Before Corona interrupted us, recall we began discussing machine languages of computers. The machine language of a computer is the language that is built into the hardware of the CPU. A PC and a Mac have completely different machine languages. When we program in Logo we use an *interpreter* to convert the Logo program (a high-level language) into the computer’s machine language. Any interpreter (sometimes called a compiler) is translator between a high-level language and a machine language. Each compiler/interpreter is specific to a particular pair of languages, one high level and one machine language. This is why the Logo interpreter for a PC will not work for a Mac (the different operating systems is not the problem). There are dozens, perhaps hundreds, of high-level languages including Logo, Java, C, Python, Perl, Lisp, Cobol, Fortran, Basic, etc. Each one needs an interpreter to translate it to the native machine language of the computer on which it will execute.

We started to learn a particular machine language for a made-up computer that has many of the features of common computers today. This computer uses a *fetch cycle* to execute a machine language program. The program sits in memory with each instruction taking two bytes of memory (i.e., 16 bits). One instruction at a time is moved by the CPU from memory to the CPU where it is stored in a location (memory in the CPU is called a register) called the Instruction Register. From there it is sent to the Control Unit of the CPU which decides how to execute the instruction. Then the ALU (Arithmetic and Logic Unit) and RAM, are told what to do by the Control Unit and the instruction is executed. After that, the next instruction is fetched.

Besides the Instruction Register, there are other important memory location in the CPU:

Program Counter: holds the location in RAM of the next instruction.

Accumulator: holds the results of the ALU.

Memory Address Register: holds the location of RAM you wish to read from or write to.

Memory Data Register: holds the value of data you want to write to RAM, or which was read from RAM. Note that, this data includes the instructions themselves during the fetch cycle. Unlike Logo commands, machine language instructions convert directly to hardware actions in the CPU and RAM.

The machine language we discuss has the following 12 instructions, each of which has one parameter X, except for End.

In X Reads data from the keyboard into location X of RAM.

Out X Prints data in location X of RAM onto the screen.

Load X Copies data from location X in RAM to the accumulator in CPU.

Store X Copies data from accumulator into location X in RAM.

Add X Adds data in location X of RAM to the accumulator.

Sub X Subtract data in location X of RAM from the accumulator.

Mul X Multiply data in location X of RAM by the accumulator.

Div X Divide data in the accumulator by location X of RAM.

Br X I will describe these three later on.

BrZ X

BrG X

End Stops the program and sends control back to the operating system.

Let’s concentrate on the first eight instructions and End.

The following program in Logo adds two numbers and prints the result.

Make “x read

Make “y read

Print (+ :x :y)

End

Its machine language equivalent is shown here. Note the comments after the semicolon on each command.

In 100 ; location 100 stores the value x

In 200; location 200 stores the value y

Load 100

Add 200

Store 300 ; The previous three instructions add x to y and stores the value in location 300

; note that the accumulator currently holds x+y, as does location 300 in RAM

Out 300; prints x+y

End

The locations 100, 200, and 300 were chosen arbitrarily. Each number is stored in 4 bytes, so we could have just as well used 100, 104, and 108 instead, as long as each location was at least 4 bytes away from the others. Note also, that the instructions themselves are stored in RAM, and their locations are determined by the operating system, as long they are stored in different places than the data, i.e. away from 100, 200, and 300. Instruction is stored in two bytes then the program might be stored starting at location 1000.

For example,

1000 In 100

1002 In 200

1004 Load 100

1006 Add 200

1008 Store 300

1010 Out 300

1012 End

Furthermore, the instructions themselves are actually stored in binary, the details of how we will discuss later. Indeed, machine language programs are pure binary. Technically, when we write the programs with words like Load and numbers like 100, it is called an Assembly language program. An *assembler* makes the final translation from Assembly language to machine language. Later on, we will learn how to change from Assembler into full machine language binary. For now, all our programs will use assembly language.

Here is a Logo language program that computers the average of two numbers.

Make “x read

Make “y read

Print (/ (+ :x :y) 2)

End

Its assembly language equivalent is shown here. Note the comments after the semicolon on each command.

In 100 ; location 100 stores the value x

In 200; location 200 stores the value y

Load 100

Add 200

Div #2; This means divide the accumulator by 2.

; It is different than Div 2

; which would mean to divide the accumulator by the number in location 2 of RAM

Store 300

; The previous three instructions compute (x+y)/2 and stores the value in location 300

Out 300; prints (x+y)/2

End

This program illustrates a feature of machine language called an address mode. An address mode is a way to interpret the parameter. Our machine language has two address modes: memory (no hash tag precedes number), and immediate (hash tag precedes the number). For example, Div 2 is memory address mode, and it means to divide the accumulator by the number stored in location 2 in RAM, while Div #2 is immediate address mode and it means to divide the accumulator by the number 2 itself. Every machine language has address modes to make programs easier to write.

Our machine language also has two address modes indicated by the presence of a hashtag, which indicates how X is to be interpreted. Some commands use only one address mode and some have both. Here are the commands that have the second address mode:

Out #X Prints the value X to the screen.

Load #X Copies the value X into the accumulator.

Add #X Adds the value X to the accumulator.

Sub #X Subtracts the value X from the accumulator.

Mul #X Multiplies the accumulator by the value X.

Div #X Divides the accumulator by the value X.

So far, all we can do is calculations, but no if-statements and no loops. The remaining three instructions are all we need to be able to simulate if-statements and loops. Let’s go through them one by one:

Br X This branches to line X.

BrZ X This branches to line X when the accumulator holds zero, else it executes the next line

BrG X This branches to line X when the accumulator is greater than zero, else the next line.

Let’s use these new instructions to simulate an if-statement.

The following Logo program prints the larger of two numbers:

make “x read

make “y read

if (> : x :y) then print :x else print :y

The idea is to subtract one number from the other and store their difference in the accumulator. Then we can branch to one line if the difference is greater than zero, and continue to the next line otherwise. To make sure e don’t just fall into the branch line in the case when we continue with the next line, we jump around it. For example, here is an assembly language program that is equivalent:

1000 In 100 ; X is stored in location 100

1002 in 200 ; Y is stored in location 200

1004 Load 100

1006 Sub 200 ; X-Y is stored in the accumulator

1008 BrG 1014 If X>Y then go to line 1014

1010 Out 200 ; Otherwise, this line is reached when X <= Y, and Y is printed

1012 Br 1016 ; Skip over line 1014

1014 Out 100 ; X is printed because X >Y

1016 End

The “branch” instruction is a very low-level way of simulating if-statements and loops. BrZ is used to check when things are equal, BrG is used to check when one value is bigger than another, and Br is used to make sure we flow around the choices correctly.

Before we do a more complicated program using loops, we will take some time to understand how to convert assembly language to machine language.

We need to convert everything to binary: the name of the instruction, the address mode, and the parameter. The easier parts are the instruction name and address mode.

The instruction name is often called the *op-code*. Recall that there are 12 op-codes: In, Out, Load, Store, Add, Sub, Mul, Div, Br, BrZ, BrG, End. We must give each of these a binary pattern. Since three bits results in 2^3 = 8 patterns, we will need 4 bits to manage. The assignment of patterns to op-codes is shown below:

In 0000 Add 0100 Br 1000

Out 0001 Sub 0101 BrZ 1001

Load 0010 Mul 0110 BrG 1010

Store 0011 Div 0111 End 1011

Notice how the 12 patterns go in numerical order from 0 through 11 inclusive. The order is the same as the order that I originally presented the instructions, and for no particular other reason.

Now we need to figure out how to represent the address mode, and the parameter.

The address mode is either memory or immediate, which is distinguished by either no hashtag or yes hashtag. We will represent no hashtag as a zero, and yes hashtag as a one.

Finally, we need to turn the parameter into binary. If the parameter is using the immediate address mode, we just turn the decimal number to binary. The tricky part is when the address mode is memory.

This part is tricky and I will try to motivate it carefully. So far, we have 4 bits for the instruction (op-code), and one for the address mode, for a total of 5 bits. Since every instruction is supposed to fit into 16 bits (2 bytes), we are left 11 bits for the parameter. That means we have 2^11 patterns for both an immediate number and a memory location. However, the problem is that here are way more than 2^11 = 2048 locations in memory – there are, in fact, billions. The solution is to not try to identify a location directly, but instead we specify a memory location by how far it is from the current location of the op-code. Since the op-code’s location is stored in the program counter register, the computer can find the correct memory location by adding the 11-bit value to the current program counter. This effectively gives us 2^10 = 1024 locations forward from the op-code and 2^10 locations backwards, for a total of 2^11 = 2048 options. Almost no program uses more than 2048 locations, so we are safe.

If you are wondering why we don’t just make the op-code larger and then get more bits and options – we can do that too, and indeed, a modern computer has op-codes of 4 bytes – or 32 bits, with typically 6 for the op-code, 2 for the address mode, and 24 for the parameter.

Let’s look back at the example assembly language program that calculates the average of two numbers and try to convert it to machine language.

300 In 100 ; X is stored in location 100

302 in 200 ; Y is stored in location 200

304 Load 100

306 Add 200

308 Div #2

310 Store 400

312 End

**In 100** turns into:

0000 0 10011001000

Let’s break it up. The pattern for the op-code **In** is 0000.

The address mode is memory – no hashtag – so we use 0.

The parameter 100 is stored by recording how far it is from the current location of 300 to get to 100, namely -200. There are many ways to represent negative numbers in computers but let’s stick to the simplest, which uses a leading 0 to mean a positive number and a leading 1 to indicate a negative number. Thus, 10011001000 means – 200. The leading (leftmost) one indicates negative, and the rest is the usual binary representation of powers of two: namely, 8 + 64 + 128 = 200, reading it off from right to left.

Similarly, **In 200** becomes: 0000 0 10001100110

The last 11 bits, in this case, represents the number –102, the distance from 302 to 200.

**Div #2** becomes: 0111 1 00000000010,

**Store 400** becomes: 0011 0 00001011010

The op-code is 0011, the address mode is 0, and the parameter represents 90, the distance from 310 to 400.

**End** is simply: 1011 0 00000000000. With no address mode or parameter, we just use zeros.

Try to translate the rest of the lines yourselves for practice.

Then (with your group) try to do [Quiz1.](http://web.stonehill.edu/compsci/How-Computers-Work/Quiz1.html) Email me if you need help or hints.

Note that if we had more instructions, and/or more address modes, and/or a larger memory range to reference, we would have to use more bits to represent an instruction. For example, if we had more than 16 instructions, we would need 5 bits for the opcode. If we had three address modes, we would need two bits for that. Modern computers typically try to represent an instruction using 32 bits split between the opcode, the address modes, and the memory range.

Next, we will consider more complicated programs that need more planning and uses loops, and if-statements and variables. After this, you should try to complete [Quiz2](http://web.stonehill.edu/compsci/How-Computers-Work/quiz2.html).

End of Week 1

Let’s find the largest of three numbers. Here is a plan.

1. Read in three numbers.
2. Compare the first two numbers and store the larger somewhere new.
3. Compare the third number to the larger number you stored.
4. Print the larger of the two from step 3.

Since we may need lots of branches, we will just label the lines for now with letters and number all the lines at the end.

Here we go:

In 100

In 200

In 300

; This reads the 3 numbers A, B and C into locations 100, 200, and 300 respectively

Load 100

Sub 200 ; Subtracts B from A in the accumulator

BrG M ; Line M will store A in location 400 if A > B

; Otherwise we store B in 400

Load 200

Store 400

Br N ; skip line M to continue to test C against 400

M: Load 100

Store 400

N: Load 300 ; Now we compare C to the larger of A and B in 400

 Sub 400

 BrG P

 Out 400 ; The larger of A and B is the largest of all.

 Br Z

P: Out 300 ; C is the largest

Z: End

Now we can renumber the entire program consecutively staying away from any numbers between 100 and 400, and replacing the labels with real memory locations.

500 In 100

502 In 200

504 In 300

; This reads the 3 numbers A, B and C into locations 100, 200, and 300 respectively

506 Load 100

508 Sub 200 ; Subtracts B from A in the accumulator

510 BrG 518 ; Line 518 will store A in location 400 if A > B

; Otherwise we store B in 400

512 Load 200

514 Store 400

516 Br 522 ; skip line 518 to continue to test C against 400

 518 Load 100

520 Store 400

 522 Load 300 ; Now we compare C to the larger of A and B in 400

 524 Sub 400

 526 BrG 532

 528 Out 400 ; The larger of A and B is the largest of all.

 530 Br 534

 532 Out 300 ; C is the largest

 534 End

Here is a more complicated assembly program to add up n numbers. The logo version is here:

make “sum 0

make “n read ; this is the number of numbers you are adding

repeat :n [ make “nextnumber read make “sum (+ :sum :nextnumber) ]

print :sum

Our plan is to have a place to store n (location 100), a place for the next number (200), and a place for sum (300). We also need to countdown from n to zero. We don’t have so many branches – only one for the loop so we will just number the instructions as we go along. Here we go:

1000 Load #0

1002 Store 300 ; These lines store 0 in sum

1004 In 100 ; This gets n

; The repeat loop starts here

1006 In 200 ; get the next number

1008 Load 300

1010 Add 200

1012 Store 300 ; The last three lines add the next number to sum.

1014 Load 100

1016 Sub #1

1018 Store 100 ; The last three lines subtract one from n.

; If n is zero then we are done

1020 Load 100

1022 BrZ 1026

1024 Br 1006

1026 Out 300

1028 End

Now you should try to complete [Quiz2](http://web.stonehill.edu/compsci/How-Computers-Work/quiz2.html). Follow the plan in the assignment to make sure you don’t get confused.

End of Week 2

Now we are going to go deeper into the hardware of the machine to see what is inside the parts of the CPU, the ALU, and the registers (like the accumulator). How do we build these smart little gadgets from electronic components?

This is a deep subject and part of a much larger topic of study in computer engineering called digital logic design. We will touch on this subject and show how to

1. Build a circuit that does addition in binary.
2. Build a circuit that stores bits.

There are plenty of other parts inside the CPU, but these two circuits are very representative of the whole picture.

Let’s begin with binary addition and review how to add like you learned in 2nd grade, but this time – it is in binary. With binary numbers you have to carry when you get to 2 instead of 10. Indeed, in binary 2 is written like 10.

Example:

10001011 + (139 in decimal)

00111101 (61 in decimal)

We go right to left column by column just like second grade.

1+1 = 2 = 10. That is, we have to carry a 1 over to the 2’s column and leave a 0 in the result.

In the 2’s column we have 1 + 0 + the carried 1 from 1’s column = 10 (2)

In the 4’s column we have 0 + 1 + the carried 1 from the 2’s column = 10 (2)

In the 8’s column we have 1 + 1 + the carried 1 from the 4’s column = 11 (3)

This continues, until we get:

11001000 (200 in decimal)

Here is a chart that shows all the rules:

TopBit BottomBit Carry Result

0 0 0 0

0 1 0 1

1 0 0 1

1 1 1 0

This chart is sometimes called a *truth table* and we can summarize the chart with some formulas, which will then be turned into circuits!

Carry is 1 whenever the TopBit is 1 ***and*** the BottomBit is 1.

Result is 1 whenever:

 TopBit is ***not*** 1 ***and*** BottomBit is 1 ***or***

 TopBit is 1 ***and*** BottomBit is ***not*** 1

I wrote the formulas in that way because you can buy electronic pieces called gates that let you calculate ***and, or, not.***

So, abbreviating TopBit as TB and BottomBit as BB, we get:

Carry = TB ***and*** BB.

Result = (***not***(TB) ***and*** BB) ***or*** (TB ***and*** ***not***(BB))

These kinds of formulas are called Boolean formulas – after George Boole, and English 19th century mathematician that invented Boolean algebra with these operations and formulas.

The operators, ***and, or, not*** are just like the logical commands in Logo, where you think of 0 as false and 1 as true.

Here are the truth table charts for each of them:

A B A ***and*** B A ***or*** B ***not*** A ***not*** B

0 0 0 0 1 1

0 1 0 1 1 0

1 0 0 1 0 1

1 1 1 1 0 0

Using what we do for Result and Carry, any truth table can be written as a Boolean formula using combinations of ***and, or, not***.

For example, consider the following truth table:

A B C Mystery

0 0 0 0

0 0 1 1

0 1 0 0

0 1 1 1

1 0 0 0

1 0 1 0

1 1 0 0

1 1 1 0

It doesn’t matter what Mystery is or whether it does anything interesting like adding numbers, but regardless, we can write a formula for Mystery.

Mystery = (***not****(*A) ***and not***(B) ***and*** C) ***or*** (***not***(A) ***and*** B ***and*** C).

If you are still not sure how to take a truth table and turn it into a Boolean formula, then email me – and I will explain it again with more examples.

Indeed, any truth table whatsoever can be turned into a Boolean formula, and you will see soon that it can then be turned into an electronic circuit. When we move to an electric circuit, the zeros are low voltages (almost 0) and the ones are higher (5 millivolts). And these voltages move around through electronic components called AND, OR, and NOT *gates* that simulate these Boolean functions.

Watch this 15 minute YouTube video:

<https://www.youtube.com/watch?v=fw-N9P38mi4>

It will show you what gates look like and how they are built out of transistors. Gates get smaller and faster each year. Nowadays, you can fit over 100,000 gates in a fingernail, and it takes electricity about a 10 billionth of a second to get through a gate! That means you can go through 10 billion gates in a second.

This is the reason why computers can tick their clocks so quickly. Every machine instruction is composed of various hardware actions, each of which must finish before the next can start. For example, an add instruction must be fetched from RAM and then it moves data from the accumulator to the ALU, it moves memory from RAM to the ALU, it adds the two together, and then stores the answer back in the accumulator. The computer ticks its clock for every one of these hardware actions. If every hardware action can be done using a sequence of at most 5 gates, then the clock can tick 2 billion times a second, because you can get through 10 billion gates in a second.

It turns out that you could also build gates out of water channels sewer grates and covers! With enough space, water, shovels, and slaves, the Egyptians could have built a computer out on the desert in 2000 B.C.E. This computer, however, would have its clock tick every 5 minutes, and would be several football fields large. Adding two numbers would involve a week’s time of hundreds of slaves and millions of gallons of water… nothing remotely practical in ancient times, when any intelligent could do it on piece of slate in seconds.

Nobody ever thought of inventing all the mathematical advances of Boolean algebra until the 1890s, and nobody considered applying it to the engineering of computers until the 1940s, when the hardware was fast enough to make it interesting. One of the most beautiful parts of computer science is this interplay between mathematical advances and engineering advances – where one area often leaps frog ahead of the other waiting for the other to catch up.

Once you understand gates, you can put them together to implement the formulas we built above.

Now watch this video to see how it is done for an exclusive Or.

<https://www.youtube.com/watch?v=BnB2m1nXZ84>

An exclusive Or circuit takes two inputs and outputs a 1 whenever exactly one of the inputs is 1.

So the chart is:

A B XOR

0 0 0

0 1 1

1 0 1

1 1 0

The formula for XOR is (***not*** A ***and*** B) ***or*** (A ***and not*** B). Make sure you understand why or go back reread the last section. This is the same as the Result circuit a page or two back.

Now go look at the 11:40 of the video <https://www.youtube.com/watch?v=BnB2m1nXZ84> and you can see the gates built to match the XOR formula.

You should be able to take a description of any circuit, make its truth table, construct its Boolean formula, and then build its circuit of gates and wires. Make sure you can before moving on.

Let’s now return to our circuit that does addition. The one we discussed had only two inputs and two outputs:

Inputs: TopBit, BottomBit

Outputs: Carry and Result.

This is not exactly what happens in each column of addition when we add numbers. In actuality, we have three inputs and two outputs, because the Carry of one column moves over to the next column. This gives:

Inputs: TopBit, BottomBit, Carry\_in

Outputs, Carry\_Out, Result

This circuit has three inputs and two outputs. Each output needs its own formula. Each formula needs its own gates. It is such an important circuit in the ALU, that it is called a “full-adder”. Quiz 3 asks you to design the circuit for a full adder. To get you started, here is the truth table below. Notice that there are now 8 rows in the table because there are 2^3 patterns of bits for the inputs.

TB BB Carry\_In Carry\_Out Result

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

This is the table that represents how we do binary addition on two binary numbers.

How much time does it take to run electricity though this circuit and get an answer? It depends on the number of levels of gates. The electricity takes a 10 billionth of second to get through a gate, and the speed of light (more or less) to get through the wires. The electricity moves in parallel through gates at the same level, so the time it takes to get through any circuit is the number of levels of gates.

For example, consider this circuit, where the ***and/or*** gates are able to take more than two inputs. An and-gate with multiple inputs fires a 1 output only when all its inputs are 1. A multiple input or-gate fires a 1 only when at least one of its inputs is 1. The inputs of this circuit are the s0, s1, I0, I1, I2, and I3. The output is Y. The electricity moves first through the not-gates, then the and-gates, and finally the or-gate. So it takes three gate delays to make it through the entire circuit, that is, 3 \* 10 ^(-10) seconds, or .3 billionths of a second.



A full adder has two outputs, and if you’re clever about designing the circuit, you can get the result out in three levels, and the carry-out in two levels. In particular, the carry-out = (A and B) or (A and CarryIn) or (B and CarryIn), another of way of saying that there is a carry when any two of the three inputs are 1.

Now let’s look at a really important circuit in the ALU of the computer. A full-adder adds up only one column of bits. If you wanted to add two 32-bit numbers – the size of numbers in modern computers – you would need to have 32 full adders and have the carry-out of one feed the carry-in of the next. See this diagram for a 4-bit version that leaves out the gates inside the individual full-adders.



The A and B values are the input bits and the S values are the result bits.

Now let’s analyze how much time it takes for a 32-bit ripple carry adder to do its job. Recall that each full adder can be built with 3 levels of gates from the inputs to the result, and two levels from the inputs to the carryout. So it takes 3 gate delays for S0 to be ready, and two for C0. S1 needs to wait for C0, so S1 needs 5 gate delays, and C1 needs 4. Similarly, S2 needs 7 and C2 needs 6. The circuit for 32 bits is not finished until S31 is done. And S31 needs an amazing 65 gate delays! That is, 31(2) + 3.

The relationship between the clock speed of a computer and the circuits inside is described next.

A ripple carry adder is freakishly slow - so slow that it is rarely used in a modern computer. There are better ways to design addition circuits with much smaller delays. If we did use a ripple carry adder, it would surely be the slowest circuit used by the computer when executing instructions. That means that the clock of the computer could not tick any faster than the time need to wait for the ripple carry adder to finish. In other words, the time of one tick of the clock in a computer that uses a ripple carry adder can be no faster that 65 \* 10^(-10) seconds per tick, or 6.5 billionths of a second per tick. The clock speed of such a computer would be 1 tick per 1 billion /6.5 = about 154 million ticks per seconds, or 154 MHz. That is like the computers of 20 years ago.

There is one more cool circuit we will discuss that you can experiment and play with at this link:

<http://www.falstad.com/circuit/e-7segdecoder.html>

This circuit has 4 inputs and 7 outputs. It is a common circuit in all sorts of electronic devices that display digits, including DVRs, clocks, cell-phones, etc.

The four binary inputs represent a single digit.

0000 is 0

0001 is 1

0010 is 2

0011 is 3

0100 is 4

0101 is 5

0110 is 6

0111 is 7

1000 is 8

1001 is 9

The rest of the bit patterns never occur.

The outputs are 7 LED lights, a through g, that light up to show the appropriate digit. One means light up and 0 means light off.



Let’s make the truth table for the light labeled c.

Inputs c

0000 1

0001 1

0010 0

0011 1

0100 1

0101 1

0110 1

0111 1

1000 1

1001 1

Make sure you can finish the truth table for the other lights. Once you have the truth tables, you would make formulas, and then gates. When you are sure you get it, look again at <http://www.falstad.com/circuit/e-7segdecoder.html> and see if you can follow the circuit through.

When you think you can do all this, try [Quiz3](http://web.stonehill.edu/compsci/How-Computers-Work/quiz3.html).

End of Week 3

To finish off this subject of circuits and the deepest part of the computer we study, let’s look at one last circuit whose inventor was the winner of a Nobel prize in the 1940s. This circuit shows how to use gates to store bits in memory.

The circuits we have looked at until now have bits coming in and out, and processing going on, but no storage – just flow. When you add numbers you need to put them somewhere. What can we do to store them?

The trick is to have the circuit feedback to itself. Take a look at this circuit which is sometimes called a D-latch. It allows you to store 0’s and 1’s. The bit you store will stay there as long as electricity remains on. This is the typical kind of storage used in CPU registers and RAM. You put 8 of these together and you get a byte. You put billions of those together and you have a RAM, but underneath the hood it’s just lots of gates. Let’s analyze this circuit and see what is going on.

You will notice the usual inputs and outputs, except that the outputs feedback to the inputs, and there is a “Clk” input. The “Clk” input comes from the computer’s clock and is used to synchronize the memory. The circuit is open for business only when the clock ticks. That helps the computer make sure that no stray bits get stored by accident. The bits get stored only when the computer ticks, (GHz - billions of times a second nowadays).

You will also notice that the Or gates have little dots at their ends. That means that they have Not gates there. So the Or gates in this picture are really Nor gates. A Nor gate fires 1 exactly when both its inputs are zero, otherwise it fires zero. Let’s make a chart.



This a tricky circuit to understand. Let’s look at it carefully. The chart has inputs for D and Clk. It has inputs and outputs for Q(t) and Q(t)’. The Q(t) and Q(t)’ always end up opposite each other. That is, they are either 1/0 or 0/1. You will see why in a minute. The computer looks at Q(t) to what bit the circuit is storing.

Notice that the D sends opposite signals to the two And gates. When D is 1, the top gets 0 and the bottom gets 1. When D is 0, the top gets 1 and the bottom gets 0. Also, notice that the And gates will fire 0 when the Clk is 0. When the Clk is 1, the top And gate fires 1 when D is 0, and the bottom fires 1 when D is 1.

When the Clk is 0, the outputs do not change, regardless of the input signal D. That is, the circuit continues to store what it had in it before. Let’s check that this is true. When the Clk is 0, the And gates fire 0, and each Nor gate gets a 0 from the And gate on its left. Recall, that a Nor gate fires 1 only when it gets 0/0; it is Or followed by Not. The other input to each Nor gate comes from the opposite Nor gate. If you start with Q(t) = 0 and Q(t)’ = 1, then the top Nor gets 0/1 and Q(t) stays 0, while the bottom Nor gets 0/0 and Q(t)’ stays 1. That is, if the Clk is 0, the outputs do not change.

Now, when the Clk is 1, then the D will change the outputs. If D is 0, the top And gate fires 1. That means the top Nor gate fires 0, so Q(t) is 0. The bottom And gate fires 0, so the bottom Nor gate is getting 0/0, so Q(t)’ is 1. The circuit now cycles back to itself and stabilizes at 0/1. It does not matter what Q(t) and Q(t)’ were to start.

Try to follow through the circuit yourself in the final case when Clk is 1 and D is 1. You should find that he circuit stabilizes at 1/0. Finally, notice that the circuit always stabilizes at 0/1 or 1/0.

To sum up, when the Clk is off, no change to the circuit occurs. When the Clk is on, then the circuit stores the bit sent in at D.

Very cool!

D Clk Q(t)-in Q(t)’-in Q(t)-out Q’(t)-out

0 0 0 1 0 1

1 0 1 0 1 0

0 1 0 1 0 1

0 1 1 0 0 1

1 1 1 0 1 0

1 1 0 1 1 0

This circuit is called a D-latch, or sometimes a D-flip-flop.

Hard Drives

Now that we have explored the inside of the CPU and RAM a little bit, let’s look at the hard drive, i.e., secondary memory. A hard drive is very different from RAM in that it is partly mechanical and partly electronic, while RAM is completely electronic. RAM loses all its storage when the electricity is turned off, and a hard drive does not. That is why we store things in a hard drive for the long term. Hard drives are cheaper per byte (100 times cheaper), way slower (10,000-100,000 time slower). Flash drives that you all use are in between. They have no mechanical parts but they do not need electricity to maintain their values. They are about 10-100 times slower than RAM.

And, if you think that if only I had enough money I would buy RAM big enough for everything, that is silly. You want to have slow memory and fast memory – because you don’t need to keep everything right on your night table. Memory hierarchy is good. It is efficient. It is the way to design a well-balanced machine. There will always be a need for different levels of performance. The stuff we don’t use much can sit in slow access disk and we can wait a few seconds as it loads into RAM. The stuff we need on our desk can stay in RAM until we are done with it, and then we put it back in the closet (on the disk). Slow goes with large and cheap. Fast goes with small and expensive.

I showed you a spinning Hard drive and the head that reads the bits from it at the start of the semester. Now we will discuss a little more detail about how they work and how you can recover data when you think all is lost.

When RAM is turned off, your data in it is gone after a few moments. This is not true for Hard drives because the bits are stored magnetically. A 0 is N-S, and a 1 is S-N. Putting a magnet near a hard drive will corrupt its data. It is as bad as spilling dust or liquid all over it. Hard drives are usually sealed tight to prevent this.

The magnets are arranged in concentric circles around the disk called tracks. Each track has millions of bits organized into sectors. The head moves on an arm above the disk so it can read any of the thousand or so tracks. The disk rotates so that the head can read various sectors. Here is a great short discussion of how the drive looks inside and how it works:

<https://www.explainthatstuff.com/harddrive.html>

The main thing to know as a user has to do with how the computer finds stuff on the disk. It would be too slow to search for a file on the disk sector by sector. Instead, there is a track that holds a directory of files currently on the disk. This called the FAT – or file allocation table. It is a long list of filenames and what tracks and sectors they are in. When the computer searches for a file, it looks for it in the (relatively small) FAT, and then moves to the correct rack and sector by adjusting the arm with the head, and rotating the disk.

So, when you lose your data, is it really lost? What can you do to retrieve lost data? When a file is deleted, the only thing that changes on the hard drive is that it is removed from the FAT directory. The file itself with all its data is still on the hard drive. But it might as well be lost because they computer as no way to find it anymore. It is like when you put something away but you cannot remember where you put it.

There are, however, programs that scan disks looking for patterns of bits that seem like files. Word files have a certain look – lots of characters. Picture files and sound files have certain formatting that are recognizable. The software will pick up these now nameless “files” and guess at their formatting and present you with a list. You can try to open each file with whatever program you think makes sense. Word for a document, Adobe for a pdf, Player for a sound file, etc. Once you open the file it may be all there or partly there depending on what the software was able to recover. You can then piece it back into health and resave it, putting it back on to the FAT once again.

That’s how the geeks do the impossible. The real magicians are the people who write the disk scanning software!

End of Week 4

The last bit left about how computers work is

1. Client/server architecture – that is, programming for the Internet, and
2. Artificial Intelligence – the early paper of Alan Turing.

For (1), you can watch this 6-minute video. It is pretty good for a brief introduction.

<https://www.youtube.com/watch?v=L5BlpPU_muY>

The main point is that it is all software. When I play poker on the Internet, there is a central “server” program that lets people log on, knows who they are, keeps track of their stats, let’s them play at various tables, and manages the game. It expects me to communicate to it with a “client” program that typically you download for free. The two programs talk to each other over the Internet, each with its own responsibilities.

In Poker, for example, the client is in charge of showing you pretty pictures of people and cards and the table. The server accepts everyone’s bets and actions and sends back info to the client about what they can do and what has happened in the hand. Then the clients display that and act as input/output feed.

For web browsing, the server delivers information from some page you are surfing, and the client displays it. The client is called a web browser, and the server is a web-server. There are millions of client-server model programs that run on the Internet. You have all used them many times.

Here is a cool client/server program that lets you play a version of Tic-tac-toe called Ultimate Tic-tac-toe. By the way, this is another option for extending your program – a difficult one, worth a lot of extra credit.

 <https://ultimate-t3.herokuapp.com/online-game/9w6tuuiqztd8f5ydc5g919k9>

Here is another TTT variant: <https://asteri.io/games/otrio/play/>

Artificial Intelligence

Start by reading Alan Turing’s article *Can Machines Think*, and his famous *Turing Test*. Then explore the other links – It is really fun.

Here is the section from our course web page:

## Supplemental Reading Assignment: Alan Turing and Artificial Intelligence

* You should read his famous [article](http://www.abelard.org/turpap/turpap.htm) introducing the *Turing Test* and artificial intelligence, published in Mind magazine in 1950 shortly before he committed suicide.
* Also, read this [exchange](http://web.stonehill.edu/compsci/How-Computers-Work/CommentaryMag.pdf) (article and letter to the editor) about Turing's piece, which appeared in Commentary magazine in the early 1980s.

This [video](https://www.youtube.com/watch?v=5LHFzNMgWzw) discusses the paper.

The movies [Breaking the Code](https://m.youtube.com/watch?v=0nBGz7WqtUI) with Derek Jacobi, and [The Imitation Game](https://www.youtube.com/watch?v=aTx7ISNlRVs) with Benedict Cumberbatch, show in different ways, the role of Alan Turing in the British decryption of German codes in World War II.  They are both excellent films and worth watching in order to understand Turing better.

Alan Turing was a brilliant scientist and a British war hero who helped decode German communications during World War II.  He was an open homosexual, in a time and place not receptive to such action, and was persecuted by his own country.  He tragically committed suicide at a young age.  See [Alan Turing homepage](http://www.turing.org.uk/turing/), to learn about this pioneer of computer science.  He was involved with the German [enigma machine](https://www.youtube.com/watch?v=-qcOCBfRRzg) and [cryptography](http://russells.freeshell.org/enigma/), artificial intelligence, game playing programs, and the [Turing machine](http://aturingmachine.com/) - an abstract model of a computer which began the field of computational complexity.

##     Other AI-Related Links

* + Here is a video about [The Turk](https://www.youtube.com/watch?v=jS4K2D3e2DM), (1770-1854), an early AI hoax. Although too early to be real AI, it showed the receptiveness of the public to the idea of AI. This video includes a Chess game between The Turk and Napolean Bonaparte!
	+ Here is [Eliza](http://www.manifestation.com/neurotoys/eliza.php3), an early AI program related indirectly to the Turing Test.
	+ This is the story of the [greatest Checkers player](http://www.wylliedraughts.com/Tinsley.htm) in the world and his connection with Checker's programs and AI.
	+ The [Loebner prize](https://en.wikipedia.org/wiki/Loebner_Prize) is a modern version of the Turing Test.
	+ This Nature article about [machine learning and the game Go](https://www.nature.com/news/self-taught-ai-is-best-yet-at-strategy-game-go-1.22858) shows the amazing strides AI has made in the area of learning.
	+ [GBT3](https://beta.openai.com/overview) - Machine Learning model to carry on conversations, and [ChatGPT,](https://openai.com/blog/chatgpt) the newest version of machine learning based Chatbots.

We usually discuss this in class. The idea is that now that you know a little about programming and how computers work, you are in a better position to have an intelligent informed opinion about whether machines can think.

Finally, we will discuss a famous 1950s AI program that learned to play Checkers using a very cool algorithm that you could write in Logo with a little guidance.

The program is by Arthur Samuels, an IBM scientist. The point of his program was to demonstrate the flaw in the argument against AI based on computers not being able to learn. The argument, rediscovered by many amateur intellectuals, goes something like this:

A computer is incapable of real thought because it can learn. All a computer can do is what the programmer told it to do. Humans, on the other hand, have creativity – doing more than they are taught and often surpassing their teachers.

This is not a silly argument. Although most of you have learned to program by coming to my lectures and doing my exercises, I routinely see final projects that are done using ideas that I never explicitly taught. These are programs I never would have written myself. That is, humans take rules and ideas, and they run with them, making them their own and expanding on them. This is learning. This is the purpose of education. The purpose is not to merely spit back facts from memory, but to internalize new concepts and apply them in creative ways. The argument suggests that computers are incapable of such creative application. In particular, there would be no way for Samuels’ program to ever outperform the programmer himself.

Samuels argues back with a program that learns! His Checkers program was programmed to monitor its own moves and progress and modify its own program in order to perform better. Samuel’s idea preceded todays “machine learning” algorithms which take Samuels’ work to a much higher level.

Samuels himself was a mediocre Checkers player, and soon enough his program started to beat him. At that point, Samuels ran the program against itself, and it continued to improve to a point at which it plateaued, below the expert level, but way above Samuels himself. Decades later, a program called Chinook, beat the greatest Checkers player in our planet’s history, Marion Tinsley, in a really cool story told here: [greatest Checkers player](http://www.wylliedraughts.com/Tinsley.htm) . Ironically, it is a remarkable *human* story, really worth reading.

Getting back to Samuel’s early learning program, I am going to try and explain how it worked. First, Samuels wrote code to generate all possible legal moves from a given Checkers position. This is not easy but not too hard either. He then wrote code to *evaluate* a given Checkers position. This eval*uation* procedure was a lot harder to write, and this was the part that learned. For the most part, the evaluation procedure counted pieces and Kings on both sides, and he figured out which side had more pieces. That was the main part, but he added subtle extra features to quantify the position more accurately. There were some 12 to 15 features such as centralization of pieces, control of important squares, number of pieces on back rows, etc. He quantified all of this intuitively, calculated a weighted average of all the features, and added this to the piece difference number, so that every position would get a single evaluation number, positive if in favor of Black, and negative if in favor of Red.

Before every move, Samuels’ program would generate all possible moves and all their replies for a few levels. See the figure below:



If the original position had Black to play, then that player wants to maximize his number, while Red would want to minimize his number. The levels alternate between Black and Red. In this example, we go down three levels from the original position. At the bottom is a collection of positions each of which his program evaluates. The numbers show these values. Notice that the original A would like to get to position O, in order to maximize the value of the position (7). But how can he get there? If he heads toward O via C, the opponent will not help him and move to G, the opponent will instead move to F, leaving with our original player a choice between -3 and -5. He can only cut his losses with -3 at position at L.

The original player at A could have chosen better! So, what is the strategy for the original player to maximize his overall value? One needs to work backwards from the bottom toward the top, alternately minimizing and maximizing the choices.

In the example above, D, E, F, and G would maximize and choose 4, 6, -3, and 7, respectively.

Then moving up another level, this time minimizing, B and C get 4 and -3 respectively. Finally, on the top level, we maximize our choice and A gets 4. That means that the best A can hope to do is get a value 4 after three moves from his current position. Thus, A needs to move toward the 4, meaning via B, not C!

Here is the final picture of the tree with all the backed-up values written in:



This algorithm is called the min-max tree algorithm, and it has been a staple of AI game programming for decades. Samuels added one more feature: the learning. Samuels knew that that his evaluation procedure was only a guess - an approximation to the true value of a position. Looking ahead using mini-max helped the evaluation procedure do better. Surely the final backed up values, after mini-max were a better evaluation of a position than the static evaluation procedure would give. So Samuels programmed his computer to change the evaluation procedure so that it would match more accurately the numbers given by the look-ahead version. He had would have the program adjust the weights on the various features until the lookahead matched more closely the static evaluation.

For example, if the evaluation procedure on A says 13, Samuels’ program would adjust it until it said 4. Often, these tweaks helped the program perform better, and the better it performed, the more it would continue to improve. It shot by Samuels’ own ability at the game, but eventually it stopped improving, because the tweaks were not robust enough. It would work for one position but then make terrible estimates on a different position.

Cool! Really cool! The program learned from its own experience using the building blocks provided by its programmer, just as you learn from the building blocks provided by your teachers. The machine learning algorithms work differently with statistics and neural nets, but in principle, they exhibit he same sort of self-assessing learning. That is, programs can learn; we just have to program them so they know how!

I hope that this technical information helps you appreciate the amazing potential of AI, and lets you think critically about the eternal question first raised by Alan Turing: *Can Machines Think*?

If you have not yet done so, to finish the class, make sure to do this reading/viewing:

## Supplemental Reading Assignment: Alan Turing and Artificial Intelligence

* You should read his famous [article](http://www.abelard.org/turpap/turpap.htm) introducing the *Turing Test* and artificial intelligence, published in Mind magazine in 1950 shortly before he committed suicide.
* Also, read this [exchange](http://web.stonehill.edu/compsci/How-Computers-Work/CommentaryMag.pdf) (article and letter to the editor) about Turing's piece, which appeared in Commentary magazine in the early 1980s.

This [video](https://www.youtube.com/watch?v=5LHFzNMgWzw) discusses the paper.

There will be a short essay question on the final about the readings.

The movies [Breaking the Code](https://m.youtube.com/watch?v=0nBGz7WqtUI) with Derek Jacobi, and [The Imitation Game](https://www.youtube.com/watch?v=aTx7ISNlRVs) with Benedict Cumberbatch, show in different ways, the role of Alan Turing in the British decryption of German codes in World War II.  They are both excellent films and worth watching in order to understand Turing better.