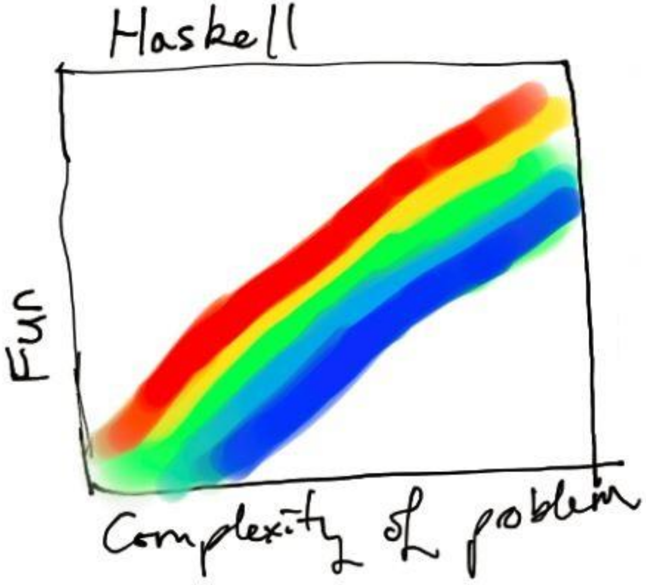
Integrated Circuits, Memory, Haskell



Recap & Agenda

Last week, we:

- Played with transistor circuits on our breadboards (made AND & OR gates)
- Learned about power supply and electricity

Today we will:

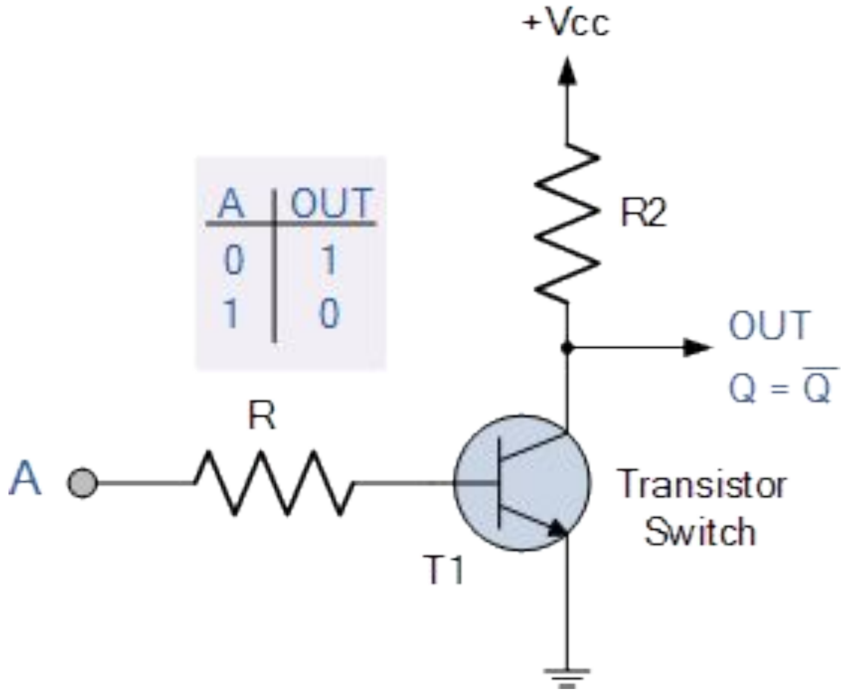
- Bump up a level of abstraction with our breadboards with integrated circuits
- Continue looking at hardware: memory/RAM
- Do some real haskell coding B)

First challenge!

Last week we made the AND & OR gates using transistors.

Now let’s build a NOT gate on your breadboard!

we’re going to need this gate for a later activity so please preserve it!

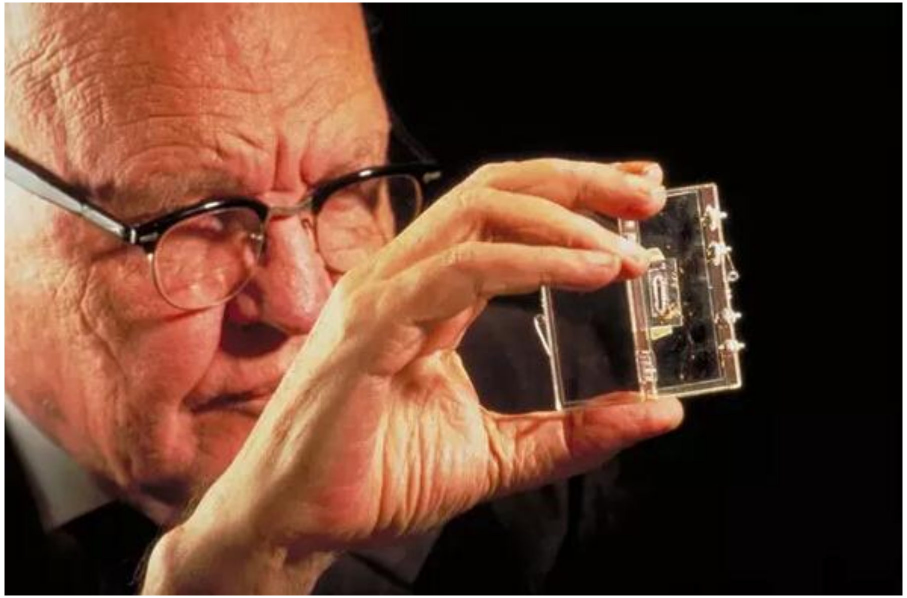


A	OUT
0	1
1	0

Integrated Circuits (ICs)

Programming using transistors is complicated! Let's bump up a level of abstraction to an *integrated circuit* (IC). ICs abstracts all the mess of wires and transistors into one small chip.

Integrated circuits were first successfully demonstrated by Jack Kilby in 1958. ICs revolutionized electronics, as ICs are more cost effective and more reliable than discrete components like transistors.



Jack Kilby, auctioning off his first IC (1958) in New York, expected to sell for $2 million.

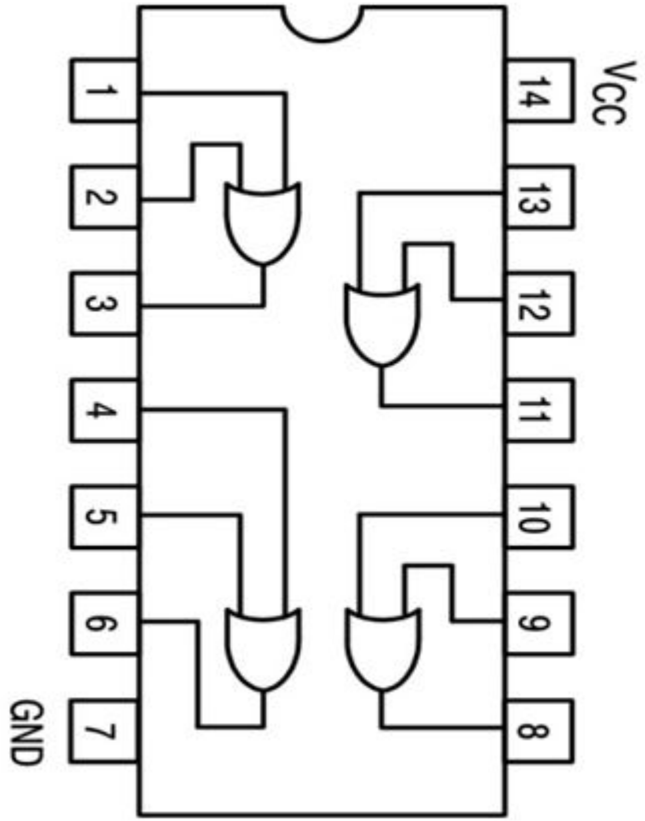
Now let's use one!

This is a "pinout diagram" for the chips we'll be using.

The top of the IC has a little divot to help you orient them.

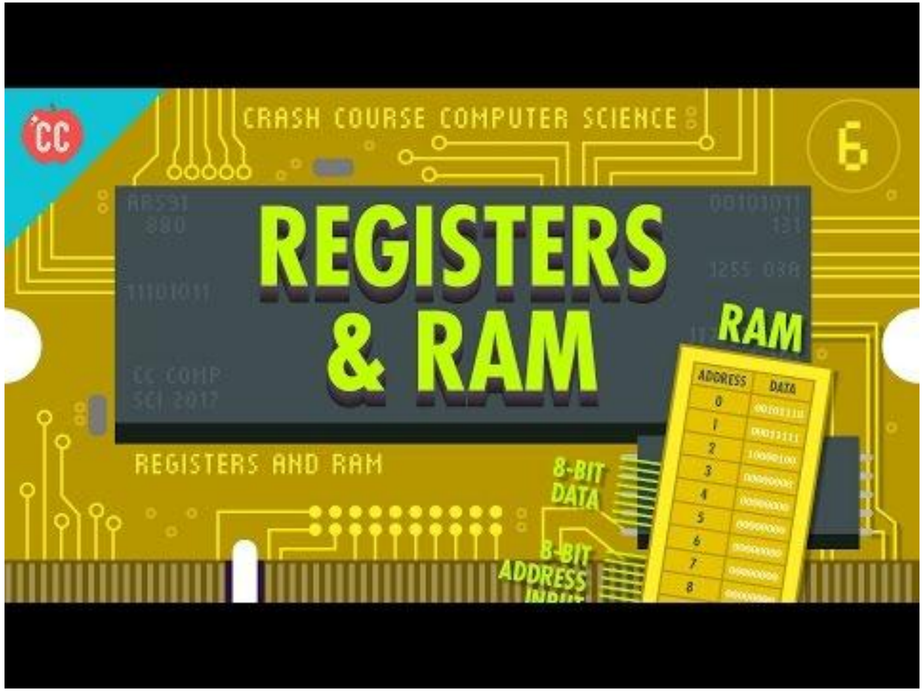
Make sure the VCC is connected to the positive and the GND is connected to the negative.

Test out the AND & OR ICs on your breadboard by observing their output using an LED! Don't try switches because we don't know how they work.



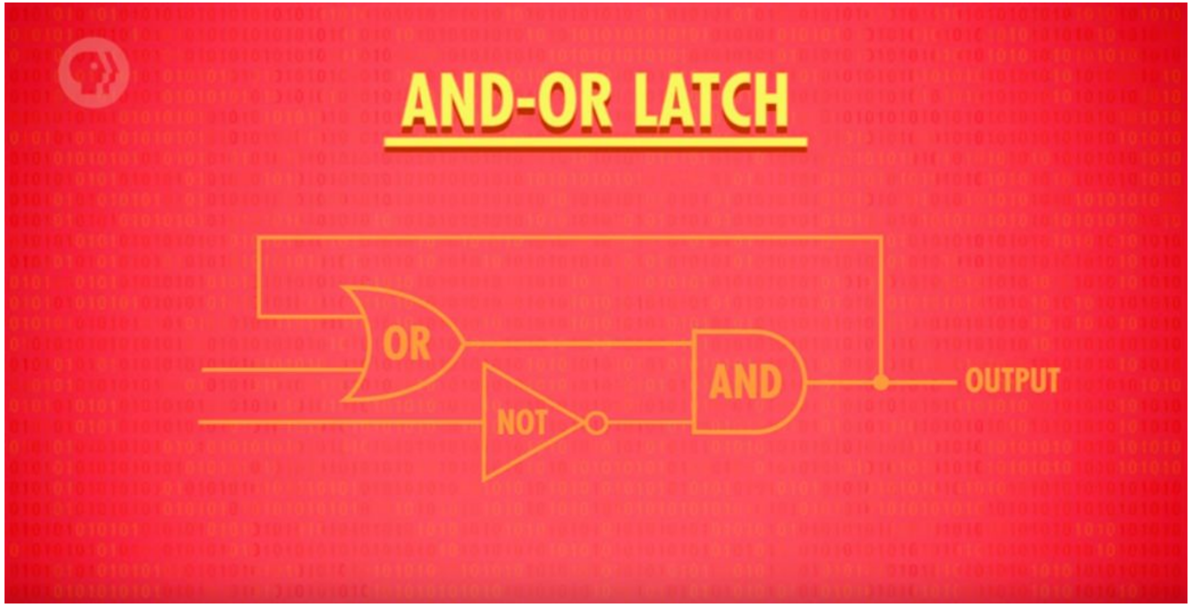
But what does this have to do with memory?

Take it away, Carrie-Anne!



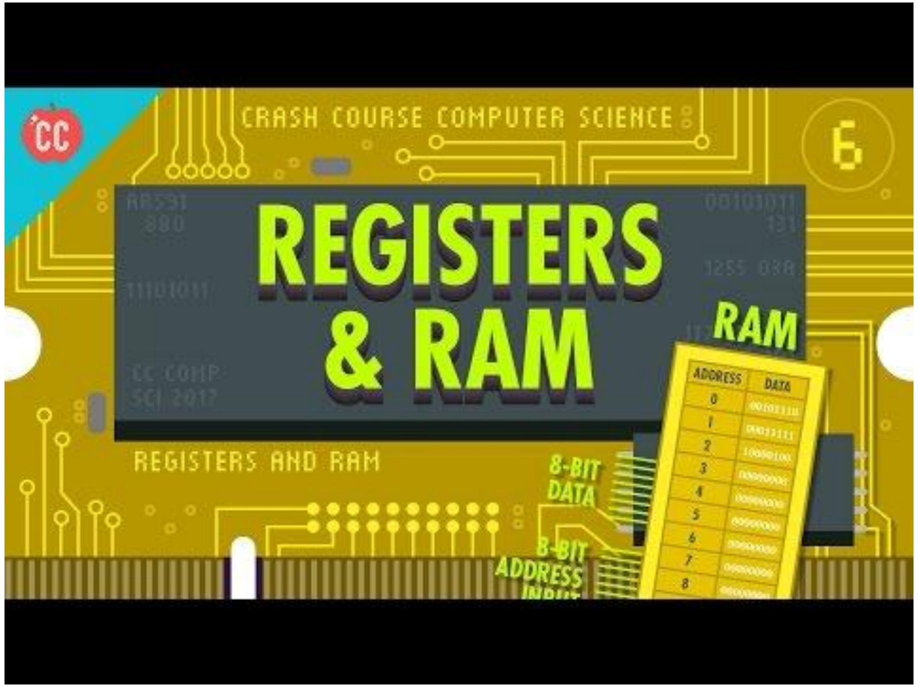
Remember all that?

Sure hope so! Let's build our own memory using all the circuits we've already made so far!

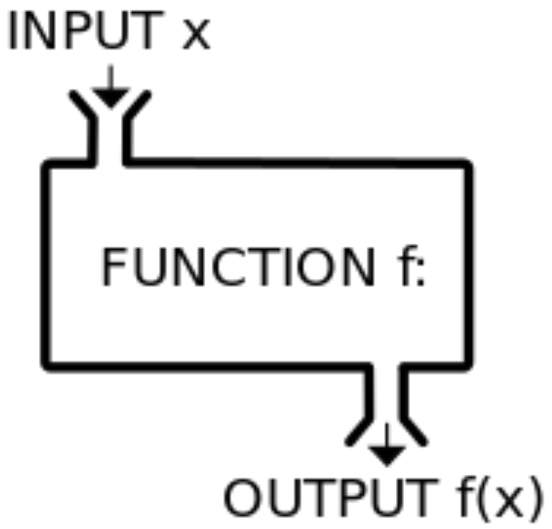


So how does your computer use AND OR latches?

Let's watch the rest of the CC video!



Moving to software: Haskell!!



Let’s try Haskell, the programming language we’re going to use to generate our website! Woohoo!

Haskell is a high level *purely functional* programing language. This means it only uses functions.

Functions are like a machine that performs some operation on the input to compute an output.

Ex. 1 + 1 = 2. (+) is a function, that takes 2 inputs and computes its sum. *1 + 1 will always equal 2.*

Haskell 101: definitions

Before we dive into Haskell, let’s look at its *syntax* and its meaning.

There are only 2 things you can write in Haskell: *definitions* and *expressions*.

Definitions give a name to an **expression**.

Ex: `age = 25`

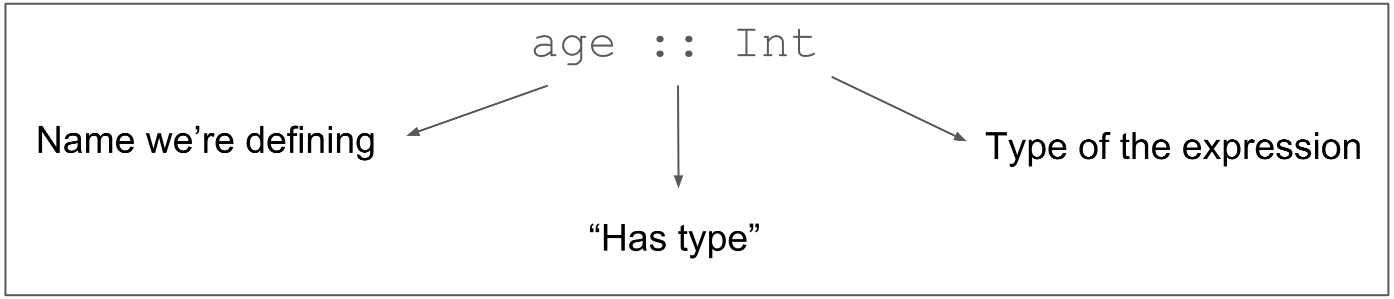
Here, we’re making our own definition for the word *age* which is 25. Age and 25 now mean the same thing.

Expressions are like phrases that use math. They can contain numbers, variables, functions. Ex. 3 + 5 `div` 4 (this is valid haskell code and would output 2)

Types types types

This is called a *type signature*, as it tells us the type of `age`.

Since everything has a type, there are *a lot* of types in Haskell. What are some types you remember from tryhaskell.org?



```
age :: Int
```

[Lists]

One really important type is the *list* type.

Lists are denoted by square brackets and the values in the lists are separated by commas. They look like this:

```
participants :: [String]

participants = [“Ani”, “Tareq”, “Kelly”, “Caiti”, “Callan”]
```

Lists can store several *elements of the same type.* Ex. this list can only have strings, so no numbers or anything else.

Type Signatures of Functions

Like previously mentioned, everything has a type, including functions. Let’s look at one now.

```
addThree :: Int -> Int -> Int -> Int

addThree x y z = x + y + z
```

Here is a function that takes 3 numbers and adds them together. In order to use this function, we must respect two things: the number of inputs and the types of each input.

In a type signature, the rightmost type **always** is the output. Arrows separate each input, so everything that comes before is an input.

Finally! We’re ready to CODE!

With your computers, let’s all head to code.world to start!!