Introduction

Hi. I'm Professor Harry Porter, of Portland State University.

Recently, I built a computer out of relays and you can see it behind me.

In this video, I'm going to describe how it works and explain my design in quite a bit of detail.

So, let's get started.

What is a Relay?

So what exactly is a relay?

At the center of every relay is a coil of wire.

When you pass an electric current through a coil of wire, a magnetic field is created and it becomes an electromagnet.

Located next to the electromagnet is a small switch which will be operated by this magnetic field.

This switch is held in one position by a small spring.

When current is applied to the coil and the relay switches on, the magnetic field will move the switch into a different position.

When the relay is off and the field collapses, the switch will return to the original position.

All the relays in this computer contain double throw switches, which are also called on-on switches.

When the relay is on, electricity can pass through the double-throw switch along one pathway.

And when the relay is switched off, electricity can still pass through the switch, but it follows a different route.

All the relays in this computer are 4 pole relays, which means that they contain, not one, but 4 switches.

All 4 switches operate together, opening and closing in unison.

When the relay is on, all 4 switches are in this position.

When the relay is off, all 4 switches are in this position.

This is a picture of the actual part I used in my computer.

For simplicity, I used the same part throughout the computer, although not all contacts or switches are used in every relay.

You can see the contacts down below and you can see a plastic cover which encloses the coil and all four switches.

By the way the first computer bug was a moth that became trapped between the contacts in one of the switches in an early computer.

It got electrocuted and its carcass remained in place, preventing the switch from operating properly.

I should note that since my computer uses relays that all have this plastic case, I have built a computer that is, of course, immune to all such computer bugs.

Schematic Diagrams

There are several ways to represent relays schematically.

On the far left you see the traditional schematic diagram for a relay.

In the center is a representation in which the positions of the contacts accurately match the physical relay part I used.

This sort of diagram is particularly handy in wiring diagrams.

The diagram on the far right is something that I came up.

Its simplicity makes it ideal for use in complex circuit diagrams.

I show only one connection for the coil because, in this computer, each relay always has one coil connection tied to ground.

The connection to ground is implicit in all my diagrams.

Normally the relay is shown in the OFF position like this.

But sometimes I'll show the relay in the ON state, like this.

The NOT Circuit

Next, let's use a relay to implement logical negation.

On the right, you see a single relay implementing the logical NOT function.

When the input is low, the relay is off and the output is connected to power.

Our convention is that logical ONE is represented as a connection to power.

However, logical ZERO is NOT represented as a connection to ground.

Instead, logical ZERO is represented by a wire that is simply not connected to power.

Tying it to ground is unnecessary and, in the case of a design mistake, could easily result in a short-circuit.

When the input changes to a ONE, the relay switches ON and the output is then disconnected from power.

The OR Circuit

Next, let's look at an implementation of the LOGICAL OR function.

If either input is a ONE, then the output should be a ONE.

On the right, we see an implementation using 2 relays.

Each input will power one of the relays.

If either relay is turned on, then the output will be tied to power.... And the output will go to a logical ONE, as it should.

Now you might suggest that a much simpler circuit would work just as well.

On the right, we've implemented the same circuit with zero relays.

This is called a WIRED-OR.

If either input is tied to power then the output will be tied to power.

The WIRED-OR circuit will work in many cases, but you have to be careful.

It won't always work.

Consider what happens if you try to implement the circuit shown on the left, using the WIRED-OR technique.

Consider the case when D is HIGH and the other two inputs are LOW.

The C-OR-D output should be high, but the B-OR-C output should be low.

But unfortunately electricity has the nasty property of being able to travel down a wire in both directions.

What's happening here is that the D input is traveling back out through the C input.

In other words, a signal is traveling through the wire in the wrong direction.

I called this problem BACK-FLOW.

Here is a version of the same circuit, using just one relay to avoid this BACK-FLOW problem.

Sometimes, you can get away with a solution using WIRED-OR, and this can save a relay or two.

But you’ve got to be careful that BACK-FLOW doesn't cause some other circuit to malfunction.

The 1-Bit Logic Circuit

Next, let's design a circuit to implement the common logical functions of NOT, AND, OR, and EXCLUSIVE-OR, as function of 2 inputs, which we'll call B and C.

The inputs are on the left and the 4 outputs will come out the right side.

TO avoid BACK-FLOW out of this circuit, we'll only use the inputs to drive the relays.

This circuit is kind of complex, so let's start with just logical negation.

Here is the NOT circuit that we looked at earlier.

Notice that LOGICAL-NOT is just a function of the B input; C is ignored.

Next, let's add in the LOGICAL-AND.

Both relays must be ON in order for the output to be connected to power.

Let's show what we've got so far in gray, to keep things simple.

Now, we can add in the circuit for LOGICAL-OR.

And finally, we can show the circuit for the EXCLUSIVE-OR function.

The output will be LOW if both inputs are low and both relays are OFF as shown.

If either relay then switches ON, then the output will go high, but if both switch ON, then the output will again be LOW.

Here’s the completed circuit.

We'll call this the 1-BIT LOGIC CIRCUIT.

We can combine 8 of these circuits to create an 8-BIT LOGIC CIRCUIT.

There are still 2 inputs, but now each input is an 8-bit quantity, instead of just a single bit.

The output is also an 8-bit quantity.

The Addition Circuit

Next, let's look at the circuitry for performing addition.

Do you remember the definition of the FULL-ADDER circuit?

It operates on 1 bit in a multi-bit addition.

It takes 2 inputs, which we'll again call B and C, as well as a CARRY INPUT from the previous stage in the addition.

It produced 2 outputs.

One is the SUM. This is the output for this bit in the addition.

The other is the CARRY OUTPUT, which is fed into the CARRY INPUT for the next stage.

On the left is the truth table for the full adder and on the right is a simplified diagram for the circuit.

I'm not going to show the details of this circuit, but it is similar to the 1-bit logic circuit we just looked at.

In the 1940s, Konrad Zuse in Germany came up with a clever design for a full adder using only 2 relays. His circuit for the FULL ADDER is the only circuit in my computer that I did not design myself.

So now let's take 8 of these full adder circuits and combine them to form an 8-bit adder.

Again, we have two inputs, called B and C.

Each input is 8 bits and the output is 8 bits.

A subscript of zero is used for the least-significant bit.

Notice that we are feeding in a ZERO as a carry IN to the full adder at the least significant end of the adder.

On the other end, the circuit produces a carry out from the entire addition.

This CARRY OUT at the upper end could be used, for example, in testing for overflow or for performing larger additions.

For example, a program for this computer could use the 8-bit ADD instruction repeatedly to achieve the result of a 64-bit addition.

We can summarize the entire 8 bit adder circuit using the traditional circuit diagram for an adder, shown below.

Up to now, each line represented a single wire.

Here, the heavy lines (with little diagonal marks labeled 8) stand for 8 parallel wires.

This technique of showing multiple parallel lines will simplify our drawings in the future.

The Zero-Detect Circuit

From time to time it’s useful to know when a value is zero.

This circuit can detect when a value is zero, namely when none of the lines is high.

Along the bottom we see 8 wires, called the RESULT BUS.

This circuit detects when a result (such as the result of an addition) is all zeros.

The output is labeled ZERO and will only be high when all relays are OFF.

We’ll also be interested in whether an 8-bit result is negative or not.

Recall that when representing integers in binary, the most significant bit is called the SIGN BIT.

We don't need any relays to compute the sign of a number.

We can just peel off the most significant bit and label it the SIGN output.

If the sign bit is ONE, the value is negative.

If it’s ZERO, the value is positive, or possibly zero.

The Enable and Shift Circuits

This next circuit to look at is called the ENABLE circuit.

It contains 2 relays, and both are driven by a control line, which is labeled ENABLE.

When the ENABLE input is high, the relays turn on and the circuit allows an 8-bit value to flow through, say from top to bottom.

For example, the ENABLE circuit might be used to allow an 8-bit value to flow onto a bus.

Or it could be used to allow the value to flow, from the bus to the other circuit.

When the ENABLE circuit is off, as shown here, the bus is isolated from whatever circuit is connected.

For example, we can use an ENABLE circuit to connect the EXCLUSIVE-OR output from the 8-bit logical circuit, which we talked about earlier, to a bus which we will now call the RESULT BUS.

If we want, as our result, the EXCLUSIVE-OR function, we can drive the ENABLE line high and the EXCLUSIVE-OR output will then be connected to the bus.

This diagram also shows 8 lights connected to the exclusive-or lines.

These lights monitor the exclusive-or output, regardless of whether it is connected to the RESULT BUS or not.

The EXCLUSIVE-OR is being computed continuously and is being displayed continuously, regardless of whether the value is being used or not.

I've used the traditional circuit notation for a LAMP here rather than for a light-emitting diode, although this computer actually uses LEDs.

The actual part that I used is meant to be used as an indicator light and consists of both the diode and a small resistor, which are combined into a single, small plastic unit.

All LEDs are wired the same way: one terminal is connected to a wire whose current value we're interested in seeing and the other terminal is connected to ground.

Obviously, the LEDs do not affect the computation; they merely allow us to see it and observe what’s happening.

LEDs are critical.

Every computer contains a lot of internal state and, without them, it would be impossible to know what’s going on inside the computer.

We've talked about ADDITION and the logical operations of AND, OR, NOT and EXCLUSIVE-OR.

While we're at it, let's also show how we compute the shift function.

It easy; we don't need any relays, except the 2 used in the enable circuit.

What we are doing is simply shifting each bit of B left by one place, and taking the most-significant bit back to the other end.

Although this computer contains just this one shift operation, to achieve the effect of shifting by several places, a program can simply just repeat the shift instruction several times.

For example, shifting to the left 7 times is the same as shifting to the right one position.

This exemplifies a general philosophy underlying the design of this computer.

One goal is to implement a set of instructions that is representative of typical computers and that's complete enough to allow any program to be coded fairly easily.

But most computers contain a whole bunch of other stuff that's thrown in just to make the computer fast at common tasks.

For example, most computers would contain a number of different shift instructions, so that any desired shift could be achieved in one instruction, rather than needing several instructions.

Including a few more shift operations would add nothing very interesting to the computer, so I just didn't do it.

The 3-To-8 Decoder

A common class of circuits is called the DECODER circuits.

For example, take a look at a 3-to-8 DECODER.

You can also have a 2-to-4 decoder or a 4-to-16 decoder, but here we've got a 3-to-8 decoder.

It has 3 inputs, which can be interpreted as a binary value between 0 and 7.

It has 8 output lines, and exactly one of them will be driven high, as selected by the inputs.

Here is a circuit implementing the 3-to-8 decoder.

Each input drives one relay and you can see that exactly one of the 8 outputs will be driven high, depending on the states of the relays.

Our ARITHMETIC LOGIC UNIT is capable of computing several different functions.

We use a 3-to-8 decoder to select which function is desired.

The function codes--which are labeled f-zero, f-one, and f-two--will select one of the 7 possible functions that this ALU can compute.

The eighth line is used when NO function is selected.

It’s essentially the ZERO function: When the eighth combination is supplied as the input to the decoder, the ALU will be disconnected from the result bus.

Remember that we are representing ZERO as “disconnected from power”, so disconnecting the ALU from the output lines is exactly the same as having the ALU produce zeros on its output lines.

The ALU

Now we're ready to look at the entire arithmetic logic unit, the ALU.

The inputs are shown in the upper left.

There are the two 8-bit input values, called B and C and there is the 3-bit function code input, which selects which function will be output.

The function code drives a 3-to-8 decoder, which in turn will drive one of 7 ENABLE units.

The 8-bit logic unit is producing the AND, OR, EXCLUSIVE-OR, NOT and SHIFT outputs at all times concurrently.

If the function code is FOUR, for example, then the FIFTH enable unit will be selected. And the exclusive-or output will be gated onto the data bus.

The ALU puts its result on something called the DATA BUS, which is an 8-bit bus that runs throughout the computer and it connects to all the registers.

We also see the SIGN, the CARRY, and the ZERO outputs.

These 3 bits are called the condition codes and, later, these will end up getting loaded into a 3-bit register, which is named the CONDITION CODE REGISTER.

For the 8-bit adder we see the SUM output, which is labeled ADD in this diagram, and we also see an output labeled I-N-C, which stands for INCREMENT.

The INCREMENT output is just ONE added to the B input, with the C input being ignored.

Although I didn’t describe exactly how the increment function is computed, it’s fairly straightforward.

Basically, we block the C input to the adder, using an enable-like circuit and then we feed a ONE into the CARRY input to the least significant bit.

We can summarize this rather complex diagram with this simpler diagram.

This shows the inputs to the ALU--the two 8-bit values called B and C and the 3-bit function code, and it shows the 8-bit result, which is gated onto the result bus.

We also see the 3 condition code outputs.

Registers

So now let's ask how we can store a value in a register.

All computers contain a few registers in their CPUs and each register is a little memory unit that can store bits over time.

To simplify our diagrams, let's start with a single bit.

This circuit clearly has a memory, of sorts.

If the line labeled A is ever driven high, the relay will switch on.

At that point, a feedback loop will prevent the relay from ever turning off, at least until the power is turned off.

If the power is turned off, it will effectively erase the register, returning it to zero.

So we can take 8 of these relays and build an 8-bit register out of them.

Now, let's connect this register to the data bus using an enable circuit.

By driving the SELECT line high, we can gate the value currently stored in the register onto the bus.

The SELECT line allows us to read the value of the register, in the sense of putting it out onto the bus, where other parts of the computer can pick it up, if they want.

But we still have the problem of loading the register with a new value.

Now we have added a couple of inverters. Let's see how this works.

When LOAD is low, power is supplied to the register and it will retain its value.

What happens if LOAD goes high?

First, the output of the first inverter will go low, causing all the bits in the register to be reset to zero.

Then the second inverter's output will go high, which will open up the enable circuit.

This will allow the value on the bus, whatever it is, to flow into the register and some of the bit-relays will switch back on.

Then, let's look at what happens when the LOAD line returns to low.

The output of the first inverter will return to HIGH, re-supplying power to the bit relays and latching the value into the register.

Then the output of the second inverted will go low, turning the ENABLE circuit off and disconnecting the register from the bus.

To simplify things, let's draw a box around the control circuitry, which controls the loading and selecting of the register.

This separates it from the bit relays, which actually store the data.

Simplifying even further in this diagram, we are showing the LOAD and SELECT control lines going into the control circuitry and we are showing the 8 bits of the register as little boxes that might contain either a ZERO or a ONE.

The computer contains 8 general purpose registers.

All 8 of these 8-bit registers are connected to the DATA BUS in the same way.

Each register has its own LOAD and SELECT control circuitry.

Each register has a name. One register is called X and another is called Y.

There are also registers named A, and B, and C and there are a couple others, too.

In addition to the 8-bit data bus, there is also a 16-bit bus called the ADDRESS BUS.

These are the two main busses in the computer: the DATA BUS and the ADDRESS BUS.

As in many computers, some registers can be combined into larger registers.

In this computer, for example, the X and Y registers are each 8-bit registers, but in some instructions they can be treated as one 16-bit register.

When treated as a single 16-bit register, they are called the XY register.

There is also control circuitry that allows the combined register to be loaded from, or selected onto, the address bus.

Overall Architecture

Now we are ready to look at the overall architecture of the computer.

Let's start in small pieces.

First, we have the 8 general purpose registers, which are named A, B, C, D, M1, M2, X, and Y.

Each register can be loaded from the data bus and each can be selected onto the data bus.

The little 2-way arrows between each register and the data bus indicate that data can be moved both ways, both LOADED from the bus into the register, and SELECTED from the register and put out onto the bus.

Thus, there are 8 LOAD lines and 8 SELECT lines to control this data movement, at least for what we've shown so far.

So now let's add in the arithmetic logic unit.

The ALU always takes its inputs from the registers named B and C and its result is always gated back on to the data bus, where it can be loaded into one of the other registers.

We've also included a 3 bit register called the condition code register over on the right.

It is only LOADED from the ZERO, CARRY, and SIGN outputs from the ALU.

Next, let's add in the 16-bit address bus.

The XY register pair can be either LOADED or SELECTED, so the data can move both directions to the address bus.

The two registers named M1 and M2 can also form a 16-bit register, which is called M.

The combined 16 bit register can only be loaded in parts from the DATA BUS; it cannot be loaded as a whole from the ADDRESS BUS.

When we talk about the instructions, we'll see that different registers are used differently and we'll see that there is no need for loading M all at once.

Next, let's add in the instruction register and the program counter, the PC.

The instruction register will contain the instruction currently being executed and will be loaded from the 8-bit bus.

Each instruction is 8-bits in length.

While it appears in this diagram that data in the instruction register can never be used, in fact this register will drive the instruction sequencing circuitry.

We'll discuss sequencing later on.

This diagram only contains lines carrying data; it does not show the control lines at all.

The PROGRAM COUNTER will only be accessed as a full 16-bit entity and is not connected to the data bus at all.

Next comes the MAIN MEMORY.

The memory takes a 16-bit address as one input.

During a STORE operation, when a byte is moved from the CPU to the MEMORY, it also takes an 8-bit data value from the DATA BUS and stores this in its memory.

During a LOAD operation, the data flows in the other direction, from the MEMORY to the DATA BUS and from there it can flow into one of the registers.

Next, let's look at the 16-bit increment circuit.

This increment circuit is completely separate from the increment portion of the adder in the ALU.

There is also an associated 16-bit register, which is called INC, the INCREMENT register.

The increment unit takes its input from the address bus,--- whatever value is on the address bus-- and adds one to it.

The incremented value is then fed straight into the INC register.

So when the LOAD line to the INCREMENT unit goes high, the INC register is loaded with one plus whatever is currently on the address bus.

And when the INCREMENT SELECT line goes high, the value in the INC register is fed back out onto the address bus.

Finally, there is a another register, called J, which is used during the JUMP instructions.

It is loaded in 8-bit pieces from the DATA BUS, but is used as a full 16-bit value on the ADDRESS BUS.

The Memory Unit

Here is a close-up photo of the main memory unit.

The memory in implemented with a single chip, which can hold 32 kilo-bytes.

I could have implemented the main memory using relays, using circuitry like the register, but it would have made the whole computer really large.

Building it would have consumed a huge number of relays as well as a huge amount of time, money, and patience.

Furthermore, the circuitry would have been exceedingly repetitive.

With 16 bits of address, the computer can actually handle up to 64 kilobytes, but 32 is more than enough.

The chip outputs are not powerful enough to drive a relay, so I used an array of 8 power transistors to step up to the voltage and current required by the relays.

The memory chip is a 5-volt part; the relays and LEDs all run on 12 volts, DC.

The memory chip looses all its contents when the power is turned off.

The only way to load the memory, is by flipping switches by hand, bit by bit.

Its a little tedious to toggle in a long program, but its not too bad.

An Example: The ADD Instruction

So now let's take a look at how an instruction executes.

We'll walk through the execution of an instructions that uses the ALU, for example the

ADD instruction.

The first step its to fetch the instruction from main memory.

We drive the SELECT line to the PROGRAM COUNTER high, which allows the address of the next instruction to flow onto the address bus.

Then we drive the READ line to the memory HIGH, which causes the memory look at the address bus and fetch the byte at that address and put it onto the data bus.

Then we drive the LOAD line to the INSTRUCTION register high, which loads it with that value.

Now the instruction register contains the instruction that we are going to execute.

Before we execute it, we need to increment the PC.

So we SELECT the PC, which puts its address onto the bus and we drive the INCREMENT REGISTER's LOAD line high.

This loads the INC register with whatever was on the address bus, plus one.

But we will need to get this value back into the PC.

So now we will SELECT the INCREMENT register and LOAD the PC.

This transfers the incremented address into the PC.

The PC has been incremented by 1 and now we're ready to execute the instruction.

The instruction decoding circuitry, which we haven't talked about yet, will now look at whatever is in the instruction register and the control lines will be driven HIGH and LOW in whatever ways are appropriate for the instruction being executed.

The ALU is always getting its inputs from the B and C registers although normally the NOP code is sent to the ALU so it doesn't put anything on the data bus.

But we are assuming this is an ADD instruction, so the instruction decoding cicuitry will now send the code for ADD to the 3 function code inputs to the ALU.

The ALU will respond by computing the sum and gating it onto the bus.

The condition code bits will also be computed.

In the next step, the LOAD line for one of the registers--let's assume it is A--will be driven high.

Also, we'll drive the LOAD line for the condition code register high, so it will get loaded with bits reflecting the result of the addition.

So, we've added registers B and C, putting the result in register A and setting the condition code register to reflect the result.

The Clock Circuit

During all this, control lines were going high and low over time.

These lines need to remain high for a short interval of time, before going low.

So next we need to look at the clock, which regulates this timing.

Take a look at this circuit, which contains a switch, a relay and a capacitor.

When the switch is closed, the relay will turn on and the output will go high.

Also, the capacitor will charge up and this happens pretty quickly.

If you're not familiar with capacitors, they're like little batteries: you can charge them up and then they can be used to power something--like a relay--for a short time until they are exhausted.

However, batteries store a lot more energy over much longer periods of time.

So next, if we open the switch, the relay will stay on because the capacitor will power it.

The capacitor will discharge in about a quarter of a quarter of a second and then the relay will turn off.

Let's look at that again: The switch is closed; the relay turns on and the capacitor charges; the switch is opened, then there is a short delay and finally the relay switches off.

The key is that the capacitor introduces a short, well-defined DELAY.

It is this delay that we'll build our clock out of.

Now take a look at this repetitive chain of relays and capacitors.

Let's start by assuming that relays B and C are on, like this.

We'll assume that B is on because it is powered by its capacitor, which is slowly discharging.

C is on because there is a path from power to its coil.

Also, C'c capacitor is also getting charged up.

At some point, B's capacitor will run out and B will switch off.

This will cause D to switch on, because there is now a path from power to D's coil.

C will remain on, but the path from power to C's coil was broken when B switched off, so C will stay on because it is now powered by its capacitor.

So now let's connect relay D back to relay A.

When C's capacitor finally runs out and C switches off, then A will now switch on.

So we've got a cycle and this will just keep going.

Let's look at one more step...

The clock circuit is rather complex, but we can understand it a little better by looking at a TIMING DIAGRAM.

Here we are showing a line for each of the 4 relays in this circuit.

In this diagram, time flows left-to-right.

We can see when each relay goes on and when each goes off.

We can also see, that at each moment, two relays are on and two relays are off.

Now our goal is to compute a nice clean clock pulse.

In particular, we want a SQUARE WAVE like this.

We can compute or derive this clock pulse as a simple function of the 4 relays.

By utilizing the unused switches on the relays, we don't even need any more relays to compute the square wave.

Here you see the 4 clock relays with their capacitors above them and over at the right you see the square wave clock pulse.

The Finite State Control

Now, let's look at a FINITE STATE MACHINE.

Recall that a finite state machine has a number of states, shown by the circles, and transitions between the states, shown by the blue arrows.

In this particular machine, there are no choices.

We just go from one state to the next, around in a circle.

The clock pulse--the square wave from the clock circuit--will drive the finite state machine.

On each clock transition--either from low to high or high to low--the machine will move from one state to the next.

We can even add some output lines.

The idea is that when the machine is in state 1, the corresponding output line is high.

So as we move from state to state, each output line goes high, in turn.

At any one time, only one output line is high.

We'll call these output lines, the TIMING LINES, and we'll label them t-1 though t-8.

Here is a timing diagram showing how the timing lines rise and fall over time.

Most of the instructions will require 8 units of time to execute.

Remember the ADD instruction we walked through?

That could be done in 8 steps, 8 time units.

Timing Diagrams

To see how an ALU instruction--like the ADD instruction--can be performed in 8 steps, let's look at a timing diagram for the control lines.

This shows exactly when the various control lines need to rise and fall.

By the way, each control line can be operated by a switch on the computer's front panel.

So you could stand in front of it and flip the switches according to this diagram and the instruction would be executed.

But when the machine is running and the clock is driving things, the control lines will rise and fall on their own.

So let's go through this diagram in more detail.

First, we will need to get the instruction from memory.

This step is usually called the FETCH step.

We drive the SELECT line for the PC high and we drive the control line to the memory HIGH to make it load a byte onto the data bus.

We wait one time unit, to make sure the operation has time to complete and the memory output is stable, and then we drive the LOAD line to the INSTRUCTION register high, loading it.

Next, we need to increment the PC.

This is done in two parts: first we load the INCREMENT register.

Since the PC is being SELECTED onto the address bus at this time, the INCREMENT register will be loaded with that value, plus one.

Then later, we SELECT the INCREMENT register and LOAD the PC.

Note that the first part is overlapped with the FETCH step, since the PC is already SELECTED onto the ADDRESS BUS anyway, and loading the INCREMENT register can be done in parallel with loading the INSTRUCTION register, since they both use different busses.

The final step is usually called the EXECUTE step.

In the case of an instruction that uses the ALU, we need to first send the 3-bit function code to the ALU.

The ALU function code comes straight out of 3 of the bits in the instruction itself, from 3 bits of the instruction register.

Then, after giving it a 1 unit delay for the result to become stable, we can load the target register--in this case the A register--and the condition code register.

Instruction Decoding

Now we need to look, in a little greater detail, at how these control lines are generated.

This is the process called instruction DECODING and instruction SEQUENCING.

So we got, as our inputs, the outputs from the finite state machine, and the current value of the instruction register.

The values from the finite state machine tells us, more-or-less, what time it is.

In the case of the FETCH and INCREMENT portions, all we need to know is, what time it is.

If, for example, it is time t1, t2, or t3, then the SELECT PC control line needs to be high.

In the case of the EXECUTE portion of the instruction, we need to know what time it is and what instruction it is.

In any case, to compute the control signals, we just need a bunch of COMBINATIONAL LOGIC.

Recall that COMBINATIONAL LOGIC is just functional... there is no state: The outputs are just a simple logical function of the inputs and past states don't matter.

The other kind of logic is called SEQUENTIAL LOGIC, which involves state and memory.

We don't need to work through all this combinational circuitry, but it does consume quite a few relays.

I guess I'll leave this as an exercise for you to design.

The combinational logic also uses, as inputs, the current value in the condition code register, but I didn't show this, since there was quite enough room in the diagram.

Now, so far we been assuming that all instructions take 8 units of time, but this is not true.

Some instructions take longer.

For example, there is a 16-bit move instruction.

Since it needs to use the address bus to transfer the data, it has to wait until the increment portion has completed.

Here is its timing diagram.

You can see it needs 10 time units.

So the simple finite state machine with only 8 states was not the full picture.

The actual finite state machine has 24 states.

Most instructions use fewer states and execute faster, but some instructions require all 24 states and therefore take much longer to execute.

We can view the finite state machine as a sequence of states--from 1 to 24--with 4 additional transitions that take a shortcut back to state 1.

So there are actually some feedback line from the combinational logic in the instruction decoding back to the finite state machine.

There is some logic in the instruction decoding that figures out how many states each instruction requires and sends signals back to the finite state machine, which cause it to take one of those short-cut transitions if the instruction doesn't need the full 24 time units.

The Instruction Set

Now let's take a look at the instructions in more detail.

The first instruction is a register-to-register MOVE instruction.

It moves 8-bits from any register--the SOURCE register-- to any other register--the DESTINATION register.

The source and the destination are each specified using 3 bits in the instruction.

If the source and destination happen to be the same register, then it will get set to zero.

The next instruction is the ALU instruction.

A 3 bit field in the instruction provides the function code, specifying whether it is an ADD instruction, an INCREMENT instruction, or whatever.

The result can be put into either the A or the D register, as specified by the bit labeled R.

The next instruction loads a constant value into either the A or B register, as specified by a bit in the instruction.

The value is given right in the instruction, in a 5-bit field.

This value is SIGN-EXTENDED, which means the most significant bit is replicated to fill the remaining high-order bits.

This SIGN-EXTENSION will take these 5 bits and turn them into a value between -16 and +15.

The last instruction shown here is an instruction to increment the XY register pair.

The next instruction to look at is the LOAD instruction.

It uses the 16-bit value in the M register as an address.

This address is sent to the memory and the byte read out of the memory will be loaded into either the A, the B, the C or the D register, as specified by a 2 bit field in the instruction.

The next instruction is the STORE instruction.

Like the LOAD instruction, the address is contained in the M register.

The data is moved from a register -- either A, B, C, or D -- to the main memory.

The next instruction loads a 16-bit value into the M register.

This is handy for getting the address into M before a LOAD or STORE instruction.

In this instruction, the op-code is 8-bits like in the other instructions, but the instruction is followed by 2 additional bytes which give the 16-bit value.

This instruction takes a full 24 time units, because the PC has to be incremented several times and we have to go to memory 3 times: once to get the instruction, then once for the next byte, and then one more time for the final byte.

The last instruction on this diagram is the HALT instruction.

There is one relay, which I haven't mention before, that will get set when this instruction is executed.

This relay will latch on and stay on, until you flip a switch to cut its power and turn it off.

This relay, when on, will disable the clock and will have the effect of freezing the machine.

So the execution of the HALT instruction will freeze the clock in its tracks and stop instruction execution dead.

The next instruction is the GOTO instruction.

This instruction is a lot like the instruction that loads the M register, except that it loads the PC register instead.

The 8-bit instruction is followed by a 16-bit value that is the absolute address to jump to.

The jump occurs when the PC is loaded with a new value; the next instruction to be fetched will come from this new address.

This is the way GOTO instructions usually work.

The first part of the instruction execution does the FETCH and the INCREMENT.

Then, if it is a GOTO instruction, the PC is overwritten with a completely new value.

Then the next instruction's FETCH part will use this modified value.

One thing to note about our GOTO instruction is that the PC is needed throughout the execution of the instruction.

We can't load the PC until the very end.

We need to increment the PC, then fetch the next byte, then increment it again, and then fetch the last byte.

So consequently, we can't load the PC until we have finished fetching the entire 16-bit value.

This is the reason for the J register.

The J register is where we put this value as we fetch it from memory.

Only at the end, do we transfer the bits from the J register to the program counter.

The next instruction is the CALL instruction.

It is very similar to the GOTO instruction.

Like the GOTO instruction, the CALL instruction fetches a 16-bit value and ultimately moves it into the PC register, causing a branch in the flow-of-control.

Like the GOTO, it moves it into the J register first, before moving it into the PROGRAM

COUNTER.

But before the final move, we save the value of the PC in the XY register.

This value of the PC is the value after all the incrementing, so it points to the first byte right after the last byte of the instruction.

This is exactly the place we want to return to, after the subroutine is executed.

This is the return address, and as part of any CALL instruction, the return address needs to be saved.

In this computer, the return address is saved in a register--the XY register--not on a stack.

This computer doesn't support a hardware stack, like most computers do.

Most computers have a register dedicated to pointing to a stack and there are instructions that push and pop things off the stack.

Usually, a CALL instruction will push the PROGRAM COUNTER onto the stack, which means storing it to memory and incrementing or decrementing some register.

A stack in memory is a good idea because it facilitates recursive functions and doesn't consume a register to hold the return address.

However this computer saves the return address in a register--the XY register--and not in memory.

Of course, you could easily program a stack.

It would take several instructions, but you could do it.

And you'd have to, if you wanted to implement a recursive routine.

You could dedicate a register to holding the stack pointer, but since regsiter storage is so limited in this computer, you'd probably want to put the stack pointer in memory, too.

In any case, if all you want is a quick little subroutine that doesn't call other routines, the CALL and RETURN instructions in this computer will work quite nicely.

I should note that there seems to be a trend of going back to the idea of saving the return address in a register.

The SPARC architecture does it, for example.

The benefit is that subroutine CALLS and RETURNS can be faster if you don't have to go to main memory.

And most all routines are not recursive and many don't even call any other routines, so this is a very real gain.

Anyway, we also need the RETURN statement and that is shown next.

All it does is move the saved value from XY back into the PC register.

This instruction can also be used to implement arbitrary jumps--indirect jumps, where the address is computed and stored in a register.

This would be useful for implementing, say, the SWITCH statement in a high-level language or for invoking closures or something like that.

The RETURN instruction is actually a special case of a more general 16-bit MOVE instruction, which is shown at the very bottom.

The destination register can be either the PC or the XY register.

The source can be either the M, the XY, or the J register.

If the destination is the PC and the source is the XY register, then you just have the RETURN instruction.

Next, we have some conditional branching instructions and these are shown here.

They are all very similar, and very similar to the straight GOTO instruction.

They each load the J register with the 16-bit value that follows the instruction byte.

Then, in the last step they move that value into the PC, CONDITIONALLY.

If the value is moved, the jump occurs.

If the value is not moved, the branch is not taken.

There are several different tests possible.

The first will do the branch--that is the PC will be loaded--if the SIGN bit in the condition code register is 1.

In other words, if the result of the last ALU instruction was negative--so the SIGN bit was set--then this instruction will take the branch.

The second instruction will branch if there was a carry out of the most significant bit in an

ADD instruction.

The last two instructions test whether the ZERO bit is set.

In other words, they test whether the result was all zero bits.

The first branches if it was zero; The last one branches if it was anything else.

You can use these last 2 conditional BRANCHes with an EXCLUSIVE-OR instruction to test whether 2 values are equal.

If you EXCLUSIVE-OR two values together, you'll get zero if and only if they were the same value.

An Example Program

So finally, we can show a complete program.

This is probably a little hard to read, so let me walk you through it.

On the far left, we see the memory addresses.

I'm only showing 8-bits of the address, not the full 16-bits, since there's not room and the program is so short all the upper bits are zero anyway.

Then you see the instructions.

First, you see the instruction in binary.

These are the binary machine instructions--not very easy to read and even harder for the poor programmer to get right in the first place.

If you were to go back, you could check all these bits against what I just finished saying about the instructions and there would be exact agreement.

These are the bits that you would switch into the memory before executing the program.

The next column gives an assembly code version of each instruction.

I have not written an automated assembler for this machine--there is no real need actually--although other people have.

All my programs have been hand assembled.

I guess I find this more to my liking.

The last column gives another, higher-level version of this code.

These are roughly equivalent to the comments that you would see--or at least OUGHT to see--in any assembly language program.

If you look at these comments, you could see what this program is really doing.

And what is it doing?

Well, we see a loop.

Each execution of the loop increments the D register and tests to see if it is zero.

Since D is first loaded with -7, the loop body will execute 7 times.

Then the program will halt.

What happens in the loop?

Well, we see Y getting shifted 1 bit on each iteration and tested.

Then, if the bit is 1, it does something.

So basically, we are looking through Y and every time we find a 1 bit, we do something.

And what do we do?

We add C to X.

Also, we can see that on each loop iteration, we are shifting X 1 bit as well.

So apparently, some sort of answer is being computed into X.

The stuff before the loop does the same thing, testing the first bit in Y and initializing X to either zero or C depending.

Maybe you recognize this pattern, maybe not.

What we're doing here is binary multiplication.

We are multiplying two 8-bit numbers.

One number is Y.

We're looking through Y and every time we see a 1, we're adding C in to the answer.

Normally, in multiplication, we would shift C one bit position to the left, then possibly add it to the result, then shift it again one more bit to the left, and then possibly add it again, and so on.

We're doing something similar here: constantly shifting the answer instead of shifting C, but this has the same effect, because the shift is a circular shift.

So I told you earlier that this computer has all the simple, important, basic instructions, like ADD, but that some of the more complex instructions were left out.

The idea is that you could always write a program to do the more complex stuff and this is an example.

There is no MULTIPLY instruction, but this program shows that you can certainly write a program to perform multiplication.

This makes two important points.

First, is that this computer, like all other computers, has full TURING power.

In other words, this computer can IN THEORY do anything that any other computer can do.

It can do anything a TURING MACHINE can do.

All computers, IN THEORY have the same power and can compute the same class of functions.

It is a TURING MACHINE in exactly the same sense that every computer is a TURING MACHINE.

The second point is that the phrase IN THEORY is very, very important.

This computer can do an addition in well under a second, but it takes minutes to execute this program to perform even a simple multiplication.

Adding a MULTIPLY instruction to the computer would obviously make a huge difference to any program that needed to multiply two numbers.

So the point is that INSTRUCTION SET DESIGN really, really matters.

Computers with carefully thought-out architectures will out-perform poorly designed machines, even with reasonable differences in clock speed and in implementation technology.

Conclusion

I've gone out of my way to create a computer that’s really simple and uses only 415 relays.

I wanted something that would show the main, core ideas inside every modern CPU out there today, but that wasn't too complicated.

I hope that in this video, I've been able to explain how a computer works, in a way that is complete and comprehensible.

If you'd like to go further with these ideas, there is an accompanying paper on the website that goes into the design in greater detail.

And perhaps you'd like to build your own computer.

I can’t encourage you enough.

This has been a fabulously rewarding project for me and I have learned a great deal in doing it.

So I would say, go ahead. Design your own machine and build it!

I'm sure that, regardless of what you end up designing and building next, you'll have fun and learn a lot from the experience, like I did from my relay computer.

I hope this video tutorial has been informative and I welcome any feedback.

I’m Harry Porter. Thanks for watching.