Segment 1

[talking head] Hello. I'm Eric Hehner. I'm at the University of Toronto, Canada. And I have the pleasure to give this short course on digital circuit design. This is the first of four online lectures. One of the nice things about an online lecture is that you can watch it on a computer or any portable device anytime and anyplace you want to. You can stop anytime, and resume later. You don't have to play it straight through as though it were a classroom lecture. You can and should pause at any point where you need to think about what was just said before you go on. Some things might go by too fast, but you can replay anything you want to see again. I hope you enjoy the course, and benefit from it.

[2] The word "circuit" means a closed loop, like a [3] circle, but not necessarily round. It's used for electrical things because the electricity follows a [4] closed path from a source, such as a battery, through whatever needs electricity to work, such as a light bulb, and back to the source. That's why an [5] electrical cord has two wires in it. At any instant, one wire goes from source to device, and the other wire goes back. In the case of alternating current, they keep switching directions, but still, the two wires make a circuit. If the cord has three wires, the third wire is a ground wire just for safety; it isn't part of the circuit.

[6] The word "digit" means finger. And since people sometimes count on their fingers, it also means a [7] number symbol. A digital circuit is a circuit that has distinct, discrete states. For example, when this clock changes state, the final digit will change from 5 to 6. As far as we can tell, or maybe I should say as far as we care, there's no halfway, when the state is halfway between 5 and 6. [8] This round clock is not digital because the hands move continuously. If a hand is somewhere, and then later it's somewhere else, then there was a time when it was halfway between. Well, if you look very closely, at the level of elementary particles, and at a time scale of 10 to the power minus 43, you will see quantum behavior, which is not continuous. But above that scale, the hands seem to move continuously. This course is not about circuits with continuous behavior. It is about the design of digital circuits.

[9] I want to start at the very beginning, with elementary particles. These are the protons, neutrons, and electrons that make up atoms. Protons are positively charged, electrons are negatively charged, and neutrons are neutral because a neutron is made of a proton and an electron. The protons and neutrons are the nucleus of an atom, which means they are at the center of the atom. Popular pictures of atoms show the electrons as little balls orbiting the nucleus, like planets orbiting a sun. But that's not even close to true. The electrons are around the atom, but they are not balls, and they don't orbit. [10] Sometimes physicists say to think of an electron as a cloud, and that's closer to true, but it's a probability cloud, which is not something we are familiar with in our ordinary experience. Electrons come in several layers, or shells, around the nucleus. And it's the outer shell that decides the electrical properties of the atom. If the electrons of the outer shell are bound strongly to the atom, then it's an electrical resistor. If the electrons of the outer shell are weakly attached to the atom, so they can move from atom to atom, then it's an electrical conductor. [11] A good example of a conductor is copper, which is what wires are made of, and [12] a good example of a resistor is silicon, which is what sand and glass and ceramics are made of.

The way to make a [13] semiconductor is to start with some silicon, heat it until it melts, then pour it onto a flat surface in wafers, which are like thin pancakes in size and shape, and let it cool. If the room is free from dust and vibrations, the silicon cools into a crystal structure, which is what this picture is trying to show. Each Si is the nucleus of a silicon atom. And the dots are supposed to be electrons in the outer shell. There are 4 electrons in the outer shell of a silicon atom, but in the crystal structure, each outer electron belongs equally to a pair of atoms. They are shared electrons. They are strongly bound in their place in the structure, so a silicon crystal wafer is a resistor. The picture is wrong in

several ways. The crystal structure is 3 dimensional, but this is a 2 dimensional picture. And a nucleus isn't a pair of letters. And electrons aren't dots. What the picture shows correctly is that it's a regular structure, and that there are 4 electrons in the outer shell of each silicon atom, and these electrons are shared between neighboring atoms. To make a semiconductor, some of the silicon atoms are replaced by phosphorus atoms, and some are replaced by boron atoms. Watch closely: [14] there. That's called doping. The phosphorus atoms have 5 electrons in the outer shell. The extra electron is there, but it doesn't have a place in the crystal structure, so it is weakly bound to its atom. That's called negative doping. The boron atoms have 3 electrons in the outer shell, so there's a hole, which means a place for another electron, in the crystal structure. That's called positive doping. The doping makes extra electrons that are weakly bound and easily movable, and holes which are places for the extra electrons to go. That's a semiconductor. Now let me remove the silicon atoms from the picture [15] so we can concentrate on the doped regions. An adjacent pair of positively and negatively doped regions is called a [16] diode. The extra electrons from the phosphorus atoms, which have no place in the structure, wander around, and when they come to places around the boron atoms where there are holes, they stay there. [17] But they are still weakly bound to the boron atoms because there's no corresponding proton in the boron nucleus.

Suppose we [18] connect the negative side of the diode to a source of electrons, and the positive side to a destination for electrons. Electrons flow from the source of electrons to the phosphorus atoms because they are attracted by the protons that don't have electrons. And then electrons move across the boundary from the phosphorus atoms to the boron atoms just as before. And electrons flow from the boron atoms to the destination for electrons. So there is a [19] flow of electrons from left to right in the picture. That's an electric current.

Now let's put a [20] source of electrons on both sides. All the places for electrons get filled, but there's no further place for them to go, so there's no electric current. And if we [21] remove the electrons from both sides, now there's a place for them to go, but no source, so no current. Finally, if we [22] have a source on the right side, and a destination on the left side, the extra electrons from the phosphorus atoms still find places around the boron atoms, but they don't get replaced, and they don't have any further place to go, so there's no current.

A diode is a one-way gate for electric current. It allows electrons to flow through it from left to right, but not from right to left in this picture.

Here's [23] a simpler, one-dimensional picture of some electrons, with a hole or missing electron in one place. Suppose [24] the electron to the left of the hole moves into the hole. [25] There. Now suppose [26] again the electron to the left of the hole moves into the hole. [27] There. We see [28] electrons flowing from left to right, and we see [29] the hole flowing from right to left. Now [30] here's another line with electrons and holes placed randomly, and a source of electrons on the left side, and a destination for electrons on the right side. A source of electrons is the same thing as [31] a destination for holes, and a destination for electrons is the same thing as a source of holes. [32] Whenever there's an electron to the left of a hole, the electron will move [33] into the hole. The electrons move from left to right, and the holes move from right to left. And [34] again, when an electron is to the left of a hole, the electron and hole will switch places, and if there's a hole at the left end, it will get filled from the electron source, or to say the same thing differently, the hole will go into the destination for holes. Likewise at the right end, an electron will go into the destination for electrons, or to say the same thing differently, a new hole will be supplied. [35] There. You see that when we talk about electric current, we could talk about electron flow in one direction, or we could talk about hole flow in the other direction. I don't know why, but when electrical engineers talk about which way electricity flows, they are talking about the flow of holes.

That's the basics of electricity. Here [36] are the symbols used for drawing electrical circuits. A destination for electrons, or a source of holes, which is also called a positive voltage, or high voltage, or power, is this symbol. [37] A source of electrons, or destination for holes, or negative voltage, or low voltage, or ground, is this symbol. The difference between them is typically about 5 volts. It's too bad we have so many different words to mean the same thing. I might say power, or positive, or high voltage, or up, and they all mean the same. I might say ground, or negative, or low voltage, or down, and they all mean the same. A [38] conductor is a line, like a wire. A [39] resistor is a zig zag line. A [40] diode symbol looks like this. Holes can flow from left to right, but not from right to left. To say the same thing differently, a diode can support a voltage difference when the left side is ground and the right side is power and no current flows. But it can't support a voltage difference with power on the left and ground on the right; that's when current flows until the two voltages are equal. A [41] transistor symbol looks like this. A transistor is three doped regions in a row. The regions could be positive negative positive, or they could be negative positive negative. When the middle region has one voltage, current can flow through the transistor in either direction. When the middle region has the other voltage, current cannot flow through the transistor. So a transistor is an on-off switch.

[42] Before we go any further toward digital circuit design, I should tell you the designer's main trick. It's called binary abstraction. That means there are only two voltages, the ones we call power and ground. At any place in a circuit, the voltage is either power or ground. It can change over time, and that's what this picture shows. Time goes from left to right. At some particular place in a circuit, the voltage might start off at ground, and then after a time, it suddenly changes to power, and then later it changes back to ground, and so on. Each time it goes up and later goes down is called a [43] pulse, so there are two pulses in this picture. That's not what really happens. What really happens looks more like this [44]. But the binary abstraction is to pretend it's the top picture. And it's the electrical engineers' job to make it be as much like the top picture as possible. That's so we can use binary algebra to design digital circuits, which is a lot simpler than continuous calculus, which we would need if we had to describe the bottom picture.

Now [45] we can look at some circuits that use binary abstraction. The simplest circuits are called gates, and this one is an **or** gate. *a* and *b* are the inputs, and *c* is the output. [46] Suppose *a* and *b* are both positive, meaning they are at the high voltage. Then *c* has to be positive because the diodes cannot have a positive voltage on their left and a negative voltage on their right. Since *c* is positive, a small current flows through the resistor, but *c* is resupplied with holes from *a* and *b*. So *c* stays positive. [47] Now suppose *a* is positive and *b* is negative, meaning it's at the low voltage. *c* still has to be positive because the *a* diode cannot have a positive voltage on its left and a negative voltage on its right. But the *b* diode can have a negative voltage on its left and a positive voltage on its right. If *a* is negative and *b* is positive, the result is the same. [48] Now suppose both *a* and *b* are negative. According to the diodes, *c* could be either positive or negative. But if it's positive, a current will flow through the resistor until it's negative, and it is not resupplied with holes from *a* and *b*, so it stays negative. So that's how it works electrically. If either *a* or *b* or both are positive, then *c* is positive. If both are negative, then *c* is negative. Now we make the binary abstraction, and its symbol is [49] this. An **or** gate can have more than two inputs. If one or more inputs are positive, the output is positive, and if all inputs are negative the output is negative.

The next gate is [50] the **and** gate. Again the inputs are *a* and *b*, and the output is *c*. [51] If both inputs are positive, the diodes allow *c* to be either positive or negative. But if it's negative, a current will flow through the resistor until it's positive, and then it will stay positive. [52] Now if *b* becomes negative, the *b* diode won't allow a voltage difference with negative on the left and positive on the right, so *c* becomes negative. This causes a small current to flow through the resistor, but the *b* diode keeps *c* negative. The result is the same

if *a* is negative and *b* is positive, or if both *a* and *b* are negative. [53] So, if both *a* and *b* are positive, *c* is positive. If one or both of the inputs are negative, the output is negative. After the binary abstraction, [54] this is its symbol. An **and** gate can have more than two inputs. If all inputs are positive, the output is positive, and if one or more inputs are negative the output is negative.

[55] A **not** gate can be built from a transistor and a resistor. If the input is positive, the output is negative, and if the input is negative, the output is positive. When the input is positive, the path between ground and output is conducting, so the output is at ground voltage. When the input is negative, the path between ground and output is not conducting, so the current through the resistor pulls the output up to a positive voltage. [56] After we make the binary abstraction, the symbol for a **not** gate is a circle.

[57] The last gate we look at is the **delay**. Its symbol is a triangle. Its output is the same as its input, but delayed. [58] That's what this diagram is trying to say. The horizontal axis is time, and the vertical axis is voltage. The top graph is a random input, and the bottom graph is the output, which looks just like the input, but shifted to the right, which means delayed in time. You can build a **delay** gate by putting an even number of **not** gates in a row. If you want a lot of delay, use a lot of **not** gates; if you want a little delay, use a few **not** gates. We'll take a break here, and resume with the next segment whenever you feel like it.

Segment 2

[59] We have talked about electrons flowing in one direction, and holes flowing in the opposite direction. Now I want to talk about information flow, which is the direction from inputs to outputs. That's not the same as electron flow, and not the same as hole flow. [60] What makes something an input is that you choose its value. What makes something an output is that you measure its value. Sometimes we show the direction of information flow with a little arrow. [61] We can join wires together, and we can have information flow from one of the branches to the other branches, like this. You choose what *a* is, and then you measure *b* and *c* and you find that they are the same as *a*. [62] But you cannot have two inputs joining to make one output. That's because you can choose different values for *a* and *b*, and then *c* would have to be both of those values, and that's impossible. If you want to join two inputs into one output, you have to use a gate: either an **and** gate or an **or** gate. Then you know what the output is for each combination of inputs. [63] Sometimes information paths have to cross each other. When they do, they are not connected.

[64] Now we are ready for our first circuit, which is called a multiplexer. Every time we show a circuit, we show two pictures. The first picture is a box, and it shows what the inputs and outputs are, and in the box there are words or symbols to say what the function of the circuit is. A multiplexer has three inputs, and in this picture they are labeled *x*, *y*, and *z*. And there's one output, labeled *q*. If *x* is positive, then *q* has the same value as *y*. If *x* is negative, then *q* has the same value as *z*. You can think of *x* as being like a knob, and when you turn it one way, *y* is connected to *q*, and when you turn it the other way, *z* is connected to *q*. The other picture fills in the box, so we know how the circuit is built. Since there are three inputs, there are 2 to the power 3 input combinations. And the only way to see that the circuit is a correct implementation of a multiplexer is to look at all 8 input combinations, and see what the output is for each of them. So right now you should pause this lecture and check each combination of inputs.

[65] The next circuit is the demultiplexer. It has 2 inputs *x* and *y*, and 2 outputs *q* and *r*. If *x* is positive, then *q* is the same as *y*, and *r* is negative. If *x* is negative, then *r* is the same as *y*, and *q* is negative. Again, *x* is like a knob, and when it is turned one way, *y* is connected to *q*, and when it is turned the other way, *y* is connected to *r*. On the right side, we have its implementation. This time there are only 4 input combinations to check. So

pause and check them.

[66] Now we have one of the most interesting circuits in all of digital circuit design: the flip-flop. Input *d* is the data input, and input *c* is the control input. When the control input is positive, the output is the same as the data input. When the control input is negative, the output remains constant. On the right side, you see the implementation. It has a multiplexer and a delay. When the control input is positive, the multiplexer output *q* is the same as the data input. When the control input is negative, the multiplexer output *q* is the same as its "else" input. The interesting thing is that [67] the output is fed back through the delay to the "else" input of the multiplexer. This is called a feedback loop. So the "else" input is what *q* was a moment earlier. So *q* is what it was a moment earlier. In other words, *q* stays the same as it was. We have to have [68] the delay because without it, there's no constraint on output *q* when the control is negative. The output could be either positive or negative. But with the delay, the output has to stay the same as it was. But a multiplexer is not instant; it involves a tiny delay, and that delay is enough. The delay in the picture just represents the delay already present in the multiplexer.

[69] Here's what happens. *c* and *d* are inputs, so I can make them be anything I want. [70] Initially, *c* is positive, so *q* has to be the same as *d*. [71] Then *c* falls. And while it's down, *q* has to be unchanging. [72] When *c* goes back up, *q* has to be what *d* is, which is up for a moment, and then goes down and up a couple of times, with *q* doing the same. [73] Then *c* falls again. At that instant, *d* happens to be down, so *q* stays down at least until the next time *c* goes up. And so on.

[74] A flip-flop is 1 bit of memory, and memory is implemented by a feedback loop. The flip-flop is remembering what *d* was the last time *c* was up.

This flip-flop is called a "sensitive" flip-flop, which means that the output is the same as the data input whenever the control input is positive. In the picture on the left, the little square box at the control input means "sensitive".

[75] Our next two circuits are called edge triggers. The first one is called a rising edge trigger, and the second is called a falling edge trigger. [76] The box picture of a rising edge trigger has an up arrow in it. Its output, as you can see from the circuit on the right, is up just when the input is up and was down a moment ago. [77] The timing diagram shows it best. Whenever the input has a rising edge, the output is up for a moment. The delay has to be just long enough to detect the rising edge, but no longer. [78] A falling edge trigger is similar. Its box has a down arrow. Its output is up just when the input is down and was up a moment ago. [79] Its timing diagram shows the short pulses at each falling edge of the input.

We can use edge triggers with various circuits, and [80] here is a rising edge triggered flip flop. Its picture, on the left, has an up arrow in a box at the control input. The circuit on the right is a sensitive flip flop, so it has an empty box at the control input, and a rising edge trigger on the control input. Let's look [81] at the timing diagram. At the start, left side, we see that the *c* and *d* inputs are both down, but that doesn't tell us what the *q* output is. We don't know *q* until [82] the first rising edge of *c*. And then *d* is down, so *q* is down from then on, at least until [83] the next rising edge of *c*. And then *d* happens to be up, so that means *q* is up until the next rising edge of *c* [84]. And then *d* is up, so *q* stays up until the next rising edge, and so on.

[85] Now here is a falling edge triggered flip flop. It is built from a sensitive flip flop, a falling edge trigger, and a delay. The delay is just so that when the *c* input falls, the output *q* will be what *d* was just before *c* fell. [86] Here's the timing. We don't know what *q* is until [87] the first falling edge of *c*. And then *d* is up, so *q* is up until the [88] next falling edge of *c*. Now *d* is down, so *q* is down until [89] the next falling edge. And *d* is up here, so *q* goes up, and so on.

[90] Now I want to show you two merge circuits. The first one is called a 1-2-merge. It emits a pulse on its output when it receives pulses on its inputs in the right order.

If there's a pulse on *a* first, and then a pulse on *b* second, then there's a pulse on *q*. [91] In the implementation on the right, I have labeled paths *r* and big *A* to help explain how it works. The circuitry between *q* and *r* is an edge-trigger. *r* is up just at the falling edge of *q*. But mostly it's down. So big *A* is up if input *a* is up, or was up. See the feedback loop? So big *A* remembers that there has been a pulse on input *a*. Now when input *b* goes up, there's a pulse on the output *q*. And at the end of that pulse, *r* is momentarily up, so big *A* goes down, ready for the next pair of input pulses. You should stop and look at this circuit a little more to satisfy yourself that it works.

[92] The other merge circuit is a symmetric merge. There's a pulse on the output when there have been pulses on both inputs, no matter which order they came in. In the implementation, big *A* remembers if there has been a pulse on input *a*. Big *B* remembers if there has been a pulse on input *b*. When there has been a pulse on both, there's a pulse on output *q*. And then, at the falling edge of *q*, *r* is up for a moment so that big *A* and big *B* will go down, ready for another pair of input pulses. You might need to look at this circuit a little longer too.

[93] Now I have to make sure you know how numbers are represented in binary. What we're all used to is decimal representation, which uses ten digits to represent numbers. Binary uses two digits. A binary digit is also called a [94] bit. When we write a number in [95] decimal, the rightmost digit is the units, the digit to its left is the tens, the digit to its left is the hundreds, and so on. Each digit is multiplied by a power of ten, and then we add. [96] Binary works the same way, except it's powers of two. These powers, these exponents, give us the way we number the digits, [97] from right to left, starting at 0. That's enough for this segment.

Segment 3

[98] Here's a useful convention for drawing circuits. A double line means some number of pathways, without saying how many. Could be one, could be a hundred. So this picture really means a picture like [99] this one, for example. And sometimes, but not always, a picture like this can be [100] broken up into separate circuits, one for each pair of input and output. [101] In this picture at the top left, the *a* input and the *b* output are double lines, so they represent some number of inputs and outputs, and again I've made it be 4 inputs and 4 outputs. The *c* input is just one line, so it's just a single input. The circuit on the top might be composed of several circuits, each one with one of the *a* inputs and one of the *b* outputs, and all of them with the *c* input. [102] Now here's a circuit with an *a* input and a *b* input and a *c* output, and all of them are double lines, meaning some number of inputs and outputs. And perhaps this circuit is composed of several circuits, each one with one of the *a*, *b*, and *c* paths. We'll be using pictures like this in the rest of the course.

[103] This next circuit compares two bits. The output is positive if the inputs are unequal, and negative if they are equal. On the right, you see that *c* is positive if either *a* is positive and *b* is negative, or, that's the **or** gate, or *a* is negative and *b* is positive. This circuit to see if two bits are unequal is also called “exclusive or”, and “parity”, and “addition modulo 2”. That's a lot of different names for unequality. Now [104] let's put a bunch of these circuits beside each other so we can compare many pairs of bits and see which pairs are unequal. That's called “bitwise unequality”. Although the picture doesn't show it, there is the same number of *a* inputs as *b* inputs, and the same number of *c* outputs.

[105] If we want to compare two numbers to see if they are unequal, we need all the inputs for each of the two numbers, but we just want one bit of output that's positive when the numbers are unequal, and negative when they are equal. And [106] here's how we build it. We compare the numbers bitwise, and feed all the outputs into an **or** gate. If one or more pairs of corresponding bits are unequal, then the numbers are unequal.

[107] Numbers can also be compared to see which one is smaller and which one is

larger. The output bit is positive if *a* is smaller than *b*. We build this circuit [108] from a sequence of simpler circuits. LT stands for less than. Each LT box takes one bit of input from each of the operands, and a bit of input from its neighbor on the right. Its output bit goes to its neighbor on the left. Each of these output bits tells whether the part of *a* to its right is less than the part of *b* to its right. So the output bit from the leftmost box tells whether all of *a* is less than all of *b*. Now we just need to build the LT boxes. [109] And here is the circuit for one LT box. When a_i is 0 and b_i is 1, then the output is 1, saying yes, the part of *a* to the right is less than the part of *b* to the right. When a_i is 1 and b_i is 0, then the output is 0, saying no, the part of *a* to the right is not less than the part of *b* to the right. When a_i and b_i are equal, then the output is the same as the c_i input. You need to spend a little time checking all 8 input combinations to see that this circuit is correct.

[110] Now we design a circuit to add two numbers. The inputs *a* and *b* are each an *n* bit number. The *s* output is their *n* bit sum, if their sum is representable in *n* bits. The *overflow* output is positive if the sum is too big to be represented in *n* bits. Before we can design an adder, we have to know how to add. [111] So here's an example. We start at the right side, which is column number 0. We add 1 plus 0 and get [112] 1. In the next column, column number 1, we add 1 plus 1 and get [113] 2, which is 1 0 in binary. The 0 is a sum bit, and the 1 is a carry bit. In column 2, we add 0 plus 0 plus 1, and get [114] 1. In column 3 we again add 1 plus 1 and get [115] 1 0. Now we have to add 1 plus 1 plus 1, and we get [116] 3. which is 1 1 in binary. In column 5, the leftmost column, we add 0 plus 0 plus 1, and get [117] 1. So there's the sum. When there's no carry, the carry is [118] 0, and I've also put a 0 carry to the left of the other carries because that's the *overflow* output. Now we can design the circuit. [119] We use one box for each column in the addition. Each ADD box takes one *a* bit, one *b* bit, and one *c* bit which is the carry from the previous column. It adds them together, and produces a sum bit, and a carry bit to the next column. So all we have to do is implement an ADD box, and then we'll have an adder. I'll work on the sum output first. That's the *s* output. If you're adding an even number of 1s, the sum bit is 0, and if you're adding an odd number of 1's, it's 1. Let me clear a space, [120] so for the sum output, all we need is [121] something that says whether there's an odd number of 1s among the 3 inputs. If the inputs are [122] 0 0 1, or [123] 0 1 0, or [124] 1 0 0, or [125] 1 1 1, then the output is 1. So that's the sum output. Now the [126] carry output. It's 1 if there are at least two 1s among the inputs. That could be the *a* and *b* inputs, or the *a* and *c* inputs, or the *b* and *c* inputs. Or it could be all three, but that's already included in the other options. I'll call it MAJ for majority. We just [127] put the MAJ and ODD boxes together, and that makes an ADD box. And that completes the adder circuit.

Now [128] let's make a subtractor. It subtracts *a* minus *b*, and the difference is *d*. The *overflow* output is 0 if *a* is greater than or equal to *b*, so we can do the subtraction, and it's 1 if *a* is less than *b*. So overflow is just the same as the less than circuit we did earlier, and overflow isn't really the right name for it. [129] So how do we subtract? Here's how I was taught, in decimal. Starting at the right, you can't take 6 from 2, so you borrow from the 0 in column 1. Well, you can't borrow from 0, so you look further left, in column 2. That's still a 0, so you borrow from the 3 [130], making it a 2, and making the 0 to its right into a 10. Now we can borrow from the 10, making it 9, [131] and making the 0 to its right a 10. And now we can make the borrow we wanted to make in the first place, so [132] that 10 becomes a 9, and the 2 becomes 12. Subtract 6 from 12 and get [133] 6. Subtract 5 from 9 and get [134] 4. Subtract 4 from 9 and get [135] 5. Subtract 3 from 2, well you can't, so [136] borrow from the 1, now take 3 from 12 and get [137] 9. And finally, 0 from 0 is [138] 0. Done. But that's way too complicated, and it leads to a poor circuit. So let's [139] start again. Starting on the right side, in column 0, we want to take 6 from 2, and we can't, so we [140] carry a 1 to column 1. I'm putting the carries in the middle because that 1 and the 2 in green make 12, and we can take 6 from 12, and that leaves [141] 6. In [142] column 1, we

add the 5 and the 1 in red, and get 6, which we want to subtract from 0. We can't, so we [143] carry 1 to column 2. Now we're subtracting 6, which is the red sum, from 10 which is in green, and we get [144] 4. In [145] column 2, we add the red 4 plus 1, and get 5, which we are subtracting from 0, so we [146] carry, and now we subtract 5, which is the red sum, from the green 10, and get [147] 5. In [148] column 3, we add the red 3 plus 1, and get 4, and subtract that from 3, so we have to [149] carry, and we subtract the red sum from the green 13, and get [150] 9. Finally, in [151] column 4, which is the leftmost column, we subtract the red sum from 1 and get [152] 0. I'm going to [153] put a carry of 0 where there wasn't a carry, and the 0 at the left side of the carries means that the number we're subtracting is less than or equal to the number we're subtracting from, so the answer is good.

[154] Here's a binary example. We'll [155] start by putting a carry of 0 in column 0. We subtract 0 plus 0 from 1, and get [156] 1 with a carry of 0. Now, in column 1 [157], we subtract 1 plus 0 from 0, so we have a carry of 1 [158], and we subtract 1 plus 0, which is red, from 2, which is green, and get 1. Now in [159] column 2, we subtract 0 plus 1 from 1 and get [160] 0 with a carry of 0. In [161] column 3, we subtract 1 plus 0 from 0, so we need to [162] carry 1, and now we subtract 1 plus 0 from 2, and get 1. In [163] column 4, we subtract 1 plus 1 from 0, so we [164] carry 1, and subtract 1 plus 1 from 2 and get 0. And [165] finally, in column 5, we subtract 0 plus 1 from 1 and get [166] 0 with a carry of 0, which means the answer is good.

[167] A subtractor looks a lot like an adder. There's one box for each column. Its inputs are a bit from each operand, and a carry bit from the right. Its outputs are one bit of the difference, and a carry bit to the left. [168] I'll clear a space, and [169] here's the SUB box. The difference bit is exactly the same as for ADD, and the carry bit is the LT circuit we saw earlier, because the *overflow* output says whether input *a* is less than input *b*. So that's subtraction.

So far we've just been talking about natural numbers, 0, 1, 2, and so on. [170] For integers, which includes positive numbers, zero, and negative numbers, computers use two's complement representation. With just 4 bits it looks like this, but in a computer it's usually 32 bits. The natural numbers look just as you'd expect. They all start with 0. The negative integers look strange if you haven't seen them before. The whole purpose of this representation is that [171] it uses the same circuits for addition and subtraction that work on natural numbers, with one little change: overflow is positive when the leftmost two carries differ. So now we have addition and subtraction for integers. We'll do a multiplier later.

Now [172] I want to make a generalization of the multiplexer and demultiplexer circuits that we saw earlier. The multiplexer we had earlier had 3 input bits. One of them, the **if** input, selected between the other two inputs, the **then** input and the **else** input, to produce the output. In this generalization, the **if** input has become *n* inputs, and the **then** and **else** inputs have become 2^n inputs. In my example implementation on the right, *n* is 2. The 2 inputs labeled *x* say which of the 4 inputs labeled *y* becomes the *z* output. Look at the circuit for a few minutes and you'll see how it works, and how to generalize it to any number of bits.

[173] The demultiplexer can be generalized in the same way. Instead of using 1 bit to choose between 2 destinations, it uses *n* bits to choose among 2^n destinations. In the picture on the right, *n* is again 2. So the *y* input goes to one of the 4 possible destinations, and the other destinations all get a negative value. Take the time to check all 8 combinations of inputs, and figure out how to generalize it to any number of inputs.

[174] This is a register. It has a control input *c*, a data input *d*, and an output *q*. A register is just a group of flip-flops that all have the same control input. This picture shows a 4 bit register, but it could be any number of bits. We say that the register holds some information, which just means that the output is that information. The information could be

a number, or it could be code for some characters, or it could be a computer instruction, or anything else. To change the information in the register, put the new information on input *d*, and then send a pulse on input *c*. After the pulse, the output *q* will be the new information, and it will continue to be that same new information until the next pulse on *c*.

The register shown here is sensitive, as you can see by the empty square at the control input. That means that while the control input is positive, if the data input changes, the output changes too. But we can [175] put edge-trigger circuitry at the control input to make an edge-triggered register. This one is a falling edge-triggered register. Its output changes only at the falling edge of a pulse on the control input.

Finally [176], our last basic circuit is a memory. A flip-flop is 1 bit of memory, and a register is *n* bits of memory. This is a random access memory. Conceptually, a memory is just [177] a collection of registers, plus a demultiplexer and a multiplexer. In a computer's memory, typically each register holds 8 bits of information. 8 bits is called a byte. And instead of 4 registers as in this picture, there are billions of registers. The registers are numbered 0, 1, 2, and so on, and a register's number is called its address. Here's how it works. If you want to write some information into one of the registers, you put the information into the data input *d*. As you see in the picture, *d* goes to all registers. But a register doesn't change its value until it gets a pulse on its control input. You also put the address where you want the information to go into the writing address input *w*. Now you send a pulse into the control input *c*. The demultiplexer sends that pulse to just the one register addressed by *w*. So just that one register changes its value. If you want to read the information that's in some particular register, you put its address into the reading address input *r*, and that goes into a multiplexer which selects just the information you want. Actually, that multiplexer on the right side of the picture is really several multiplexers, one for each bit in a register. This memory is sensitive, but we can put edge-trigger circuitry at the control input if we want to.

That's all the basic circuits we need. We're going to design more interesting circuits by a new method, starting in the next segment.

Segment 4

[178] In this segment we see how to design interesting and complex circuits. First we write a program, then we compile the program to a circuit. You can write the program in any programming language, so choose your favorite. [179] Each variable in your program gets compiled to a falling edge-triggered register. This one is for a variable named *x*. The register needs enough bits to store the values that can be assigned to the variable. If the type of the variable is 32-bit integers, then the register needs 32 bits. If it's an ASCII character variable, then the register needs 8 bits. If it's a binary variable, then 1 bit is enough, and that's just a flip-flop. The control inputs and the data inputs come from all the places in the program where the variable is being assigned a value. And the output goes to all those places in the program where the value of the variable is needed.

Each [180] array in your program gets compiled to a memory. This one is for an array named *A*. Each register in the memory needs enough bits to store the values that can be assigned to an array element, and there must be enough registers in the memory for the number of elements in the array. All the inputs on the left, the control inputs, the writing address inputs, and the data inputs, come from all places in the program where an array element is being assigned a value. On the right, the reading address comes from all places where the value of an array element is needed, and the memory output goes to all those same places.

[181] Suppose the assignment statement *x* gets *x* plus *y* appears somewhere in your program. Maybe it looks like the example on top, or maybe the one on the bottom, or maybe some other way, depending on the syntax of your programming language. Anyway,

it gets compiled to the circuit shown here. *x* and *y* come from the registers for those variables, and they go into an adder. Input *c* is a control input. A pulse on *c* causes the assignment to be executed. It first goes through a delay that's just long enough for the addition to happen. Then the pulse goes three places. The bottom one is to some **and** gates, so the result of the addition can go to the data input for variable *x*. The pulse also goes to the control input for variable *x* to cause the register to change its value. The pulse also goes through another delay that's just long enough for variable *x* to latch onto its new value, and then it's an output called *c* prime; that means the assignment is finished.

[182] An assignment statement that assigns a value to an array element is a little more complicated. Suppose *A* of *i* is assigned the value of variable *x*. The syntax may be like one of the two shown here, or maybe something else. In the circuit, we need the value of variable *x*, and we need the value of variable *i*, and we need a control pulse to execute the assignment. The pulse goes to **and** gates that let the value of *x* go to the data input for the memory that stores array *A*. And the pulse also goes to **and** gates that let the value of *i* go to the writing address for the memory that stores array *A*. The pulse also goes to the control input for the memory, and that causes the value of *x* to be stored at address *i* in the memory. The delay is just long enough for the memory to latch onto the new value, and the pulse comes out *c* prime to say it's all done.

[183] To put two assignment statements in sequence, or to put any two parts of a program in sequence, just connect the *c* prime output of the first part to the *c* input of the second part. The pulse from the first part that says the first part is all done is the pulse into the second part to start the second part working.

[184] Most programming languages are not very good at expressing parallel computation, and your favorite language might not even have any way of expressing it. Which is a pity, because it's very useful. If you can put two parts of a program in parallel, then this is the circuit you get. The control input goes to both parts to start them at the same time. But they might not finish at the same time. The parallel composition is finished when both parts are finished, so the *c* prime pulses go into a symmetric merge, which has a pulse on its output when both inputs receive a pulse in either order or simultaneously.

[185] All programming languages have an **if** statement. Its syntax might be one of these, or something else. *b* is any binary expression, and *P* is any part of a program. The box labeled *b* evaluates expression *b*. If this expression makes use of some variables, then it has inputs that come from those variables. The control *c* goes through a delay that's just long enough for *b* to be evaluated. If *b* evaluates to true, which is a positive voltage, then the pulse from *c* goes to start execution of the box labeled *P*. And when *P* is done, it sends out a pulse to say so, which goes through the **or** gate and says this **if** statement is done. But if *b* evaluates to false, then the input control pulse goes straight to the **or** gate and says we're done.

[186] An **if** statement might have an **else** part, in which case the circuit looks like this. Box *b* evaluates the binary expression, and the demultiplexer routes the control pulse to one of two boxes, either box *P* or box *Q*. When that box finishes, it sends out a pulse that goes through the **or** gate and says the **if** statement is all finished.

[187] Now let's look at a loop construct. The most common kind of loop is the **while** loop, whose syntax might look like one of these examples, or maybe like something else. Anyway, the circuit starts off like the **if** statement circuit. The binary expression *b* is evaluated. This binary expression probably makes use of some variables, and so it has inputs coming from those variables, but they aren't shown in the picture. The control pulse *c* goes through an **or** gate, and then through a delay that's just long enough for expression *b* to be evaluated. If the value of *b* is true, then the pulse comes out the **then** output of the demultiplexer, and starts box *P* working. When box *P* is finished, its final pulse goes back through the **or** gate and through the delay that allows expression *b* to be reevaluated, and the pulse again comes out of the demultiplexer. If the value of expression *b* is false, the pulse

comes out the **else** output of the demultiplexer, and says that the loop is finished. There are other kinds of loop constructs, and they all work this same way.

[188] The last programming language construct I want to look at is the procedure, or it's sometimes called a function or a method. It's a piece of program that you define or declare one place, and then call it from other places. In the top picture, the definition, or declaration, of the procedure produces a circuit, which is box *P*, and its control input comes from all the places where *P* is called. Any one of the calls starts *P* executing. The control output from *P* goes back to all the places where *P* is called. The bottom picture is the circuit for one of those places where *P* is called. When a pulse comes in its control input, that pulse goes up to *P* and starts it executing. The same pulse also goes into the 1 input of a 1-2-merge. Then later, when *P* is finished, its final pulse comes back to this call circuit, into the 2 input of the 1-2-merge. Since the merge has now had a pulse first on the 1 input, then on the 2 input, now there's a pulse on its output control saying the call is finished, so execution can continue with whatever follows the call. The final pulse from *P* also goes to all the other places where *P* can be called from, and goes into the 2 input of their 1-2-merge circuits. But for those other call places, there wasn't any previous pulse in the 1 input, so no pulse comes out of those 1-2-merges. I think this will all become clearer with an example.

[189] Here is a C program to compute the greatest common divisor of two positive integers. It starts with a declaration of two integer variables *a* and *b*. On the next line we have the greatest common divisor procedure. The first **void** means it doesn't have a functional result, and the second **void** means it doesn't have any parameters. Instead, it computes the gcd of the values of variables *a* and *b*, and it puts the result in both those variables. The starting values of *a* and *b* are its input, and the ending values of *a* and *b* are its output. The body of this procedure is a **while** loop. The body of the **while** loop is an **if** statement with an **else** part. It says: while *a* is not equal to *b*, if *a* is less than *b* then *b* is assigned *b* minus *a*, and otherwise *a* is assigned *a* minus *b*. And that's the end of the procedure. On the bottom line we have the main program, which starts by assigning *a* the value 3, and in parallel, assigning *b* the value 27. The C language doesn't have any parallel connective, so I've used two vertical lines to mean in parallel. After those assignments, it calls gcd. gcd will change the values of variables *a* and *b*. It will change them both to 3, which is their greatest common divisor. Well, that's not a change for *a* but it is a change for *b*. If this were a sensible program, the gcd would get used in some way, maybe just get printed. But this program is just to show how a circuit gets produced. To continue with main, it assigns *a* the value 12 and in parallel assigns *b* the value 30, and then calls gcd again. After that call, both *a* and *b* have the value 6, and again a sensible program would do something with that, but we stop there.

[190] The two variable declarations give us two 32-bit registers whose inputs come from the places where the variables are assigned. Variable *a* is assigned in three places. One of them is in gcd, and two of them are in main. So the data inputs for *a*, that is, *da*, come from those three places, and the control for *a*, that is, *ca*, comes from those same three places. The value of variable *a* is used in four places. One of them is in the **while** condition. One of them is in the **if** condition. One of them is in the expression *b* minus *a*, and the last one is in the expression *a* minus *b*. So the output from register *a* goes to those four places. Variable *b* also is assigned in three places and used in four places. So these are the circuits we get from the variable declarations.

[191] Here's the circuit we get from the declaration of procedure gcd. In the bottom left corner we get the control signal that starts it working. gcd is called from 2 places, so two of these paths coming in are from those places. I'll tell you where the other two paths come from later. In the previous picture we saw that *a* and *b* each go four places. Those are the unequal-to box, the less-than box, and the two subtraction boxes in this picture. gcd starts out by saying while *a* is unequal to *b*, so that's at the top left, and the result goes into a demultiplexer. If *a* and *b* are unequal, the control pulse comes out the **then** output. If

they're equal, the control pulse comes out the **else** output, and goes back to the calling places to say gcd is all finished. If *a* and *b* are unequal, then we see if *a* is less than *b*. And if it is, the control pulse goes to the top right part of the picture. This is the assignment statement *b* gets *b* minus *a*. You see *b* minus *a* from the subtraction box, and its result goes to the data input to variable *b*, and the control pulse goes to the control input for variable *b*. That's the paths labeled *db* and *cb*. And at the top right corner, we've just finished the **if** statement, and so we've just finished an iteration of the loop body, so the control output has to go back and restart the loop, which is the same place as the call that starts the whole procedure. So that's one of the inputs at the bottom left that I said I would tell you about later. I could have drawn the line going down and back, but I thought the picture would be neater if I just labeled it *gcd* because that's what the start of the procedure is labeled. If *a* wasn't less than *b*, then the control pulse went to the bottom right part of the picture, where we have the circuit for the assignment *a* gets *a* minus *b*. And from there, the control pulse also has to go back and restart the loop, so that's the final path of the four paths into the bottom left corner of the picture.

[192] Here's the circuit for the main procedure. It starts at the top left, where we have the circuit for the assignment *a* gets 3. The box with a 3 in it has no input and its output is 32 paths with constant values representing 3 in binary. They go to *da*, which is the data input to the register for variable *a*, and the control pulse goes to *ca* to cause the assignment *a* gets 3 to happen. That assignment is in parallel with the assignment *b* gets 27 in the bottom left corner. In general, when two things are in parallel, they might take different lengths of time, so their control outputs go into a merge, and when they are both finished we get a pulse out of the merge box. After that merge, the next thing is a call to *gcd*, so that's the path going upward to one of the inputs we saw on the previous picture. And when *gcd* is done, we get a pulse back down the path labeled *gcd* prime from the previous picture. So that's two pulses into the 1-2-merge in the right order. Way over on the top right corner of the picture, the pulse from gcd saying it's done also comes down the path labeled gcd prime from the previous picture, and into the 2 input of a 1-2-merge, but there, in the top right corner, there wasn't any pulse into the 1 input, so there's no pulse out the right side. Right now, the control pulse has just left the first 1-2-merge box in the middle of the picture, and it starts both the *a* gets 12 and *b* gets 30 assignments in parallel. When they're both done, *gcd* gets called again. When the pulse comes back from the gcd procedure to say it's done, once again it comes to both 1-2-merges, but only the rightmost 1-2-merge has the right combination of input pulses to emit an output pulse to say main is done.

So now we have a working circuit design, but the job is still only half done. The next phase is called optimization. What we have so far is what you get from applying the general case design patterns. But there are improvements that can take advantage of the specific program we're compiling. The course notes explain the process. Here, I'll [193] just show you the result. The circuit at the top gets simplified to the circuit at the bottom of the page by taking advantage of the fact that the expressions being assigned are constants. That's a smaller and faster circuit to do the same job.

[194] We've already covered circuits for addition and subtraction of integers. Here's how we do multiplication of integers. We start by declaring integer variables *a*, *b*, and *p*, and that gets compiled to 3 registers, which I won't bother to show you. The *mult* procedure multiplies the values of *a* and *b* together without changing the values of *a* and *b*. The result of the multiplication is the final value of variable *p*. The *mult* procedure begins by declaring two local variables *x* and *y*, so that's two more registers. *x* is initialized to the value of *a*, *y* is initialized to the value of *b*, and *p* is initialized to 0, and those three initializations happen in parallel. Then there's a loop. The first test is *y* not equal to 0, [195] so we'll need a box for that. But that's just an **or** gate. If any bit of *y* is 1, then *y* is nonzero. The body of the loop is three statements in parallel. In the first one, after the word **if**, that's *y* divided by 2 and

then take the remainder and see if that's equal to 1. In other words, if *y* is an odd number, then *p* gets *p* plus *x*. The test, [196] to see if *y* is odd, is just the rightmost bit of *y*. [197] To add *p* plus *x*, we need an adder. The middle statement says *x* gets *x* times 2, and that looks like we need a multiplier, but we don't have one. If we need a multiplier to build a multiplier, then we're out of luck. [198] Fortunately, multiplication by 2 is just a shift left. If you take all the bits of *x* and shift them left one place, that multiplies *x* by 2. We don't need any gates for that. All the bits of *x* are input, we just discard the leftmost bit, and stick a 0 on the right, and that's the output. The last statement is *y* gets *y* divided by 2. [199] Division by 2 is just a shift right. All the bits of *y* come in, but we discard the rightmost bit, and put a 0 on the left. [200] Here's the circuit. A pulse comes in *mult* on the left side to start the circuit working. In the bottom left corner you can see the assignments *x* gets *a* and *y* gets *b*. The assignment *p* gets 0 just needs the pulse to go to *cp*, which is the control input for *p*. It doesn't need all those 0s to go to the data input for *p*. Just on top of the first demultiplexer you see the **or** gate that is the test for *y* not equal to 0. If *y* is 0, the pulse goes out *mult* prime to say all done. If *y* is not 0, the second demultiplexer tests whether *y* is odd, and if it is, we have the circuitry for *p* gets *p* plus *x*. At the same time as that, down below it, we have the shift left and the shift right. You see bits 0 through 30 of *x* come in, and they go out as bits 1 through 31. And you see bits 1 through 31 of *y* come in and go out as bits 0 through 30. Then on the very right side, the merge says that when all that's done, we have to go back and start the loop again.

I won't show you a division circuit, or any circuits to do floating-point arithmetic, but they can all be built the same way. Just write a program, and compile it to a circuit.

[201] Now I want to show you how to build a computer. The way to build a computer is to [202] write a program, then compile it to a circuit. This is the first half of that program, and [203] this is the last half. That's just to show you how long the program is. Now I'm going back [202] to the first half to point out a couple of things. The top line defines the memory size. If we want a different memory size, just put a different number there. The next line declares an array called RAM. An array gets compiled to a memory, and that's the main memory of the computer. And right after it is a variable called AC. That gets compiled to a register: the accumulator register. The next line declares IR, which is the instruction register. The next line declares PC, which is the program counter. And the line after that is a condition code register called E. That's all the declarations. Now there's one procedure called execute. And in it there's one loop. Computer designers call it the fetch-execute loop. After a couple of lines you see it says fetch starts here. And then a couple of lines later it says execute starts here. The execute part is just a single switch statement, and that gets compiled to a general demultiplexer, with one case for each instruction. There's a little program for each computer instruction. There's LDA, which is the load accumulator instruction. Then there's STA, which is store accumulator. And so on. [203] The second page is just more instructions. And that's the whole program. We could execute this program on an existing computer, and we would be simulating the operation of the new computer. If we compile this program to a circuit, then that circuit is the new computer.

A computer is one of the most complex circuits that people have been able to design. That's because they have been designing circuits by deciding what gates they need and arranging them in the right way to produce the right effect. What they should be doing is writing programs and compiling them to circuits. Look at the length of this program: just 2 pages. That's a very short program. A commercial cpu might be a 10 page program. But a web browser program might be hundreds of pages long. We can design circuits to do much more complex and interesting tasks by writing programs and compiling them to circuits.

[204] So let's say you write a program. There are two ways to get your program executed. The usual way is to compile it to the machine language of some computer, and then execute it on that computer. [205] But now you know there's another way. You compile it to a circuit design, then you fabricate the circuit, and then power up the circuit. So which

way is better? [206] Well, the first way is certainly easier. You need a computer, which you have, and you don't need a circuit fabrication facility, which you probably don't have. [207] And it's less expensive, by a lot. [208] And you want to test the program, and debug it, and that's a lot easier the first way. [209] Even after you get the program right, you might want to add something to it, and [210] especially if that happens a lot, it's better to execute your program on a computer. On the other side [211] of the ledger, once you've got your program written and debugged, if it's going to be embedded in a car or a camera or any other piece of hardware where you're not updating it every week, you might want to turn your program into a circuit. And if it does get updated once a year, it's easy enough for a garage mechanic to pull out the old circuit and plug in a new one. But for most cars and appliances, it's never updated. This might change as cars become self-driving. We'll see. [212] A big advantage of compiling to a circuit is that the result is a hundred times faster. There's no memory bottleneck, and much more parallelism. So if speed is important, this might be the better way. [213] A circuit is much more secure than software because there's no way to subvert it. So if security is important, this is the better way. [214] And finally, if you need a circuit for a task that's too complex for traditional circuit design, you have to design it by writing a program and compiling it to a circuit. Even if you're perfectly happy to execute your program on a computer, the computer you execute it on is best designed by writing a program and compiling it to a circuit.

[talking head] I hope you found these lectures interesting, and I hope they increased your understanding of digital circuit design.

end