{n-2}+...+\mathrm{y}_0 2^{n-i}$. Obviously, when $i{=}n$ we have our final result. To get from step i to step $i+1$, we need to divide the result by 2 and add x times $\mathrm{y}_i 2^{n-1}$. But this is the same as first adding x times $\mathrm{y}_i 2^n$ and then dividing the result by 2. This process must be repeated n times.

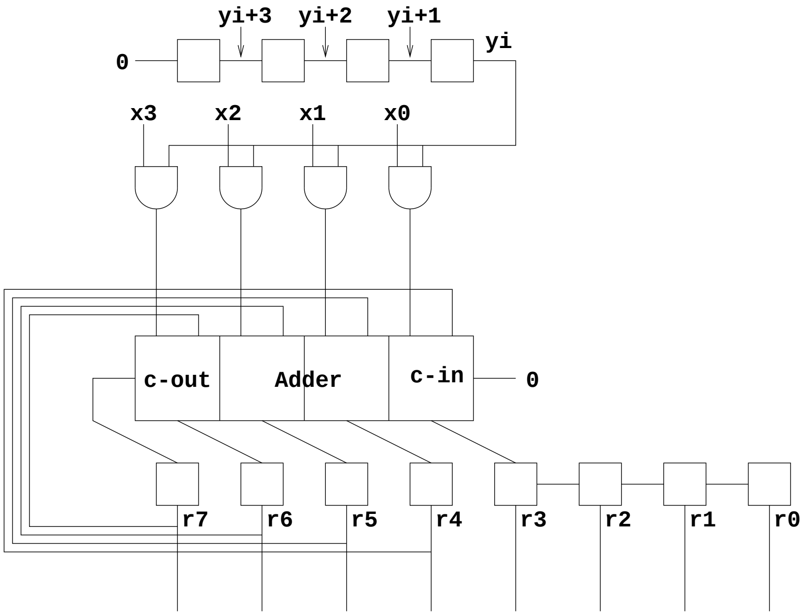
Here is our first attempt at a circuit for multiplication :



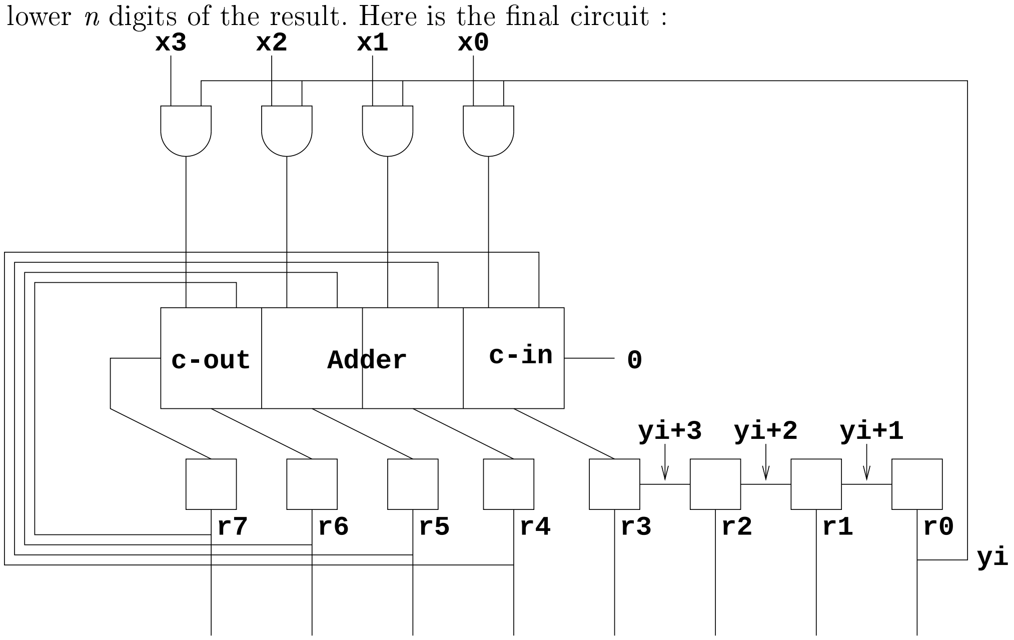
This circuit works fine, but it is wasteful. First, notice that the lower part of the adder always adds r_3, r_2, r_1, and r_0 with 0. Using an adder is therefore not necessary, and we can simplify the circuit like this :



Next, since r_3 (or r_{n-1} in general) contains 0 initially, the rightmost D-flip-flop always contains 0 as well, which is fortunate, because we never use it. We obtain this circuit :



For our final simplification, notice how the flip-flops containing r_{n-1} to r_0 and the n upper flip-flops never simultaneously contains any information. Initially, all of the upper flip-flops contain useful information, and none of the n rightmost bottom flip-flops do. After step one, r_{n-1} contains useful information, and the leftmost of the upper flip-flops no longer does, etc. We can therefore use only one set of n flip-flops to hold both y and the lower n digits of the result. Here is the final circuit :



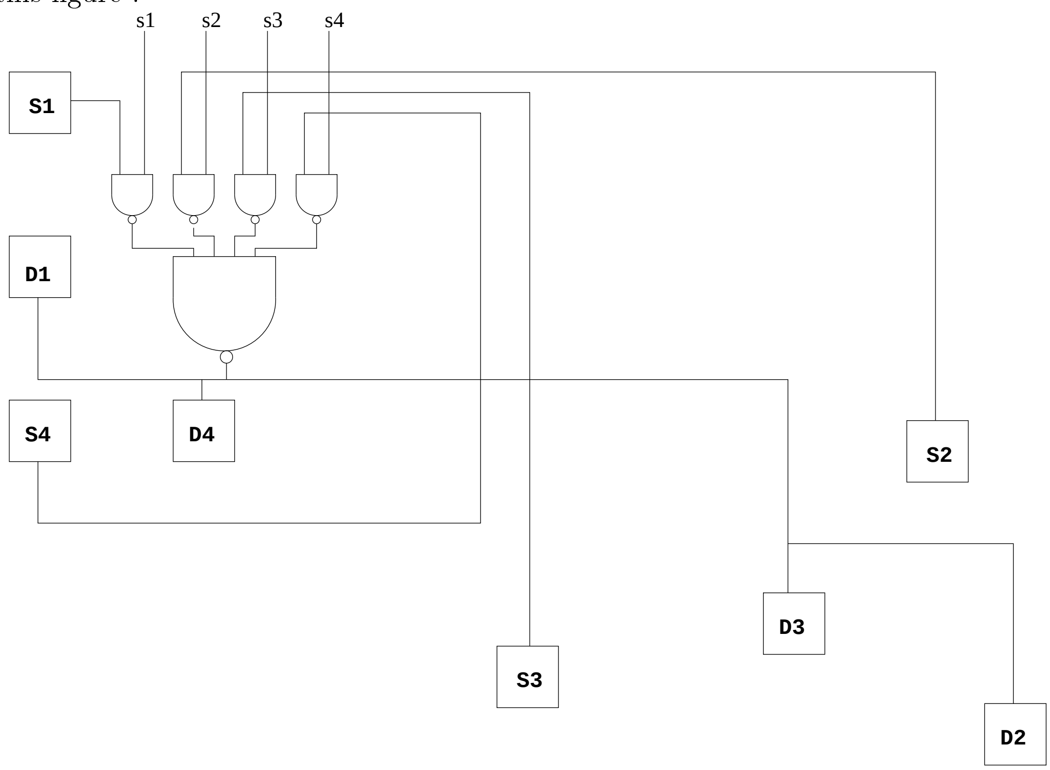
16. Tri-state logic

16.1. Introduction

Both combinatorial circuits and sequential circuits are fairly easy to understand, since we pretty much only have to understand logic gates and how they are interconnected.

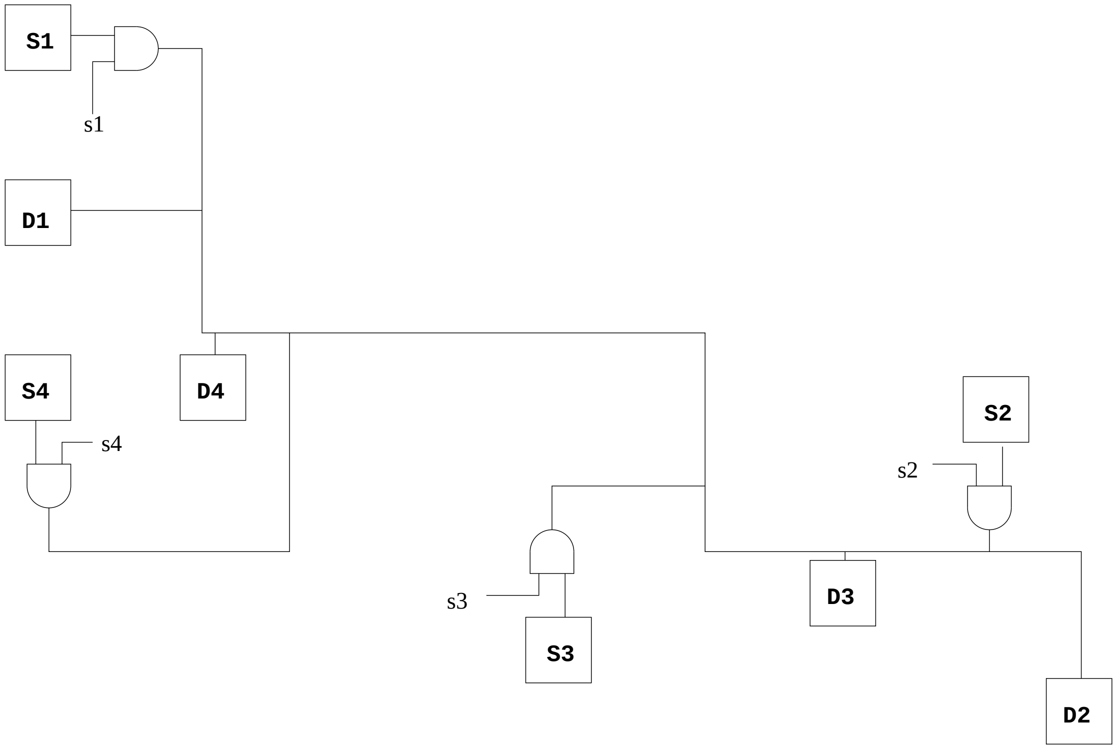
With *tri-state logic circuits*, this is no longer true. As their names indicate, they manipulate signals that can be in one of *three* states, as opposed to only `0` or `1`. While this may sound confusing at first, the idea is relatively simple.

Consider a fairly common case in which there are a number of source circuits S_1, S_2, etc in different parts of a chip (i.e., they are not real close together). At different times, exactly one of these circuit will generate some binary value that is to be distributed to some set of destination circuits D_1, D_2, etc also in different parts of the chip. At any point in time, exactly one source circuit can generate a value, and the value is always to be distributed to all the destination circuits. Obviously, we have to have some signals that *select* which source circuit is to generate information. Assume for the moment that we have singals s_1, s_2, etc for exactly that purpose. One solution to this problem is indicated in this figure :



As you can see, this solution requires that all outputs are routed to a central place.

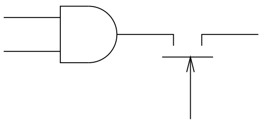
Often such solutions are impractical or costly. Since only one of the sources is "active" at one point, we ought to be able to use a solution like this :



However, connecting two or more outputs together is likely to destroy the circuits. The solution to our problem is to use *tri-state* logic.

16.2. Return to the transistors

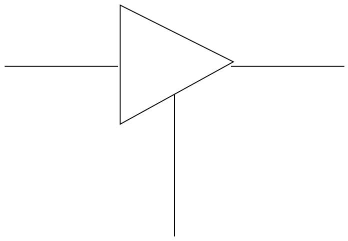
A *tri-state* circuit (combinatorial or sequential) is like an ordinary circuit, except that it has an additional input that we shall call *enable*. When the enable input is `1`, the circuit behaves exactly like the corresponding normal (not tri-state) circuit. When the enable input is `0`, *the outputs are completely disconnected from the rest of the circuit*. It is as if there we had taken an ordinary circuit and added a *switch* on every output, such that the switch is open when enable is `0` and closed if enable is `1` like this :



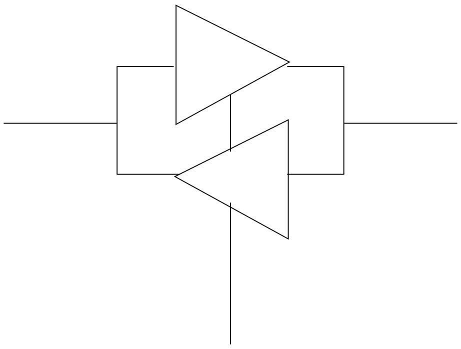
which is pretty close to the truth. The switch is just another transistor that can be added at a very small cost.

Any circuit can exist in a tri-state version. However, as a special case, we can convert any ordinary circuit to a tri-state circuit, by using a special tri-state combinatorial circuit that simply copies its inputs to the outputs, but that also has an *enable* input. We call such a circuit a *bus driver* for reasons that will become evident when we discuss buses.

A bus driver with one input is drawn like this :



Here is a version for bidirectional signals :



17. Buses

With tri-state logic we can use a single wire to connect all the sources and all the destinations of some common signal. This wire may save a significant amount of space on the chip and also make the distance a signal has to travel considerably shorter.

Such a wire, or in general, a collection of wires that carry some multi-bit information, is called a *bus*.

18. Memories

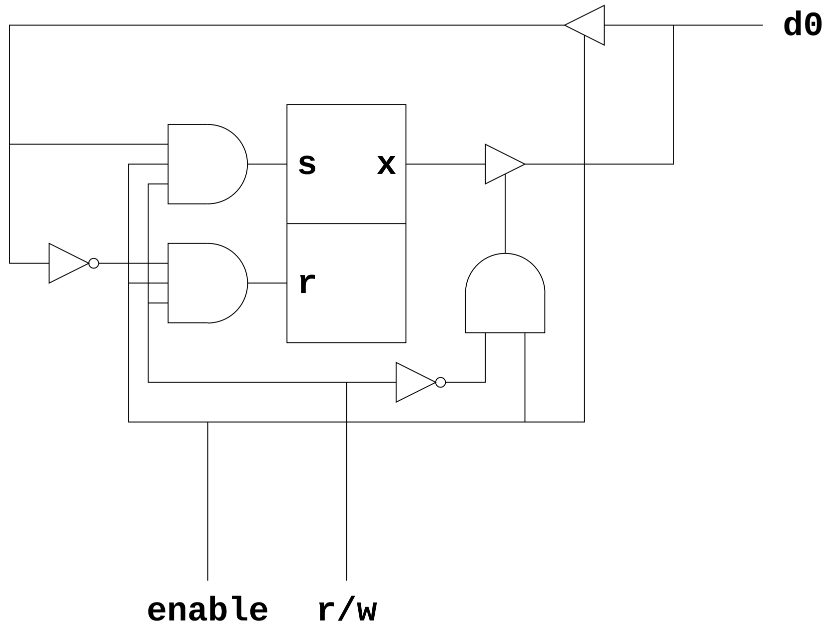
A memory is neither a sequential circuit (since we require sequential circuits to be clocked, and memories are not clocked), nor a combinatorial circuit, since its output values depend on past values.

In general, a memory has m inputs that are called the *address inputs* that are used to select exactly one out of 2^m *words*, each one consisting of n bits.

Furthermore, it has n connectors that are *bidirectional* that are called the *data lines.* These data lines are used both as *inputs* in order to store information in a word selected by the address inputs, and as *outputs* in order to recall a previously stored value. Such a solution reduces the number of required connectors by a factor two.

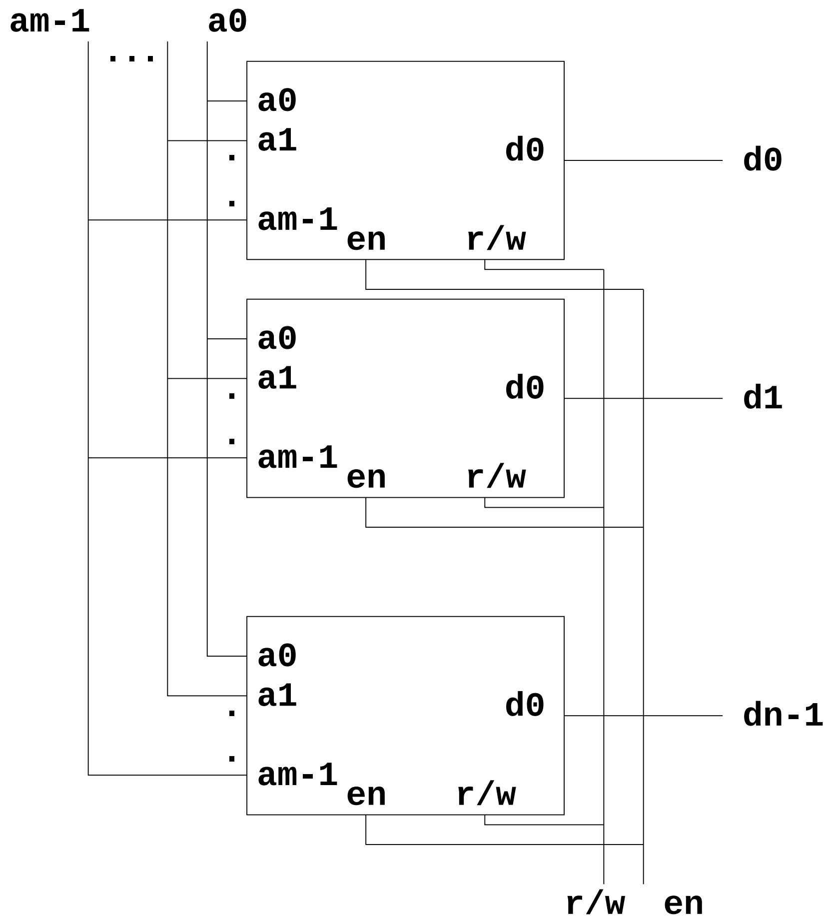
Finally, it has an input called *enable* (see the section on tri-state logic for an explanation) that controls whether the data lines have defined states or not, and an input called *r/w* that determines the direction of the data lines.

A memory with an arbitrary value of m and an arbitrary value of n can be built from memories with smaller values of these parameters. To show how this can be done, we first show how a one-bit memory (one with $m = 0$ and $n = 1$) can be built. Here is the circuit :



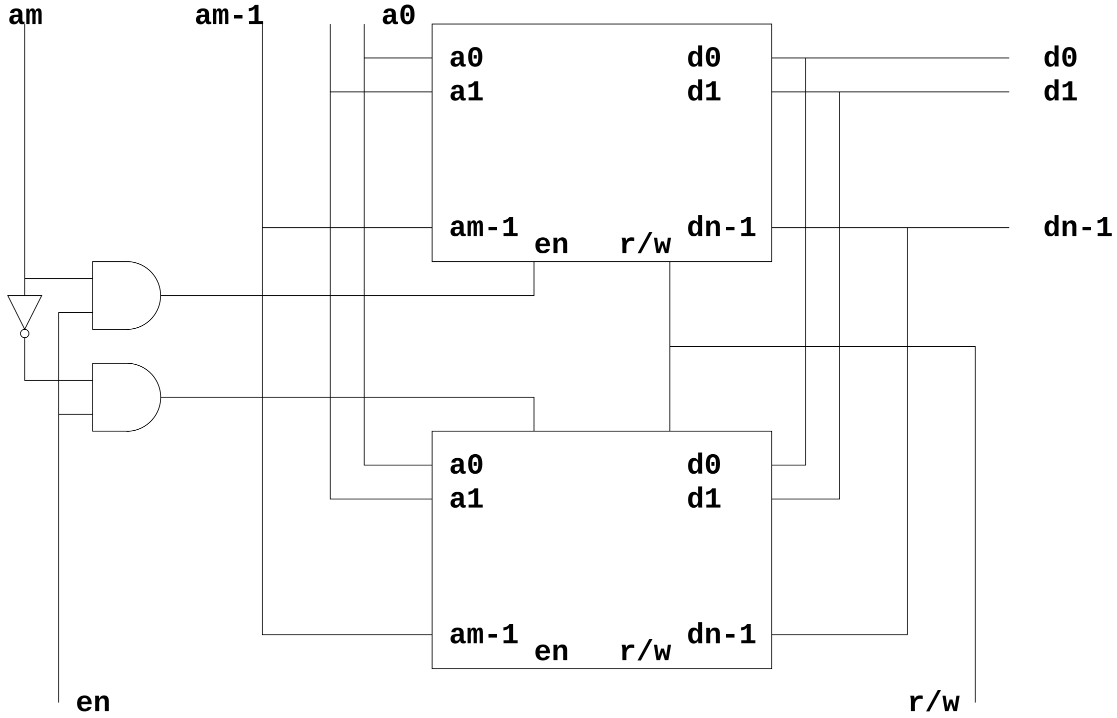
The central part of the circuit is an SR-latch that holds one bit of information. When `enable` is `0`, the output d_0 is isolated both from the inputs to and the output from the SR-latch. Information is passed from d_0 to the inputs of the latch when `enable` is `1` and `r/w` is `1` (indicating *write*). Information is passed from the output `x` to d_0 when `enable` is `1` and `r/w` is `0` (indicating *read*).

Now that we know how to make a one-bit memory, we must figure out how to make larger memories. First, suppose we have n memories of 2^m *words*, each one consisting of a single bit. We can easily convert these to a single memory with 2^m *words*, each one consisting of a n bits. Here is how we do it :



We have simply connected all the address inputs together, all the enables together, and all the read/writes together. Each one-but memory supplies one of the bits of the n-bit word in the final circuit.

Next, we have to figure out how to make a memory with more words. To show that, we assume that we have two memories each with m address inputs and n data lines. We show how we can connect them so as to obtain a single memory with $m + 1$ address inputs and n data lines. Here is the circuit :



As you can see, the additional address line is combined with the `enable` input to select one of the two smaller memories. Only one of them will be connected to the data lines at a time (because of the way tri-state logic works).

19. Read-only memories

A *read-only memory* (or *ROM* for short), is like an ordinary memory, except that it does not have the capability of writing. Its contents is fixed at the factory.

Since the contents cannot be altered, we don't have a `r/w` signal. Except for the `enable` signal, a ROM is thus like an ordinary combinatorial circuit with m inputs and n outputs.

If a ROM is like an ordinary sequential circuit, then why do we treat it separately? The answer is that ROMs are usually *programmable*. They are often sold with a contents of all `0`s or all `1`s. The user can then stick it in a special machine and fill it with the desired contents, i.e. the ROM can be *programmed*. In that case, we sometimes call it a PROM (programmable ROM).

Some varieties of PROMS can be erased and re-programmed. The way they are erased is typically with ultra-violet light. When the PROM can be erased, we sometimes call it EPROM (erasable PROM).

Deuxième partie .

Specific Combinatorial and sequential circuits

1. Counter with clear

A *counter with clear* is a sequential circuit with *1* input and n outputs. It differs from an ordinary counter in that it has an input that sets the contents of the counter to `0`.

Here is a state table for a 4-bit counter :

cl	o_3	o_2	o_1	o_0	o'_3	o'_2	o'_1	o'_0
0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	1	0
0	0	0	1	0	0	0	1	1
0	0	0	1	1	0	1	0	0
0	0	1	0	0	0	1	0	1
0	0	1	0	1	0	1	1	0
0	0	1	1	0	0	1	1	1
0	0	1	1	1	1	0	0	0
0	1	0	0	0	1	0	0	1
0	1	0	0	1	1	0	1	0
0	1	0	1	0	1	0	1	1
0	1	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0	1
0	1	1	0	1	1	1	1	0
0	1	1	1	0	1	1	1	1
0	1	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0
1	0	1	1	0	0	0	0	0
1	0	1	1	1	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0
1	1	0	1	0	0	0	0	0
1	1	0	1	1	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0
1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0

As you can see, our conter behaves like an ordinary one when the `cl` signal is `0`, and always goes to `0` when the `cl` signal is `1`.

2. Counter register

A *counter register* is a sequential circuit with $n + 1$ inputs and n outputs. It differs from an ordinary counter in that it can also function as a register. It differes from a register in that when not explictly loaded, it works as a counter, rather than maintaining its previous value.

Here is a state table (abbreviated) for a 4-bit counter register :

ld	i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0	o'_3	o'_2	o'_1	o'_0
0	-	-	-	-	0	0	0	0	0	0	0	1
0	-	-	-	-	0	0	0	1	0	0	1	0
0	-	-	-	-	0	0	1	0	0	0	1	1
0	-	-	-	-	0	0	1	1	0	1	0	0
0	-	-	-	-	0	1	0	0	0	1	0	1
0	-	-	-	-	0	1	0	1	0	1	1	0
0	-	-	-	-	0	1	1	0	0	1	1	1
0	-	-	-	-	0	1	1	1	1	0	0	0
0	-	-	-	-	1	0	0	0	1	0	0	1
0	-	-	-	-	1	0	0	1	1	0	1	0
0	-	-	-	-	1	0	1	0	1	0	1	1
0	-	-	-	-	1	0	1	1	1	1	0	0
0	-	-	-	-	1	1	0	0	1	1	0	1
0	-	-	-	-	1	1	0	1	1	1	1	0
0	-	-	-	-	1	1	1	0	1	1	1	1
0	-	-	-	-	1	1	1	1	0	0	0	0
1	c_3	c_2	c_1	c_0	-	-	-	-	c_3	c_2	c_1	c_0

As you can see, the counter register behaves like an ordinary counter when the `ld` signal is `0` and as a register when the `ld` signal is `1`.

3. Counter register with clear

A *counter register with clear* is a sequential circuit with $n + 2$ inputs and n outputs. It differs from an ordinary counter register in that it also has a signal `cl` for clearing the contents.

It is a matter of taste what we want the circuit to do when both the `ld` and the `cl` signals are `1`. In our application, we want the `cl` signal to take priority over the `ld` signal.

Here is a state table (abbreviated) for a 4-bit counter register with clear :

cl	ld	i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0	o'_3	o'_2	o'_1	o'_0
0	0	-	-	-	-	0	0	0	0	0	0	0	1
0	0	-	-	-	-	0	0	0	1	0	0	1	0
0	0	-	-	-	-	0	0	1	0	0	0	1	1
0	0	-	-	-	-	0	0	1	1	0	1	0	0
0	0	-	-	-	-	0	1	0	0	0	1	0	1
0	0	-	-	-	-	0	1	0	1	0	1	1	0
0	0	-	-	-	-	0	1	1	0	0	1	1	1
0	0	-	-	-	-	0	1	1	1	1	0	0	0
0	0	-	-	-	-	1	0	0	0	1	0	0	1
0	0	-	-	-	-	1	0	0	1	1	0	1	0
0	0	-	-	-	-	1	0	1	0	1	0	1	1
0	0	-	-	-	-	1	0	1	1	1	1	0	0
0	0	-	-	-	-	1	1	0	0	1	1	0	1
0	0	-	-	-	-	1	1	0	1	1	1	1	0
0	0	-	-	-	-	1	1	1	0	1	1	1	1
0	0	-	-	-	-	1	1	1	1	0	0	0	0
0	1	c_3	c_2	c_1	c_0	-	-	-	-	c_3	c_2	c_1	c_0
1	-	-	-	-	-	-	-	-	-	0	0	0	0

This is exactly the circuit we will use for the *micro program counter* (or micro PC) of our simple example computer, except that in our example computer, the micro PC will be 6 bits wide instead of 4.

4. Counter register with clear and increment

A *counter register with clear and increment* is a sequential circuit with $n + 3$ inputs and n outputs. It differs from an ordinary counter register with clear in that it also has a signal `incr` for explicit counting. When the `incr` signal is `0` the output does not change, and when `incr` is `1`, the circuit works as a counter.

For this circuit, we shall allow the `ld` signal to take priority over the `incr` signal, so that if both signals are `1`, we load a new contents from the inputs. We shall also make the `cl` have the highest priority of all.

Here is a state table (abbreviated) for a 4-bit counter register with clear and increment :

cl	ld	incr	i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0	o'_3	o'_2	o'_1	o'_0
0	0	0	-	-	-	-	c_3	c_2	c_1	c_0	c_3	c_2	c_1	c_0
0	0	1	-	-	-	-	0	0	0	0	0	0	0	1
0	0	1	-	-	-	-	0	0	0	1	0	0	1	0
0	0	1	-	-	-	-	0	0	1	0	0	0	1	1
0	0	1	-	-	-	-	0	0	1	1	0	1	0	0
0	0	1	-	-	-	-	0	1	0	0	0	1	0	1
0	0	1	-	-	-	-	0	1	0	1	0	1	1	0
0	0	1	-	-	-	-	0	1	1	0	0	1	1	1
0	0	1	-	-	-	-	0	1	1	1	1	0	0	0
0	0	1	-	-	-	-	1	0	0	0	1	0	0	1
0	0	1	-	-	-	-	1	0	0	1	1	0	1	0
0	0	1	-	-	-	-	1	0	1	0	1	0	1	1
0	0	1	-	-	-	-	1	0	1	1	1	1	0	0
0	0	1	-	-	-	-	1	1	0	0	1	1	0	1
0	0	1	-	-	-	-	1	1	0	1	1	1	1	0
0	0	1	-	-	-	-	1	1	1	0	1	1	1	1
0	0	1	-	-	-	-	1	1	1	1	0	0	0	0
0	1	-	c_3	c_2	c_1	c_0	-	-	-	-	c_3	c_2	c_1	c_0
1	-	-	-	-	-	-	-	-	-	-	0	0	0	0

This is exactly the circuit we will use for the *program counter* (or PC) of our simple example computer, except that the PC in our computer will be 8 bits wide instead of 4.

5. Micro memory

The *micro memory* of a processor is a combinatorial circuit sometimes in the form of a read-only memory, sometimes in the form of an ordinary combinatorial circuit. At some point, it was fairly common to be able to alter the contents of the micro memory in order to create specialized instructions for particular programs. In that case, the micro memory was similar to an ordinary read/write memory. Such processors are called *micro programmable*.

The number of address lines required depends on the number of instructions of the processor and their complexity. In a traditional CISC machine (Complex Instrution Set Computer) such as the VAX, the micro memory can be huge, perhaps structured with subroutines and other control structures otherwise seen on the instruction level. For simple RISC machines (Reduced Instrution Set Computer), the micro memory can be so simple that an ordinary combinatorial circuit will do. In our example computer, the micro memory will have 6 address lines, for a maximum of 64 addresses.

The number of data lines is the same as the number of micro operations necessary in the machine. In our example computer we initially need 15 such micro operations, but we will extend it somewhat in order to accomodate new functionality such as conditional jumps, stack pointers, etc.

6. Instruction decoder

The *instruction decoder* of a processor is a combinatorial circuit sometimes in the form of a read-only memory, sometimes in the form of an ordinary combinatorial circuit. Its purpose is to translate an instruction code into the address in the micro memory where the micro code for the instruction starts.

In our example computer, the input to the instruction decoder is the low 5 bits of the data bus, for a total of 32 different instruction codes. The output is 6 bits to correspond to the width of the micro PC and the micro memory.

7. Arithmetic and logic unit (ALU)

An *arithmetic and logic unit*, or ALU, of a processor is a combinatorial circuit that is capable of performing the basic operations such as addition, subtraction, bitwise and, bitwise or, etc.

We exclude mutiplication, since it is not implemented as a combinatorial circuit.

A possible implementation of the ALU is that it performs all the operations in parallel and, with the help of a multiplexer selects one of the outputs according to the operation wanted.

In our example computer, the ALU will have two 8-bit inputs, an 8-bit output (later extended to 9 for overflow and carry detection), and 3 control lines. The three control lines will select the operation to be performed. In our example computer, 000 means copy (output is the same as first input), 001 means shift left (output is the first input shifted left by one position), 010 means shift right, 011 means add, 100 means subtract, 101 means logic and, 110 means logic or, and 111 means logic not (output is first input with bits inverted).

8. The first computer

Here is a diagram of our first example computer :

To the left in the diagram, you find the micro memory containing up to 64 different addresses each with 15 bits (numbered 1 to 15) that we call the *micro operations* or MOPs for short. The MOPs control every aspect of the computer.

To the left, you also find the micro PC, which is a counter register with clear. The width of the micro PC is 6 bits for a total of 64 different values. The micro PC is controlled by MOPs number 1 and 2. When MOP number 1 has the value 1, the micro PC will be cleared after the next clock pulse. Similarly, MOP number 2 will cause a new value to be loaded into the micro PC from the instruction decoder. The micro PC can also be cleared manually from the *reset* button. Since the `clr` input of the micro PC takes priority over the `ld` input, this will always happen when the reset button is pressed, independently of the value of MOP number 2.

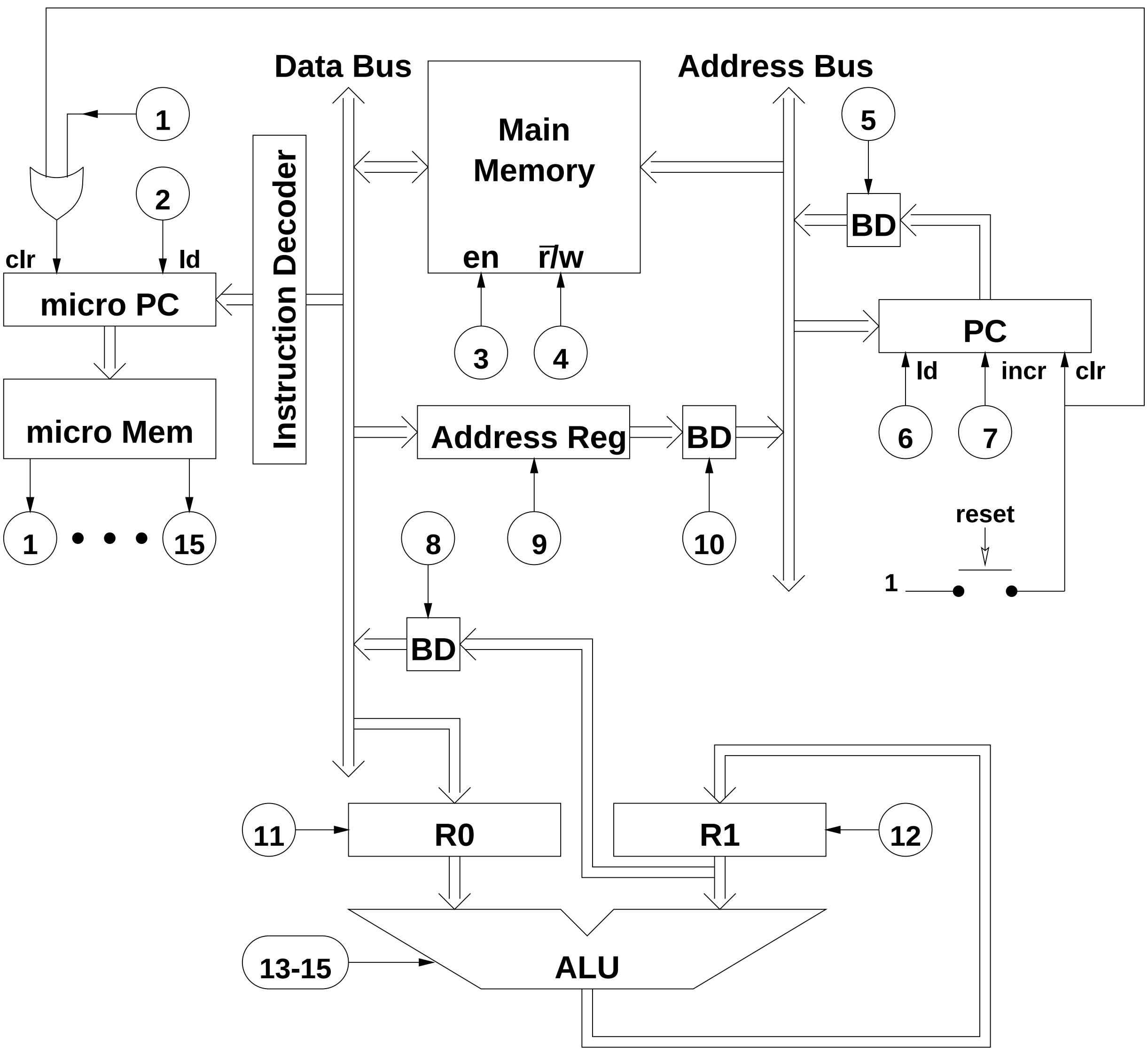
The final item to the left of the data bus is the instruction decoder. Its job is to translate instruction codes to addresses in the micro memory where the micro program for the instruction starts. The input to the instruction decoder is the low 5 bits of the data bus, and the output is 6 bits wide.

The lower part of the diagram is the arithmetic unit with two general-purpose registers, R0 and R1. Whether R0 is loaded from the data bus is determined by MOP number 11. Whether R1 is loaded from the output of the ALU is controlled by MOP number 12. The ALU uses MOPS 13 to 15 to control what operation is to be performed. The inputs (8 bits wide) are taken from the registers R0 and R1 and the output (also 8 bits wide for now) goes to the input of R1. The output of R1 can be connected to the data bus by MOP number 8 which is connected to a bus driver (BD) which in effect closes the connection from the output of R1 to the data bus.

The upper middle part is the main memory. The `enable` input is conrolled by MOP number 3, and its `r/w` input is controlled by MOP number 4. When MOP number 4 has the value 1, the memory is written. The address of the memory is taken from the address bus and that date is output to the data bus (in the case of read) or taken from the data bus (in the case of write). Thus the data lines of the memory are bidirectional.

In the middle of the diagram, you find another register called the address register. It is used to communicate data from the data bus to the address bus. It is loaded whenever MOP number 9 has the value 1. The contents of the address register can be communicated to the address bus through a bus driver conrolled by MOP number 10. This way, data from main memory or from R1 can be used as addresses in main memory.

To the right of the diagram, you find the program counter (or PC) in the form of a counter register with clear and explicit increment. The `clr` signal takes priority so that the reset button always works. If the `clr` signal is 0 but the `ld` signal (MOP 6) is 1, then PC is loaded from the address bus. If `clr` and `ld` are 0 but `incr` (MOP 7) is 1, PC is


Data Bus
Main Memory
Address Bus
1
2
5
clr
ld
micro PC
Instruction Decoder
BD
en
r/w
3
4
PC
ld
incr
clr
micro Mem
Address Reg
BD
6
7
1
15
8
9
10
reset
1
BD
R0
R1
11
12
13-15
ALU

incremented after the next clock pulse. Finally, if `clr`, `ld` and `incr` are all 0, then PC retains its value after the clock pulse. The value of PC can be communicated to the data bus through a bus driver by MOP number 5. This is the way instructions are fetched for execution.

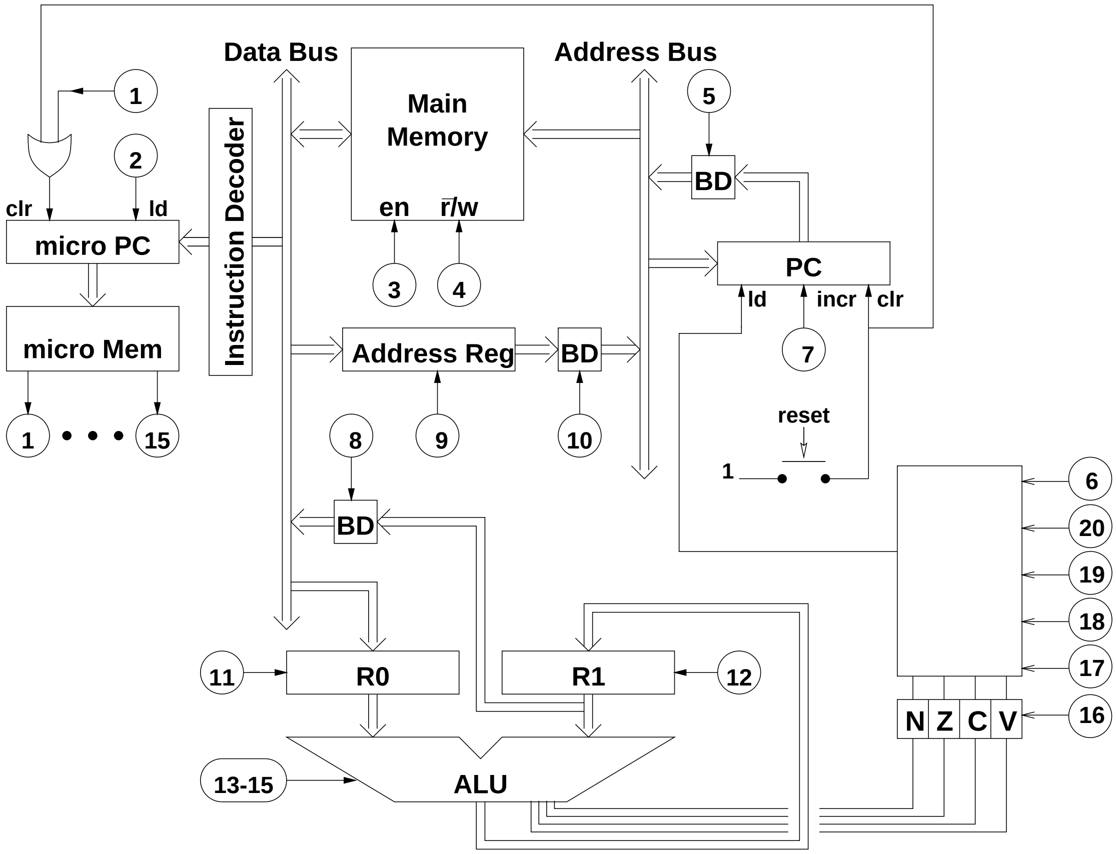
Troisième partie .

Our Simple computer

1. Conditional jumps

Our first computer was not able to jump conditionally. It only has a `JAL` (jump always) instruction.

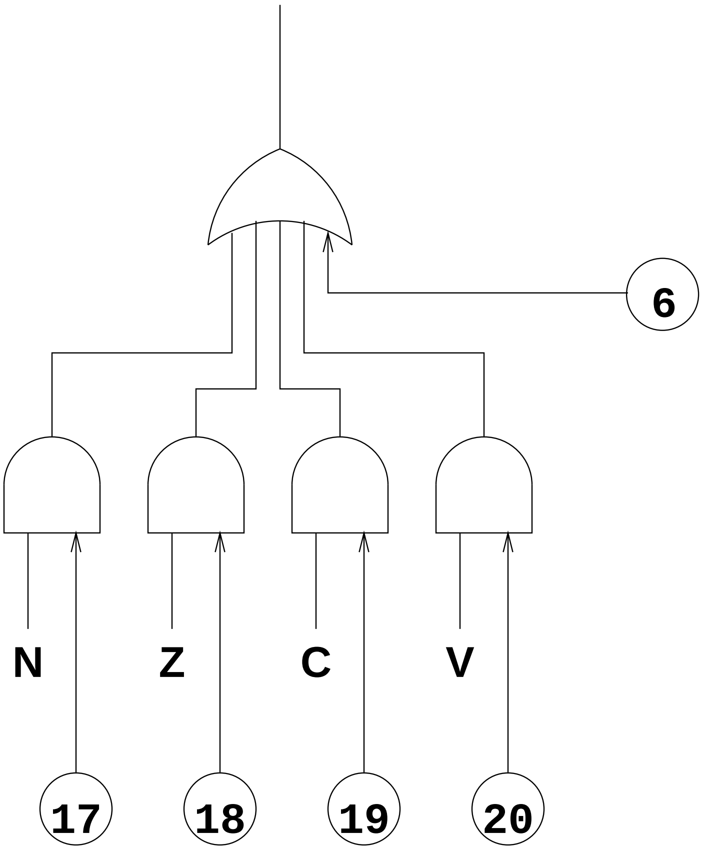
In fact the first computer does not even have the circuits necessary to implement such an instruction. First we need an additional register to hold information about the last operation computed by the `ALU`. This register is called the *condition codes*, usually abbreviated `CC`. The condition codes is a 4-bit register and the bits indicate respectively whether the value of the last operation was negative (the bit `N`), was zero (the bit `Z`), generated a carry (the bit `C`), or generated an overflow (the bit `V`). The `CC` register will be controled by another micro-operation which, when `1` will load the register with the information generated by the `ALU`. We could have used the same micro-operation as the one used to load `R2` since these operations are currently used simultaneously. However, in the future, we might want to control them separately, for instance in order to write a comparison instruction. Such an instruction is identical to a subtract instruction, except that the result of the subtraction is not kept, only the condition codes. In addition to the condition codes, we we also need to control the `ld` signal of `PC` in more detail. This signal is currently `1` if and only if we generate the micro operation for unconditional jumps. It should now be generated by a more complicated circuit that generates `1` also when the micro memory has issued a *jump zero* micro operation and the `Z` bit is set, etc. Here is our modified computer :



For completeness, here is the (abbreviated) state table for the condition code register. The signals `an`, `az`, `ac`, and `av` are the outputs from the `ALU` :

ld	an	az	ac	av	N	Z	C	V	N'	Z'	C'	V'
0	a	b	c	d	e	f	g	h	e	f	g	h
1	a	b	c	d	e	f	g	h	a	b	c	d

As you can see, it is an ordinary register. Now we must explain what is in the box that computes the `ld` signal of `PC`. It is a combinatorial circuit with 9 inputs, so the truth table will be quite large. It is much easier to show the circuit directly. Here it is :



As you can see, the output (the `ld` signal of `PC`) is 1 if either mop number 6 is one, or if both the bit `N` and mop number 17 are one, or if both the bit `Z` and mop number 18 are 1, or if both the bit `C` and mop number 19 are 1, or finally if both the bit `V` and mop number 20 are 1.

As mop 6 indicates unconditional jump, generating this mop automatically makes the `ld` signal of `PC` 1, so that `PC` is loaded with a new address just like before.

When we want to execute say a `JN` (jump if negative) instruction, we will emit mop number 17. If then at the same time, bit `N` is 1, `PC` will be loaded with the new value. If bit `N` is 0, `PC` will keep its old value, which is the address of the instruction immediately following the `JN` instruction and no jump will take place. A similar argument is valid for the `Z`, `C`, and `V` bits.

Now, we have to start thinking about the contents of the micro memory. First, since we have increased the number of mops from 15 to 20, we have to make the micro memory wider, adding these bits to existing instructions. The only thing we have to make sure is that mop 16 is generated at the same time as mop 12 so that the condition codes will be loaded at the same time as `R1`.

2. Subprograms

So far, our architecture can do simple programs with conditional jumps, which corresponds to loops in high-level languages. But, high-level languages can do more interesting things. One such feature is *subprograms*, i.e. procedures and functions.

We might for instance have a suprogram like this (in C) :

```
h()
{
  ...
  return;
}
```

and then two or more different callers like this :

```
f()
{
  h();
}
...
g()
{
  h();
}
```

While we can implement the *call* to the subprogram using a JAL instruction, the problem is with the return address. The subprogram cannot simply execute a JAL instruction at the end, since it does not know whether to jump to a place inside the function `f` or a place inside the function `g`.

Early versions of implementations of the Fortran programming language solved the problem by storing the return address in the memory cell immediately preceding the subprogram itself, and using an instruction JIN (jump indirect) to return to the caller. The JIN instruction is similar to the JAL instruction in that it jumps unconditionally. But whereas the argument of the JAL instruction is the address to jump to, the argument of the JIN instruction is the address of a memory cell containing the address to jump to.

With such an instruction, we can implement the subprogram above like this :

```
h-1:   0
h:       ...
         jin h-1
```

Here, we have two addresses, symbolically indicated as `h-1` and `h` respectively. The address `h-1` is simply `h` minus one, so both are known at compile time. The address `h-1` is used to store the return address, whereas the address `h` is the entry point of the subprogram. Notice that when the subprogram has finished its computation, it jumps indirectly to the address stored in the cell whose address is `h-1`. We can now implement our callers like this :

```
f:      ...
        ldimm fhret
        copy
        st h-1
        ...
        jal h
fhret:  ...
...
f:      ...
        ldimm ghret
        copy
        st h-1
        ...
        jal h
ghret:  ...
```

Here, we have introduced in each caller, an address to return to after the subprogram has finished (fhret and ghret). As you can see, we use this address as argument to an LDIMM instruction. We then store the address in the cell reserved for the return address in the callee (the subprogram). Finally, we jump unconditionally (JAL instruction) to the entry point of the subprogram.

We know have a rudimentary *subprogram call protocol*, i.e., a set of rules that determines exactly the responsibilities of a caller and a callee in order for subprograms to work.

Our call protocol depends on the existance of a JIN instruction, which we don't have. This instruction can, however, be implemented without further modifications to the existing architecture. The plan for its implementation is as follows : First use MOPs 5, 3, and 9 to address main memory and store the argument of the instruction in the address register. Also increment PC. Then, in the next cycle, use the contents of the address register to address memory (MOPS 10 and 3) to obtain the address to jump to and store it in the address register (MOP 9 again). Then in the final cycle, store the contents of the address register into PC (MOPs 10 and 6). Here is the complete micro program :

```
......: 00101010100000000000 (JIN)
......: 00100000110000000000 (JIN)
......: 10000100010000000000 (JIN)
```

We have omitted the exact addresses in the micro memory as well as the instruction code, since these are determined as usual by the first ones available.

There is a major inconvenience with this solution to subroutine calling, however. The caller must use the registers in order to load and store the return address. We can do better than that if we have a special instruction for subprogram calling. Most architectures contain such an instruction that is usually called JSR (jump to subroutine).

The JSR instruction takes care of both storing the return address, which it takes from the current value of PC, and jumping unconditionally to the subprogram. With such an instruction, our subprograms could be implemented like this :

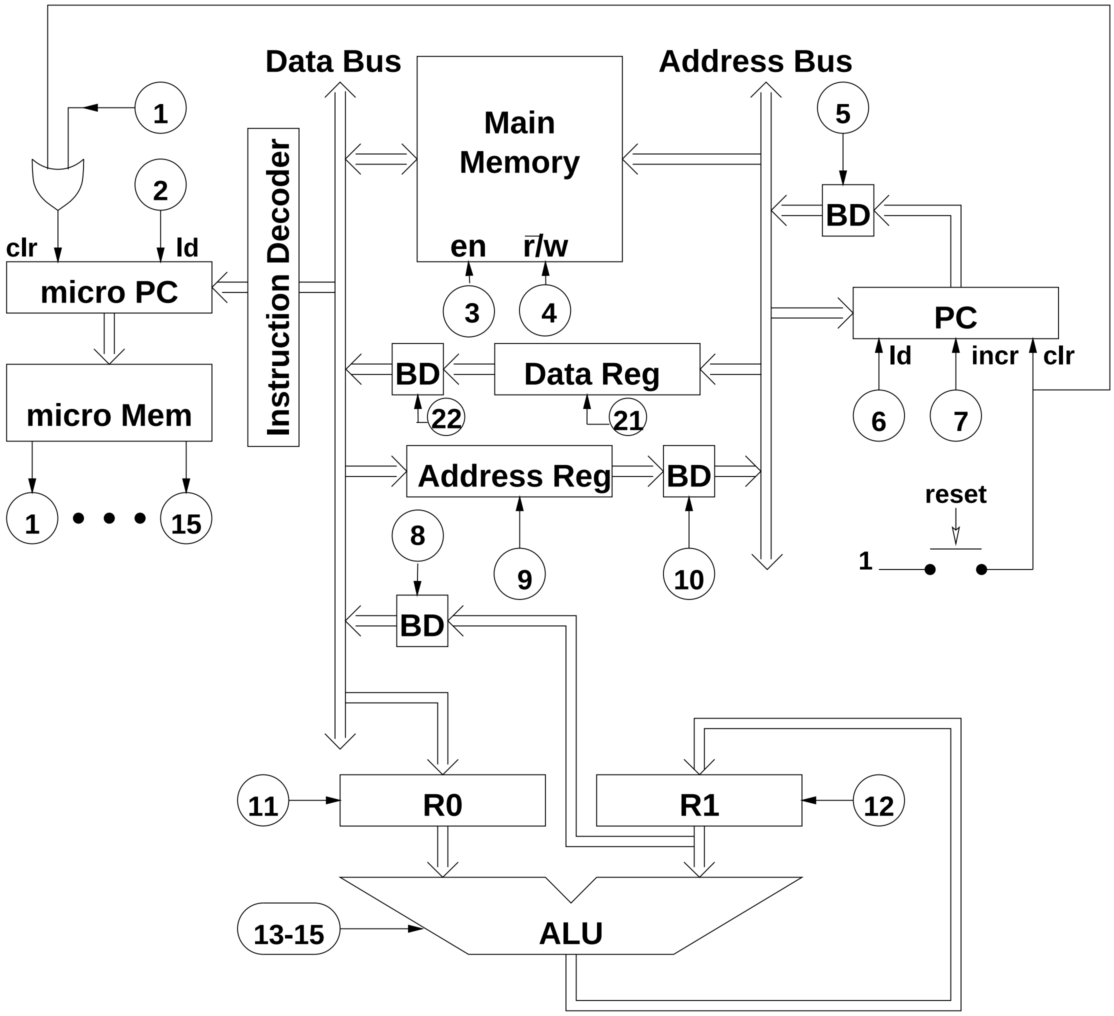
```
h-1:    0
h:      ...
        jin h-1

...

f:      ...
        jsr h-1
        ...

...

g: ...
        jsr h-1
        ...
```

The JSR instruction is not possible to implement with our current architecture. The reason is simple : we need to store the current value of PC in a cell in main memory. However, there is no data path from PC to the data bus. In order to implement JSR, we need to add such a data path.

Here is a complete description of what JSR must do :

- store the value of PC in memory at the address given as argument of the JSR instruction (h-1 in our example above)
- load the given address plus one into PC (h in our example)

To implement JSR, we add a new register, that we call the data register (for symmetry with address register). Here is our modified architecture :



Notice the new MOPs number 21 and 22 to load the data register and to communicate its contents to the data bus respecitively. To implement JSR, we use the following plan :

– In the first cycle, load the argument of the jsr instruction into the address register and increment PC
– Next store the contents of PC in the data register
– In cycles 3-6, write the contents of the address register into memory using the address contained in the data register
– Then, store the contents of the data register into PC
– Finally increment PC to obtain one plus the argument

Here is the microprogram :

```
......: 0010101010000000000000 (JSR)
......: 0000100000000000000010 (JSR)
......: 0000000001000000000001 (JSR)
......: 0001000001000000000001 (JSR)
......: 0011000001000000000001 (JSR)
```

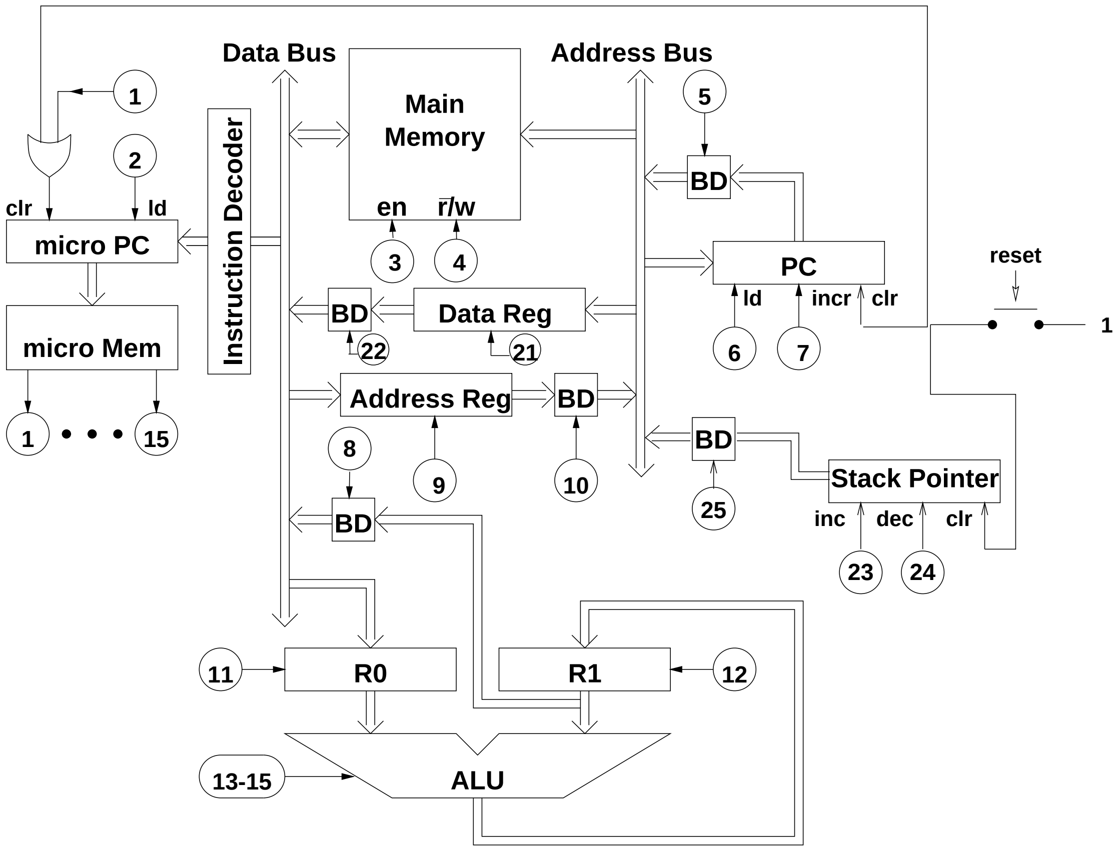
```
......: 0001000001000000000001 (JSR)
......: 0000010001000000000000 (JSR)
......: 1000001000000000000000 (JSR)
```

2.1. Recursion

The call/return protocol presented above does not allow for recursion, i.e. subroutines that call themselves directly or indirectly. The reason is simple : the second time a subroutine is called, the stored return address will be overwritten and lost forever. For a language such as Fortran, this is not a problem, since recursion is not allowed by the languages. But most modern languages allow recursion. So how can we improve our architecture to allow for recursion ? The answer is that we need a *stack*.

A stack is a sequence of cells in memory one end of which is pointed to by a register called the *stack pointer*. That end is called the *top* of the stack. Four different but equivalent conventions are used according to whether the top of the stack is the highest or the lowest address in memory, and according to whether the stack pointer points to the top element of the stack or to the first free position beyond the top element. Here, we use the convention used by most Unix systems : the stack top is the lowest address and the stack pointer points to the top element.

Here are the modifications to our architecture in order to support a stack :



The stack pointer can be cleared, incremented and decremented. Clearing the stack pointer at reset time is justified, since one can view the memory as circular. Address 0 in memory can be seen as lying just beyond the end of the memory. Three new MOPs are needed, 23 for incrementation, 24 for decrementation, and 25 for communicating the contents of the stack pointer to the address bus.

With this new architecture we can redefine the JSR instruction and add a new instruction RET (for return). Here are our subprograms with these new instructions :

```
h:      ...
ret

...

f:      ...
        jsr h
        ...

...

g: ...
        jsr h
        ...
```

Here is the new description of the modified JSR instruction :

- push PC on the stack
- load the address given as argument to the JSR instruction into PC

And here is the description of the RET instruction :

- load the top of the stack into PC
- pop the stack

Here is the plan for implementing the JSR instruction :

- in the first cycle, load the argument of the instruction into the address register and increment PC
- next load the contents of PC into the data register. At the same time, decrement the stack pointer
- in cycles 3-6 emit a write cycle to memory using the data register as data and the stack pointer as address
- finally, load the contents of the address register into PC

Here is the micro program for JSR :

```
......: 0010101010000000000000000 (JSR)
......: 0000100000000000000010010 (JSR)
......: 0000000000000000000001001 (JSR)
......: 0001000000000000000001001 (JSR)
......: 0011000000000000000001001 (JSR)
......: 0001000000000000000001001 (JSR)
......: 1000010001000000000000000 (JSR)
```

Here is the plan for implementing the RET instruction :

- in the first cycle, use the stack pointer to address memory, and store the top of the stack in the address register
- in the second and last cycle, load the value of the address register into PC and increment the stack pointer

Here is the micro program for RET :

```
......: 0010000010000000000000001 (RET)
......: 1000010001000000000000100 (RET)
```

3. Parameter passing

In the section on subprograms we saw how to use a stack for storing return values for procedures and functions. In this section, we extend the use of the stack to *parameter passing* for subprograms.

For this, we need two new instructions PUSH and POP. The PUSH instruction pushes the contents of R1 on top of the stack, and the POP instruction pops the top of the stack and puts it in R0. These two instructions are possible to implement without any further modifications to our architecture. Here is the microprogram for push :

```
......: 0000000000000000000000010 (PUSH)
......: 0001000100000000000000001 (PUSH)
......: 0011000100000000000000001 (PUSH)
......: 1001000100000000000000001 (PUSH)
```

And here is the microprogram for POP :

```
......: 1010000000100000000000101 (POP)
```

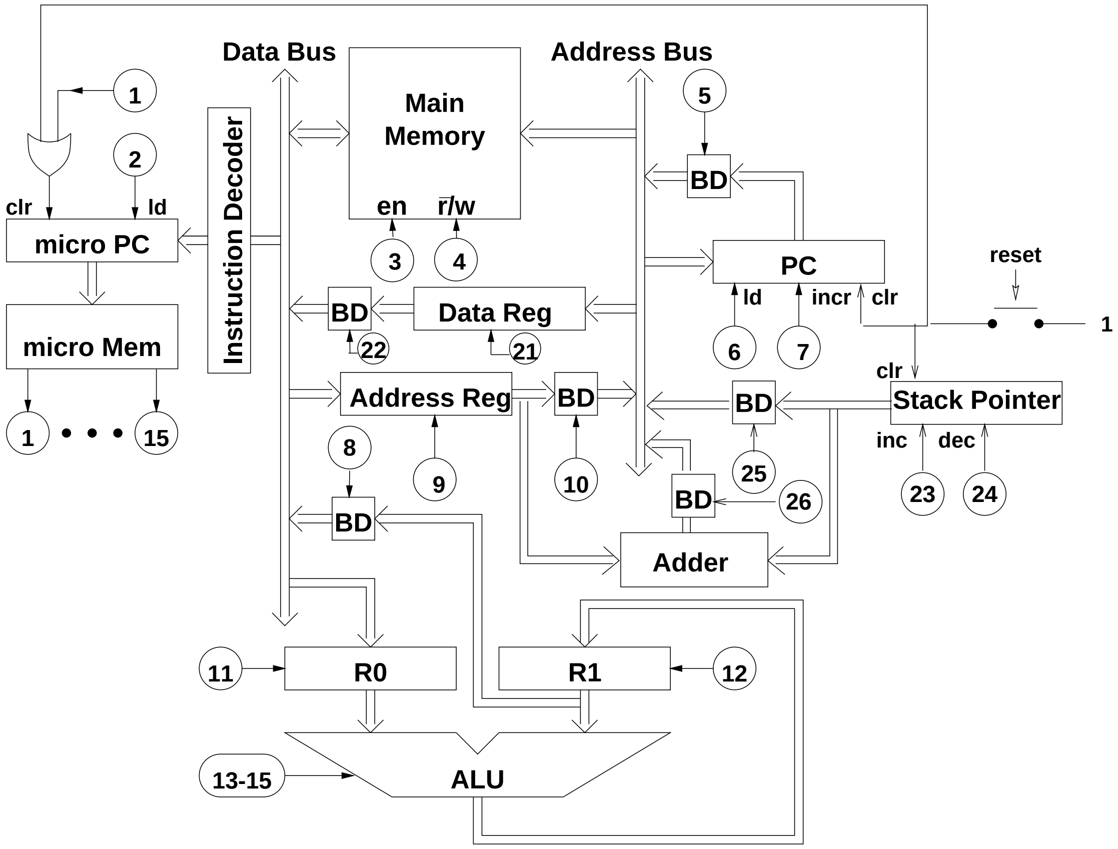
With these two instructions, we can make a more complete call/return protocol that contains rules for parameter passing as well. Here is an example of such a protocol. It pushes the arguments in reverse order (argument n first, argument 1 last). That is common for the C language since it allows the subprogram to find the first argument without knowing how many arguments have been passed :

```
Caller                                   Callee
------                                   ------

compute argument n in R1
push R1
...
compute argument 1 in R1
push R1
jsr callee
                                         compute return value in R0
       ret
pop n arguments off the stack
```

3.1. Access to arguments

While this way of passing paramenters works, it is not very practical for the callee to access those parameters. The only possibility currently available is to use the POP instruction. It would be more practical if we had instructions that could use addresses made up of the contents of the stack pointer plus a small constant. If we had such instructions, we could access argument number i by using address SP+(i+1), where SP is the contents of the stack pointer.

What we need is thus to be able to add a constant to the stack pointer. That constant will be stored as an argument to a new instruction. We can do this by storing the constant in the address register, and make it possible to add the contents of the address register and the stack pointer to form an address for main memory. The following modifications will allow that :



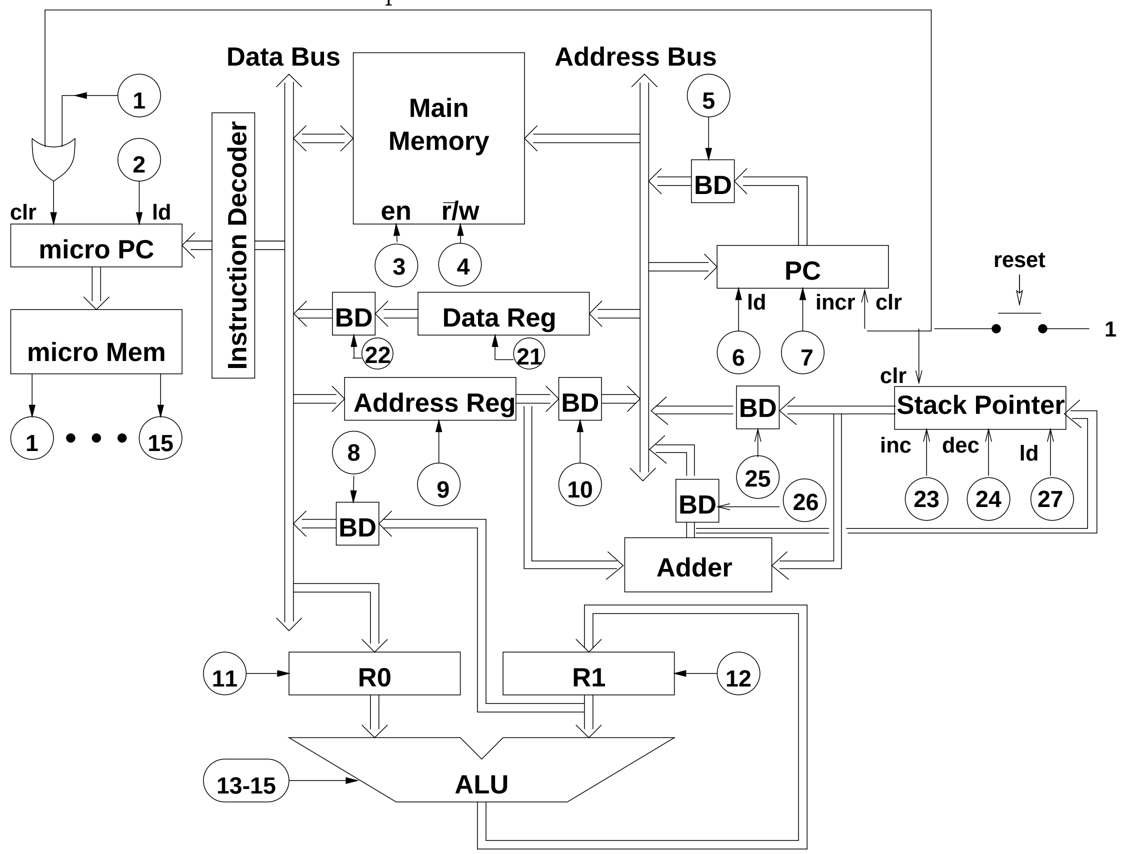
Notice the new MOP number 26 to communicate the sum to the address bus.

We can now write a new instruction LDS (for load from stack) that does what we want :

```
......: 00101010100000000000000000 (LDS)
......: 10100000001000000000000001 (LDS)
```

3.2. Popping the arguments

After the callee returns, the caller has to pop off hte arguments that it pushed before the call. Currently, the only possibility is to use POP as many times as there are arguments. An improvement would be to the possibility to add a constant to the stack pointer. For that, we can use the existing adder, but we need to be able to store the result into the stack pointer. This requires additional lines and an additional signal allowing a value to be loaded into the stack pointer. Here is the modified architecture :



Notice the new MOP number 27 for loading a value into the stack pointer. We can now write an instruction ADDSP that adds a constant value to the stack pointer. Here is its implementation :

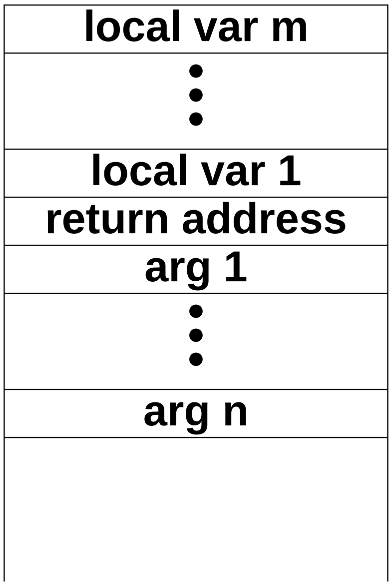
```
......: 001010101000000000000000000 (ADDSP)
......: 100000000000000000000000001 (ADDSP)
```

4. Local variables

In the section on parameter passing, we saw how to pass parameters from a caller to a callee and how to access those parameters during the execution of the callee.

To make the call/return protocol complete, we also need to show how to allocte variables local to a particular subprogram. These variables are allocated on the stack. Fortunately, we already have all the architectural support we need for local variables.

We will use the following *stack frame layout* :



Allocation of local variables must now be integrated into the call/return protocol as follows :

```
Caller                                 Callee
------                                 ------

compute argument n in R1
push R1
...
compute argument 1 in R1
push R1
jsr callee
                                       addsp -m (to allocate m vars)
                                       compute return value in R0
                                       addsp m (to deallocate vars)
       ret
```

```
addsp n (to pop arguments)
```