Pegasus 42 Project

The goal for this Gambiconf 2025 workshop is to show how the most basic elements of digital computers and videogame work and how they can be combined to create more complex blocks up to a whole computer.

A simulator (called Digital) will be used to allow the workshop participants to experiment for themselves with all the presented circuits.

1. Switches, Logic Gates, Combinational Circuits

These are the basic blocks of digital circuits.

2. Sequential Circuits

More complex circuits need memory.

3. Processors

mcpu16h is a very simple processor with only 4 instructions and is a good introduction while drv32h is more practical, being compatible with the RISC-V standard.

4. FPGAs and Shin JAMMA

With reconfigurable circuits (FPGAs - Field Programmable Gate Arrays) it is possible to see the simulates circuits working in the real world. As there are many different FPGA boards, the Shin JAMMA standard was created so a single project can work on any of them

5. Video and Audio

Though interesting games can be created to use the text terminal, color graphics output and sound like those from the classic NES (Nintendo Entertainment System or Famicom) allow games that are more fun

6. Pegasus 42

With cache memory and high resolution graphics it becomes possible to run the Squeak Smalltalk programming language

A. History

In addition to the material presented in the workshop, a history of the Pegasus project is also included as an appendix.

1. Switches, Logic Gates, Combinational Circuits

The goal of this workshop is to build a retro computer (roughly equivalent to what people would have bought in the early 1990s for their home) called Pegasus 42 and use it to program simple games. The idea is that the project can be completely understood from the lowest level to the overall system.

For this we will initially use a simulator, and then move on to a FPGA (Field Programmable Gate Array - a chip which can be reconfigured to implement any digital circuit). For each level of abstraction there are several different simulators we can use. From a high to low level we have:

Level	Example Simulators
Architecture	QEMU, MAME
Micro-architecture	SPIM, SimpleScalar
Register Transfer	Verilator, ModelSim
Logic Gate	Digital, TkGate
Switches	IRSIM, MOSSIM
Analog Circuits	Spice, Xyce
Components	TCAD, DEVSIM
Physics	Elmer, Matlab

The advantage of using a simulator instead of the real thing is being able to see details that would be very hard, if not impossible, to measure in the actual circuit. The low level simulators show far more details than the high level ones, but are propotionally slower when running on the same computer. So while it would be possible to simulate a whole computer using Spice, it might take minutes for the simulated circuit to execute a single instruction. We might have to wait for weeks to see if it correctly boots or not. Normally we use the low level simulators for small subcircuits and then higher level simuladors for the whole system.

When we said that the goal was to understand Pegasus 42 at the lowest level we exagerated a bit. We will consider the switch level as being the lowest one in this workshop. Even though *Digital* is not optimized for this level, it is sufficient to illustrate the ideas that will be presented. It is also not optimized for the higher levels, mas for the reduced projects that will be studied it is sufficient (though too slow to show the operation of circuits that output video in a usable manner).

An abstract representation of a system is a box with a number of inputs and some outputs. Normally we will show the inputs coming from the left and the outputs going right, but we can ignore this rule if it makes the drawing more confusing.

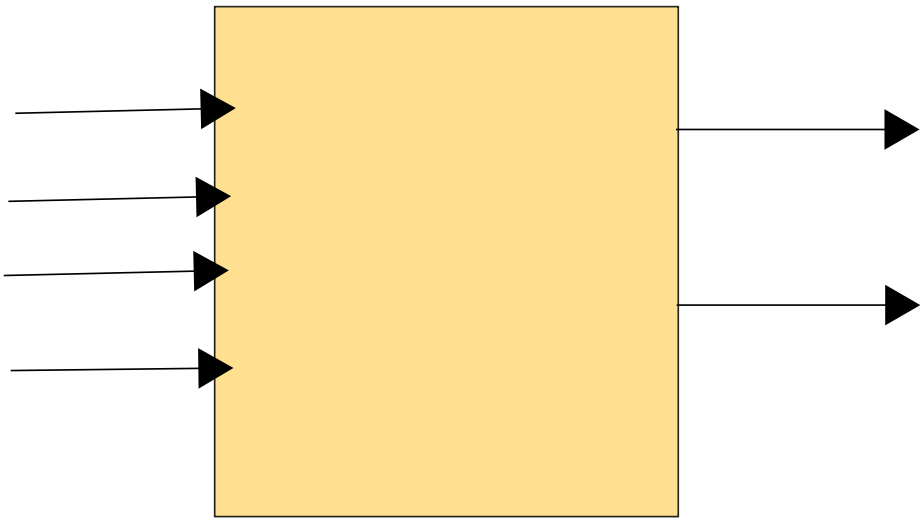


Figure 1: system

Digital ou Analog

The first choice we need to make is the nature of the inputs and outputs of our system. For the analog inputs and outputs some quantity in our circuit (voltage, current, etc) is analog to some quantity in the world (temperature, brightness, etc). For the digital inputs and outputs several quantities in the circuit represent a single value in the world. This set of quantities can use separate inputs and outputs (a parallel representation) or a single input ou output over time (a serial representation).

characteristic	Analog	Digital
number of circuits	one	one per digit
precision	depends on circuit quality	always equal to the number of digits
noise	accumulates at each operation	does not pass from input to output

Analog circuits dominated computing until the middle of the 20th century, and telecommunications until the end of the 20th century. The most important factor was the number of circuits since the components were very expensive and their connection was a manual process. With the evolution of integrated circuits the cost became extremely low and the other factors lead to the digitalizations of tecnology. Our project is digital.

Humanity has used several different digital systems to represent numbers, with the most popular the positional decimal system wth hindu-arabic digits. The

more values each digit can have, the more sensitive to noise it becomes. But the fewer values each digit can have, the more digits are necessary to represent the same number. The best possible protection against noise is when each digit can have only 2 values, like in the positional binary system.

Though the binary system needs more digits (and so, more circuits) than the alternatives, each circuit is simpler so that is the option we shall use.

Logic Gates

Combinational circuits are those whose output (or outputs) depends only on the combination of the inputs. In the case of binary, each digit can only be either 0 or 1. There are several areas of mathematics which are equivalent when only two values are used.

Área					
Boolean Algebra	1	0	inversion	addition	product
Predicate Logic	true	false	not	or	and
Set Theory	universal set	empty set	complement	union	intersection
Switch Circuits	5V	0V	normally closed	parallel	series

Notations from all of these areas can be used to represent combinational circuits. Yet another possible representation is simply a table with a line for each combination of input values and the corresponding output. We call this a "truth table" even with the values shown are 0 and 1 instead of false and true.

We will not be completely consistent and might talk about a circuit being the in form of "a sum of products" (Boolean Algebra) and another circuit of using "not and" (Predicate Logic). The latter case is why the basic circuits are known as "logic gates".

To illustrate these ideas we will use Digital, the simulator we had previously mentioned. *Digital* was written in Java, so it is necessary to first insteall this language on your computer. The advantage of this is that *Digital* runs on computers with different operating systems and different processors. The indicated site is where the source code is, but that is only needed by those wanting to modify the simulator. That page includes a "Download" button which will fetch *Digital.zip* with the most recent version of the tool.

Switches

Since we talked about circuits with switches, lets start there connecting two switches in parallel between a lamp and a power supply. In the "File" menu we select "New". Using the "Components" menu with "Switches" and "Switch" we

can position two simple switches anywhere we want. Then “Components”, “IO”, “LED” will give us a reasonable approximation of the lamp we wanted (*Digital* has fancier options, but we won't use them here). Finally in “Components”, “Wires”, “Supply voltage” gets us the power supply we need for our circuito. Note that all circuits need both a power supply and a ground wire, but we normally don't show those and the simulator works just the same. But if we want to actually build the circuit we need to remember them.

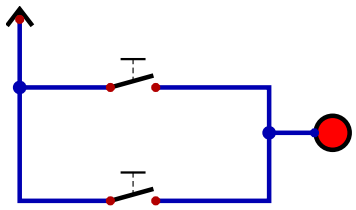


Figure 2: parallel switches

If we imagine two gardens connected using two gates, with one beside the other (in parallel), if one *or* the other is open we can go from one garden to the other. If we simulate this circuit (menu “Simulation”, “Start simulation” ou the button with the simple triangle pointing right) we will see that the LED is off. But if we process the switch on top *or* the switch on the bottom (or both) it turns on.

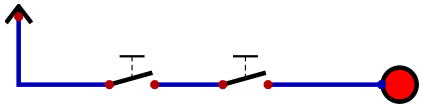


Figure 3: serial switches

If we imagine two gardens connected using two gates, with one after the other with a narrow path between them (in series), it is not enough for one of them to be open. It will only be possible to pass from one garden to the other if the first gate *and* the second gate are open. In this second circuit we connected the switches in series and in the simulation we can see that the LED stays off unless the first *and* the second switch have been pressed.

We have seen two of the three equivalencies between switches and areas of mathematics. The table indicates the remaining equivalency (inversion, not, complement) as being a normally close swtich. This kind of switch opens the circuit when pressed. But here we will show an alternative that depends on the device level (the only time we will go down to this level in this project).

We only need one switch here, and in place of the supply we use “Components”, “Wires”, “Pull-Up Resistor”. We also need “Components”, “Wires”, “Ground”. When the switch is open, a current goes through the resistor and into the LED, which shines. As the switch is pressed it offers a path to 0V that draws the current from the resistor instead of the LED, which turns off.

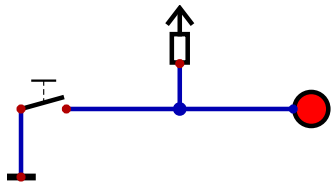


Figure 4: inversion with a switch

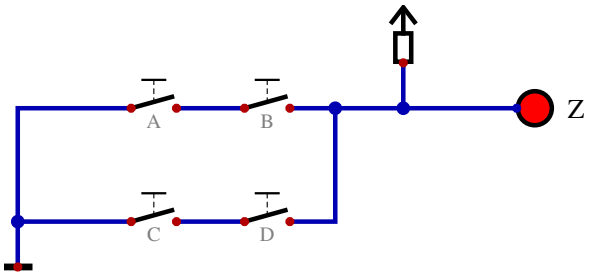


Figure 5: and, or, invert with switches

Here we have a more complex example using the same idea. Normally the AOI gate(and/or/invert) is not considered to be a basic lógic gate and we will not see it again in this project, but it is sufficiently useful that it is often included in a library for integrated circuit design. In Boolean algebra we have:

```
Z = !(AxB + CxD)
```

while in predicate logic it would be:

```
Z = not((A and B) or (C and D))
```

A fourth representation of the AOI circuit (with the first being the figure, or schematic, the second the boolean algebra and the third the logic equation) would be the truth table:

A	B	C	D	Z
open	open	open	open	on
open	open	open	closed	on
open	open	closed	open	on
open	open	closed	closed	off
open	closed	open	open	on
open	closed	open	closed	on
open	closed	closed	open	on
open	closed	closed	closed	off
closed	open	open	open	on
closed	open	open	closed	on
closed	open	closed	open	on

A	B	C	D	Z
closed	open	closed	closed	off
closed	closed	open	open	off
closed	closed	open	closed	off
closed	closed	closed	open	off
closed	closed	closed	closed	off

The table shows a problem - the input have one nature (they are controled by a human finger) while the output is of a different nature (light coming from the LED). If we want the outputs of a circuit to be used as inputs of another circuits in order to build larger systems, they need to be of the same kind. Fortunately switches which are controled using electricity have been invented: relays (1835), vacuum tubes (1904) and transistors (1947). Even though *Digital* can simulate relays in a limited way (like its switches) we will replace the switches in the AOI circuit with MOSFET (Metal/Oxide/Silicon Field Effect Transistors) of the N type (negative) which is used to make integrated circuits.

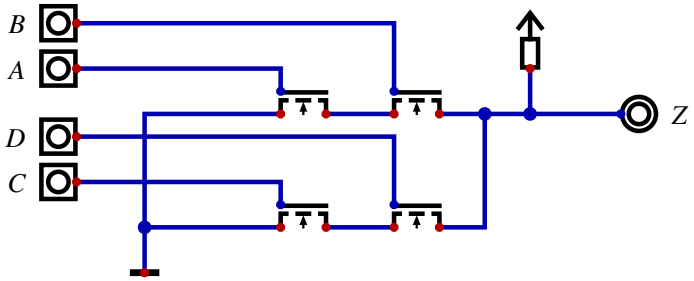


Figure 6: and, or, invert NMOS

Besides replacing the switches with “Components”,“Switches”,“N-Channel FET” we also use “Components”,“IO”,“Input” as well as “Output” from the same menu to show that these signals can come from another circuit and that the result can go to another circuit. While simulating we can change the values of the inputs and observe the value of the output.

In the “Analysis” menu, the “Analysis” item will create a truth table for the circuit and we can verify that it is the same one as for the circuit with switches. A problem with this type of circuit, which we call NMOS, is that whenever the output is 0 there is a current going through the resistor and generating heat (and draining the battery if that is where the power is coming from). Another kind of transistor, the “P-Channel FET”, is the opposite of the N type and conducts current when the input is 0. If we replace the resistor with a complementary circuit of the N transistors using the P transistors (placing them in series where the other is parallel) the circuit will work just the same but without a constant current.

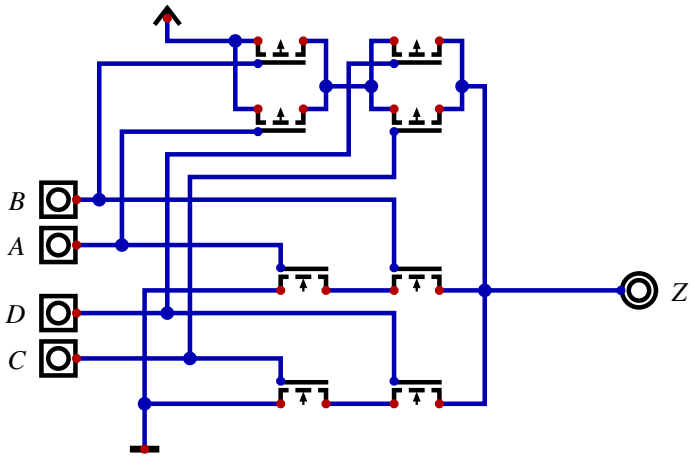


Figure 7: and, or, invert CMOS

The extra complexity of CMOS relegated it to niche applications (like digital watches) in the 1960s and 1970s, but it replaced nearly all other kinds of circuits in the 1980s when the increasing number of transistors per chip (Moore's Law) made having twice as many transistors worth it to reduce power.

one input Logic Gates

So far we have seen a single logic gate with one input: Not.

With a single input, only two combination of inputs are possible: either 0 or 1. Its truth table will be two lines in size. The output in each line can have two values, so there are 2^2 = 4 possible truth tables.

A	Z
0	0
1	0

In the first gate the output is always 0. It isn't surprising that we didn't talk about it. In circuit terms we only need to connect the output to the ground wire.

A	Z
0	1
1	0

This is the Not gate that we have already seen.

A	Z
0	0
1	1

Here the output is the same as the input. Just like in the first circuit we can implement this with just a wire.

A	Z
0	1
1	1

The final gate is an output that is always one. This can also be done with a wire connected to the power supply.

So out of the 4 possible gates, only Not is interesting. Note that if we consider the value of Z in each table as the bits of a binary number with the first line being the least significant digit, we can call these gates "gate 0" (Z = 0), "gate 1" (Z = !A), "gate 2" (Z = A) and "gate 3" (Z = 1).

two input Logic Gates

Using the same thinking, a two input gate has 4 possible input combinations and therefore a truth table with 4 lines. Using the same scheme to number them we will have a 4 bit binary number indicating that there are 2^4 = 15 possible logic gates.

outputs	equation	name
0 0 0 0	Z = 0	
0 0 0 1	Z = !(A+B)	NOR
0 0 1 0	Z = Ax!B	
0 0 1 1	Z = !B	
0 1 0 0	Z = !AxB	
0 1 0 1	Z = !A	
0 1 1 0	Z = (!AxB)+(Ax!B)	XOR
0 1 1 1	Z = !(AxB)	NAND
1 0 0 0	Z = AxB	AND
1 0 0 1	Z = (AxB)+(!Ax!B)	XNOR
1 0 1 0	Z = A	
1 0 1 1	Z = A+!B	
1 1 0 0	Z = B	
1 1 0 1	Z = !A+B	
1 1 1 0	Z = A+B	OR

outputs	equation	name
1 1 1 1	Z = 1	

Gates 0000 and 1111 don't actually have any inputs, while 0011, 0101, 1010 and 1100 ignore one of the inputs. They are actually they gates we already saw above.

6 gates have names in the menu "Componentes", "Logic" as well as a corresponding drawing. There is a special shape for AND (1000), OR (1110) and XOR (exclusive OR - 0110) and for their inverses we just add a little circle to the output and an "N" to the begining of the name. *Digital* also allows a little circle to be added to any of the inputs which means we can use an AND for gates 0010 and 0100 and an OR for 1011 and 1101.

Sum of Products If we look at the line for XOR and XNOR we will notice that they are the most complicated ones. In the case of XOR the first product (AND) is !AxB which corresponds directly to the left 1, while Ax!B is what generated the right 1.

outputs	product
0 0 0 1	!Ax!B
0 0 1 0	Ax!B
0 1 0 0	!AxB
1 0 0 0	AxB

So the XOR is the sum (OR) of the second and third prodcuts of this table. This is actually true for any logic gate and can be expanded for any number of inputs. This means that nobody will ever discover as new logic gate in the future that we don't know how to implement. The truth table can be transformed directly into a circuit.

That doesn't mean that the circuit created this way will be very good. Using this method for the next to the last logic gate we would have

```
Z = (Ax!B)+(!AxB)+(AxB)
```

But we know that a simple OR gate will do the same thing. Fortunately Boolean algebra has rules for simplification much like the rules in normal algebra, and there is a graphical method (called Karnough Map, which is one of the things *Digital* can generate) to reduce the logic to the minimum while keeping the same operation.

Multiple Bits

One disadvantage of digital circuits relative to analog ones is the repetition of the same circuit for each digit. If we use 32 bits to represent values, we will have 32 copies of each circuit.

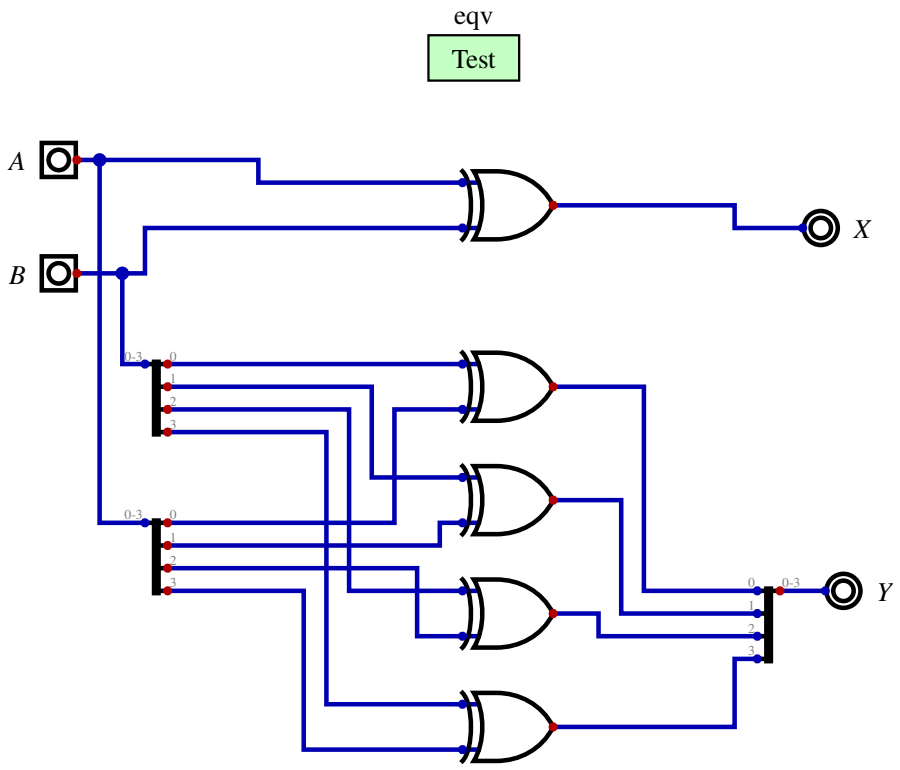


Figure 8: multipĺe bits

Digital has a feature which can reduce this complexity. For each input and output, besides the name we can define a "width" in number of bits. In the above circuit we changed A, B, X and Y to have 4 bits each. In "Components","Wires","Splitter/Merger" we have a way to connect signals with different numbers of bits. In the two on the left the input was configured as "4" and the output as "1,1,1,1" while the one on the right was configured as the opposite. In addition, each normal component can be configured to have a given width. The exclusive or gate at the top was configured to 4 bits while the four below it as 1 bit each.

Both circuits should be completely equivalent, but the one on top is easier to understand since it is much smaller. And this is for only 4 bits - the gain for, for example, 32 bits is proportionally greater. While editing the circuit, *Digital* does not give any visual indications that the top gate is different than the others, that

the inputs and outputs are for multiple bits nor even that the wires connecting them will be carrying multiple bits. During simulation, however, the wires with 1 bit are shown in dark green (for 0) or light green (for 1) while those with several bits continue dark blue with the current value shown right above the wire. And clicking on a 1 bit input will invert its value while on a multiple bit input this will open a dialog box to define the new value.

Using "Analysis","Analysis" we can see the truth table and compare the bits fomr X and Y to check that the circuits are actually equivalent. With 256 lines, however, this confirmation is quite tiring. And if we make any changes to the circuit we will have to repeat this careful examination of the truth table. Fortunately, *Digital* allows us to automate this using "Components","Misc","Test Case". We edit the test (which we name "eqv") to:

```
A B X Y

loop(a,16)
  loop(b,16)
        (a)  (b)  (a^b)  (a^b)
  end loop
end loop
```

Using "Simulation","Run tests", all 256 combinations of a and b are generated for A and B and X and Y are compared with a *XOR* b. All tests that pass are shown in green and all tests that fail are shown with a red "x". It is possible to examine the details to know where a failure happened.

Another way that *Digital* hides complexity is the use of hierarchical projects. Several components can be combined into a circuit that is then shown as a single block in a higher level circuit. Any practical project should make extensive use of this as well as include many tests for each sub-circuit. But since the goal of this workshop is to show the complexity of a computer, the projects that will be shown will be as "flat" as possible. Sub-ciruits are often compared with subroutine calls in programming languages but they are actually more like macros.

Decisions

In programming languages we have constructions like "if A then B else C" to use one input to select between two other inputs.

Looking at the truth table we can see that Z = B whenever A é 1, but Z = C when A is 0. This means that combinational circuits can make decisions. It is possible to select between more than two alternatives, like in the "switch/case" statements that programming languages have.

This is more complicated to test because with 8+3 = 11 inputs there are 2048 possible combinations. The complete test (t8x256) is possible:

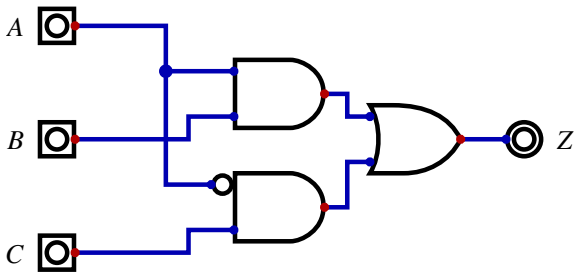


Figure 9: 2 input multiplexer

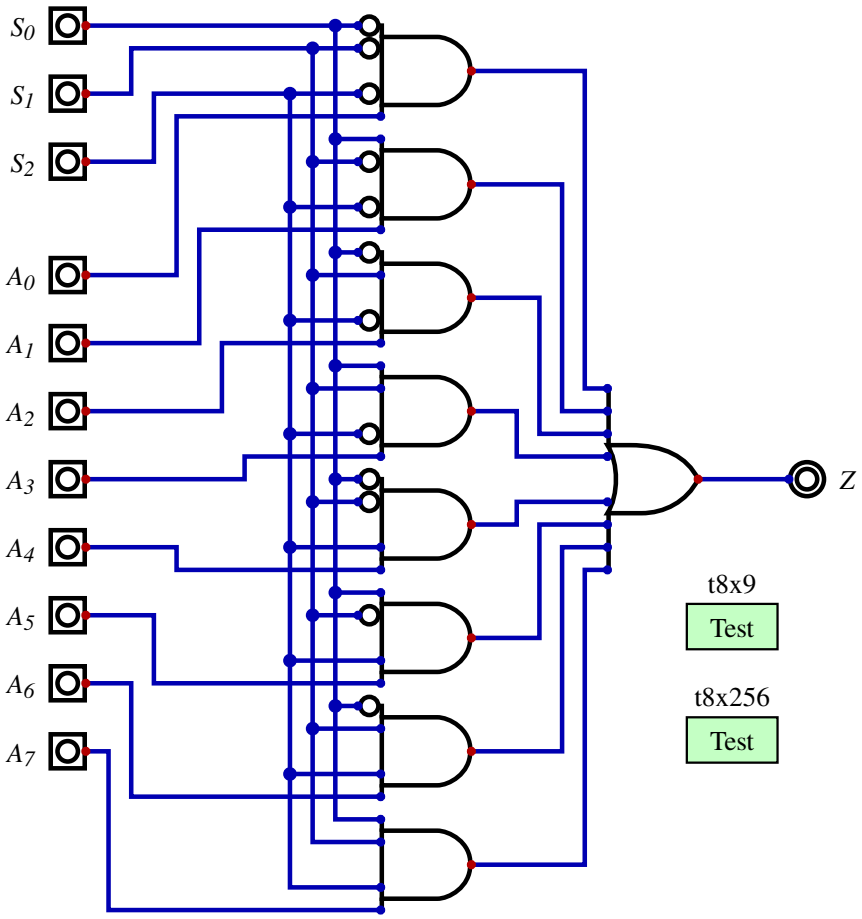


Figure 10: 8 input multiplexer

```
S_2 S_1 S_0   A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0  Z

loop(s,8)
  loop(a,256)
        bits(3,s) bits(8,a) bits(1,a>>s)
  end loop
end loop
```

Here we are confirming that the output corresponds to the bit selected by S (Z is calculated by right shifting so that the least significant bit is the one from the desired input). A more careful test would be to check that when all inputs are 0 the output is also 0 and then to turn on each input, one at a time. The output should remain at 0 except when it is the selected input that goes to 1. Instead of 8x256 tests we need only 8x9. In this case the more complete test is actually better, but during the fabrication of some product where part of the cost is the time spent using test equipment the reduced solution would be more interesting.

Our project will use many multiplexers and the circuit above is rather big (and would have to be repeated 32 times to select between 8 values of 32 bits each). At the logic gate level this is the best solution, but if we go down to the switch level we can save transistors.

This circuit passes the same tests as the previous one. In practice this use of NMOS pass transistors reduces the signal level, but adding two inverters to the output will eliminate this problem. Using pairs of NMOS and PMOS pass transistors is another solution, but the wiring becomes a bit more complex. The idea of showing this was so we can use multiplexers in our projects without worrying too much about their cost.

Numbers

A very popular idea is that computers only manipulate numbers but we can interpret these numbers as being letters, colors, sounds, etc. This is subtlely wrong - computers can manipulate representations of many things, including numbers. This is not easy to notice for positive integer numbers, but for negative numbers or floating point numbers this becomes clearer.

The first detail we need to notice is the difference between "+" in Boolean algebra and in binary arithmetic:

A	B	Boolean A+B	Binary A+B
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	10

The result in the last line has more digits than the inputs in the case of binary

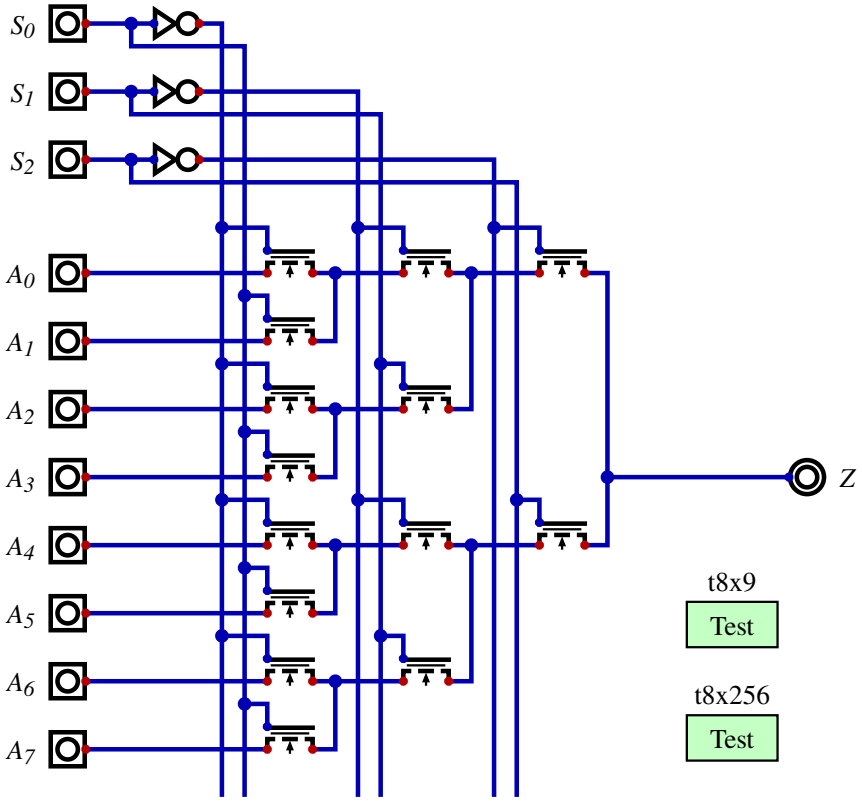


Figure 11: 8 input multiplexer

arithmetic. That is also the case for decimal arithmetic: adding two decimal numbers with 5 digits each might generate a result with 6 digits. And for each digit the result will be from 0 to 18, with the last being 2 digits.

Note that the last line in the table is not saying that "one plus one equals ten". The same way that the decimal number 307 means 3x100 + 0x10 + 7x1, the binary number 1010 means 1x8 + 0x4 + 1x2 + 0x1 which is the number ten. In the decimal positional system the rightmost digit is the unity and each digit to the left is worth ten times as much. In the binary positional system the rightmost digit is the unity and each digit to the left is worth two times as much. The binary number 10 is 1x2 + 0x1 which is two.

We shall call the digits from the addition of binary 1 bit numbers S (from "sum") for the least significant and C (for "carry") for the most significant.

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

C has the truth table of the AND logic gate and S of the Exclusive OR gate. So an adder is:

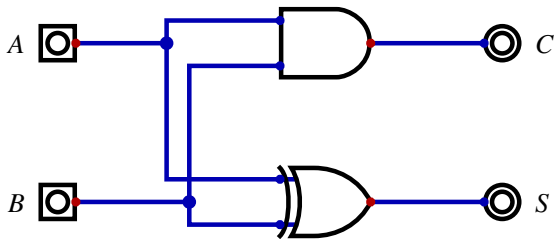


Figure 12: half adder

For the least significant digits of the input numbers this does the job. But for any other digits we need to take into account the "carry" from the digit immediately to the right. That is why we call this circuit a "half adder" and use two of them to create a "full adder", which is a circuit with 3 inputs and 2 outputs.

Repeating the full adder 4 times we can add two binary numbers with 4 bits each. Actually, we could have used a half adder for the least significant digit but doing it this way it becomes easy to combine several of these blocks to handle larger numbers.

We can test that we are actually adding numbers:

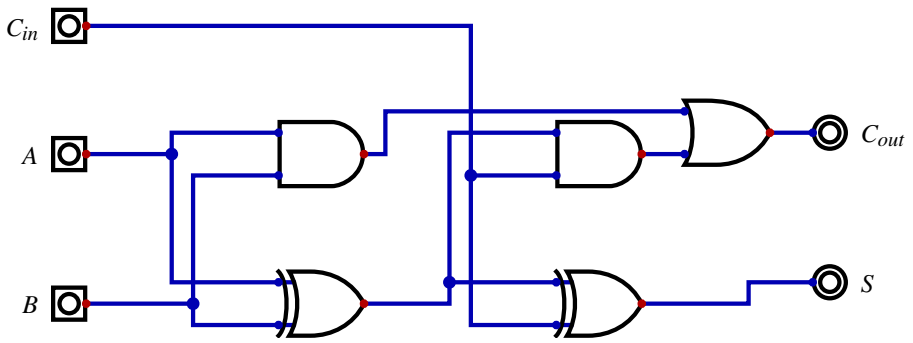


Figure 13: full adder

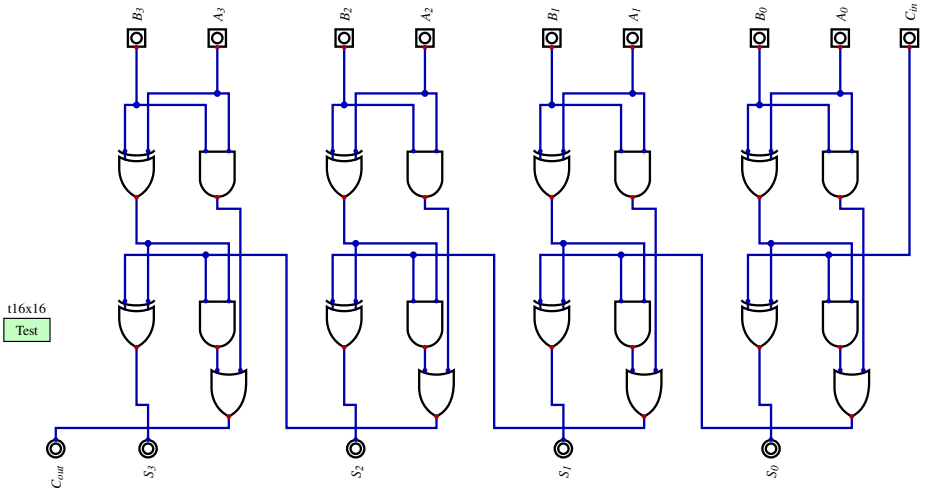


Figure 14: 4 bit adder

```
C_in   A_3 A_2 A_1 A_0   B_3 B_2 B_1 B_0   C_out S_3 S_2 S_1 S_0

loop(a,16)
  loop(b,16)
      0 bits(4,a) bits(4,b) bits(5,a+b)
      1 bits(4,a) bits(4,b) bits(5,a+b+1)
  end loop
end loop
```

Note that with *C_out* and *S_3* to *S_0* the result is 5 bits long, one more digit than the inputs.

This "ripple carry adder" is not very fast. The carry goes from one digit to the next and only arrives at the most significant digit with a long delay. If we double the number of digits the adder will operate at half the speed. There are more complex circuits that reduce this problem, but for out project this simple solution is good enough.

For subtraction we face a new problem: negative numbers. We menctioned previously that computers manipulate representations and not numbers, but it is easy to ignore the difference in the case of positive numbers. For negative numbers we must choose between several representations, each with its advantages and complications.

The most popular representation for decimal numbers is the sign/magnitude one. A special digit to the left indicate if the number is positive ("+" or nothing) or negative ("-"). The remaining digits indicate the number's absolute value. The same idea can be applied to binary numbers, and we can even use 0 and 1 to indicate positive and negative numbers to make the system even more uniform. Two problems with this system is the existence of two different zeros (+0 and -1) and the fact that addition and subtraction are slightly different operations and it is necessary to examine the signs of the operands and compare their magnitudes to select the right operation.

A representation that was only every used in a few of the very first mechanical calculators was nine's complement. A negative number is represented by replacing each digit by nine minus that digit. The negative of 307, for example, would be 692 (and in the case of a calculator with six digits 000307 negated would be 999692 which makes it easy to interpret the leftmost digit as indicating the sign). Now subtractions is just inverting the second operand and adding (ignoring any carry generated during that operation): 000531 - 000307 = 000531 + 999692 = (1)000223. Except this is not the right answer since 307+223 = 530. This is one of the problems with nine's complement: we need to add one to correct the result. And it also has two zeros. In the binary case the equivalent is one's complement (which is the NOT function).

Ten's complement is a slight modification of the previous scheme that corrects both problems. To negate a number we subtract each digit from nine and then we add one. The ten's complement of 000307 is 999693 and now additions will

directly give the right answer. Besides that 000000 continues to represent zero but 999999 now means negative one - there no longer is a negative zero. One detail is that there are more negative than positive numbers for a given digit length, but that is only an aesthetic issue. The equivalent binary system is the two's complement.

The last system we will look at is the bias. Here we add a value to all numbers so they all are positive. Selecting the bias as half of the largest possible number plus one, the addition of two number will convert the two biases into a carry, and then we need to add a third bias to adjust the answer. 531 is represented by 500531 and -307 by 499693. 500531 + 499693 + 500000 = (1)500224.

Comparing these representations in the case of 3 digit binary numbers (with the values shown in sign/magnitude decimal notation):

representação	000	001	010	011	100	101	110	111
positive	0	1	2	3	4	5	6	7
sign/magnitude	+0	+1	+2	+3	-0	-1	-2	-3
one's complement	+0	+1	+2	+3	-3	-2	-1	-0
two's complement	+0	+1	+2	+3	-4	-3	-2	-1
bias	-4	-3	-2	-1	+0	+1	+2	+3

All these representations were used in the history of computing, but in the 1960s two's complement became dominant and it is what we will use in our project. In the IEEE 754 standard for floating point numbers, however, the mantissa uses the sign/magnitude representation while the exponent uses the bias representation.

With a small change to the 4 bit adder it can also subtract using the two's complement representation. We said that NOT can generate the one's complement, but using a XOR for each digit we can control whether we do this inversion or not. In the *-16x16* subtraction test we see that we are actually calculating *a-b-1* which is a problem we saw with one's complement. But using *Cin* we can correct that, getting the two's complement. In the subtract mode the circuit generates an inverted *Cout* and the test takes this into account.

```
invB  C_in   A_3 A_2 A_1 A_0   B_3 B_2 B_1 B_0   C_out S_3 S_2 S_1 S_0

loop(a,16)
  loop(b,16)
      1 0 bits(4,a) bits(4,b) bits(5,(a-b-1)^16)
      1 1 bits(4,a) bits(4,b) bits(5,(a-b)^16)
  end loop
end loop
```

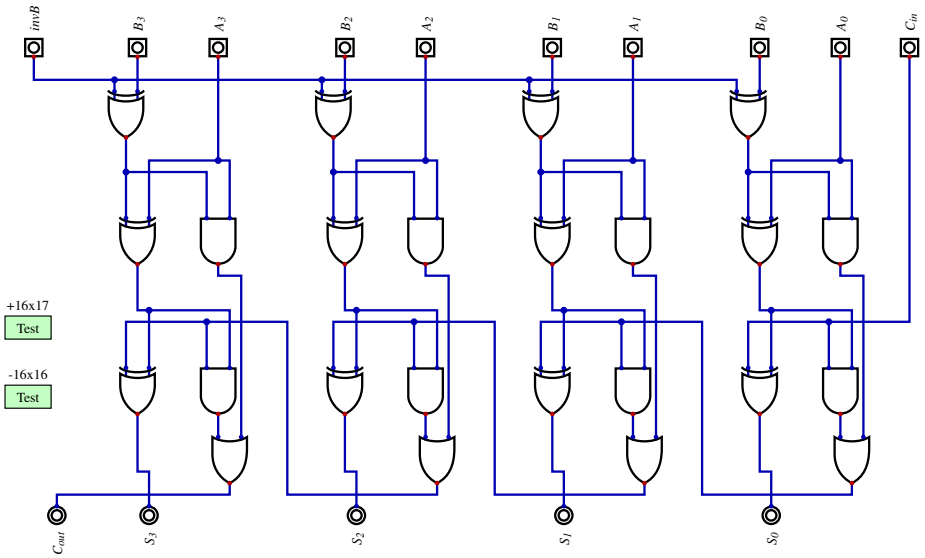


Figure 15: 4 bit adder and subtractor

2. Sequential Circuits

Time is not a factor for a combinational circuito. Of course, given that it is not infinitely fast the answer will only be ready after a certain delay after the inputs receive their values.

In contrast, time is fundamental for sequential circuits. The inputs arrive as a sequence over time using the same signals and the results are also sent as a sequence of values over time. We can convert the abstract combinational circuit we previous saw into a sequential system by connecting some of its outputs to inputs. We call the value being fed back the "current state" of the system.

In practice a circuit like this can even work, but it would be rather unstable since some paths generating some of the bits of the current state might have longer delays than those for other bits. The most popular solution is to add a "clock" signal so the feedback can happen in a more controlled fashion.

Oscillators

Digital refuses to simulate this circuito, which is the simplest sequential system. One problem is that the circuit is contradictory: the output would have to be 1 for a 0 input, or 0 for a 1 input. But there is a wire connecting them so they should have the same value. If we actually build this circuit it will keep quickly alternating between 0 and 1. This is what we call an "oscillator". The oscillation frequency depends on the speed of the inverter and the delay of the wire. The main problem from a simulation viewpoint is that the initial value is unknown,

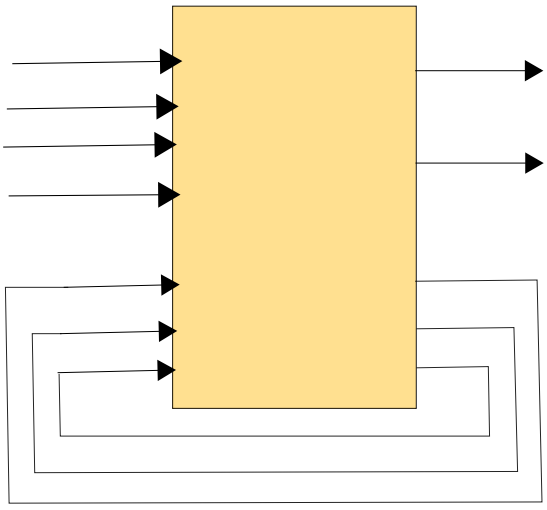


Figure 16: sequential system

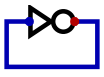


Figure 17: one inverter oscillator

which we can solve with an initialization signal (we will call it “reset”).

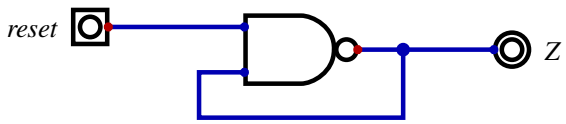


Figure 18: one nand oscillator

If we try to simulate this circuit it seems to work, but as soon as we change *reset* to 1 an error occurs. The solution is to simulate step by step. After changing *reset* to 1, at each step in the simulation the output alternates between 0 and 1.

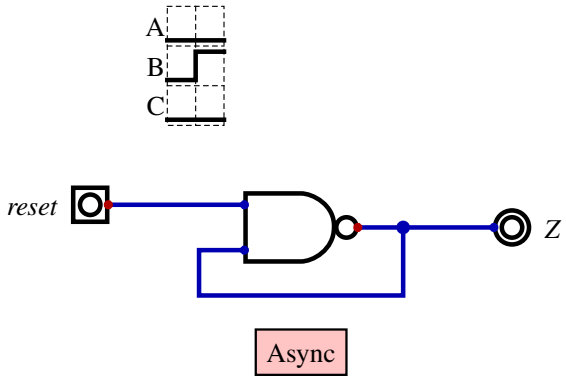


Figure 19: one nand oscillator

Adding “Components”,“Misc”,“Async” changes how the simulation’s visualization happens, and with “Componentes”,“IO”, “Graph” we can visualize the input and outputs over time, which is to sequential circuits what truth tables is to combinational circuits.

Memory

Connecting two such oscillators in series makes it stop oscillating since an even number of inversions is not contradictory. This circuito is “bi-stable”, meaning there are two different situations in which the circuit stays put. The output of the first part is always the opposite of the output of the second part, so we use a “!” in front of its name. We also do that for the inputs to indicate that these should normally stay at 1 and should only go to 0 when we want it to do its job.

The name of this circuito is “flip-flop” to indicate its bi-stable nature, and it is a computer’s simplest memmory capable of holding 1 bit. A more complete name is “RS flip-flop” since the inputs *!reset” and* !set* are separate. The circuit “remembers” that a negative pulse has arrived at *!set* even after that pulse is no longer there. Additional pulses don’t do anything. The same thing for pulses at *!reset*.

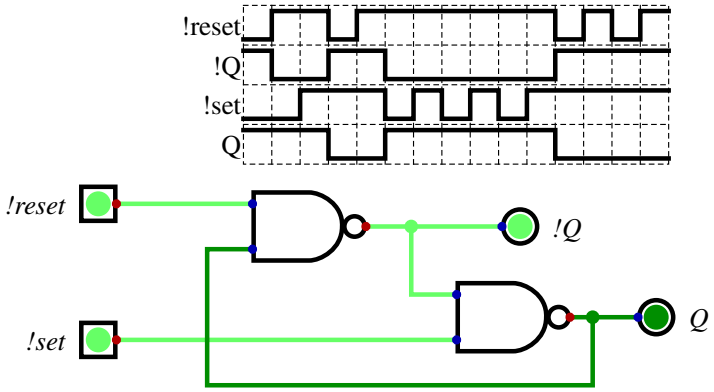


Figure 20: flip-flop with nands

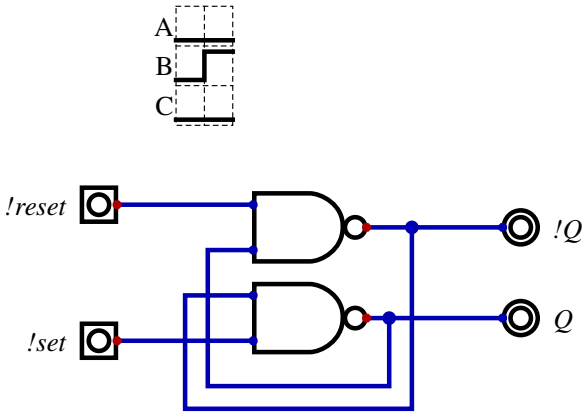


Figure 21: flip-flop with nands

This is exactly the same circuit, but drawn in the style of the abstract sequential system with the outputs going all the way around the circuito to connect to the inputs. The idea is that even with the complications of the next circuits we can still keep in mind that this feedback is happening.

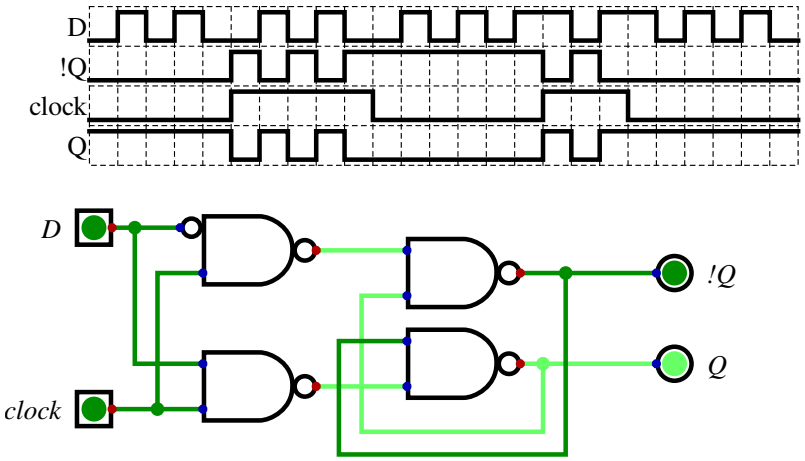


Figure 22: D flip-flop

Instead of directly generating *!reset* and *!set* it is more convenient to have a data signal *D* and a *clock* signal. When the clock is high, the output reflects *D* with a small delay. When the clock is low the output value doesn't change no matter what is happening in *D*.

We can also use a multiplexer to have the same functionality. Though in this case we don't have the inverted input. Modern project avoid using latches because during half of the clock cycle we can't trust the output. Using two latches in a row operating on oppostive levels of the clock we can have an output that remains stable during nearly the whole cycle, with the only uncertainty very close to one of the clock edges.

We can have this same funcionality by selecting "Components","Memory","Register". But it is important to understand what is happening inside the component. We also replaced the *clock* input from a regular one to a special input. In this simulation it didn't make any difference, but we can configure it to pulse at a give rate without having to keep manually clicking on it. And in the tests this kind of input can be set to not only 0 and 1 but also C to indicate a rising edge.

Finite State Machines

Using registers we can solve the stability problem mentioned for the abstract sequential circuit. We just have to pass the output signals to be fed back through registers and make the register outputs be the ones to actually go back into the inputs. This way it doesn't matter if different bits have different delays since all

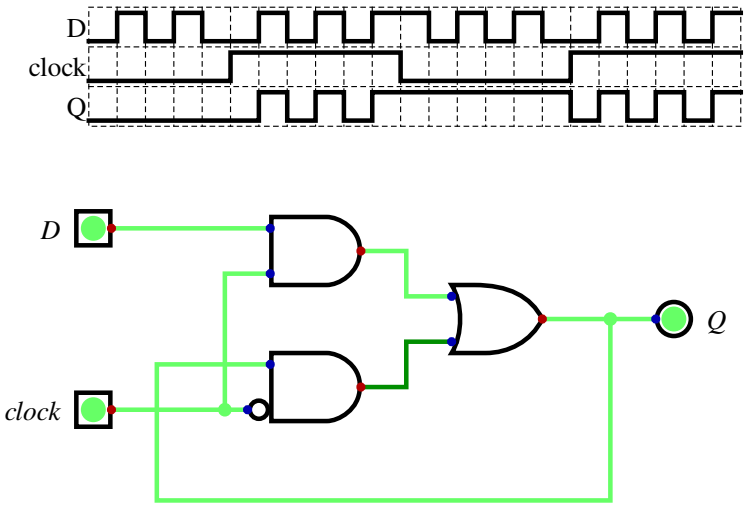


Figure 23: latch

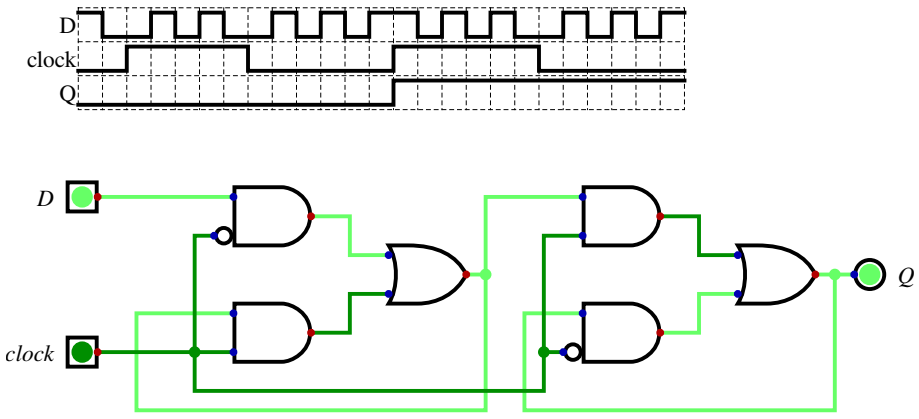


Figure 24: edge triggerd D flip-flop

of them will be sampled at the same time on the rising edge of the clock. The only remaining worry related to time is to check that these edges are sufficiently far apart that the combinational circuit has time to finish its job. This is why we say that one processor can run at 1.5 GHz while another can go up to 3.2 GHz.

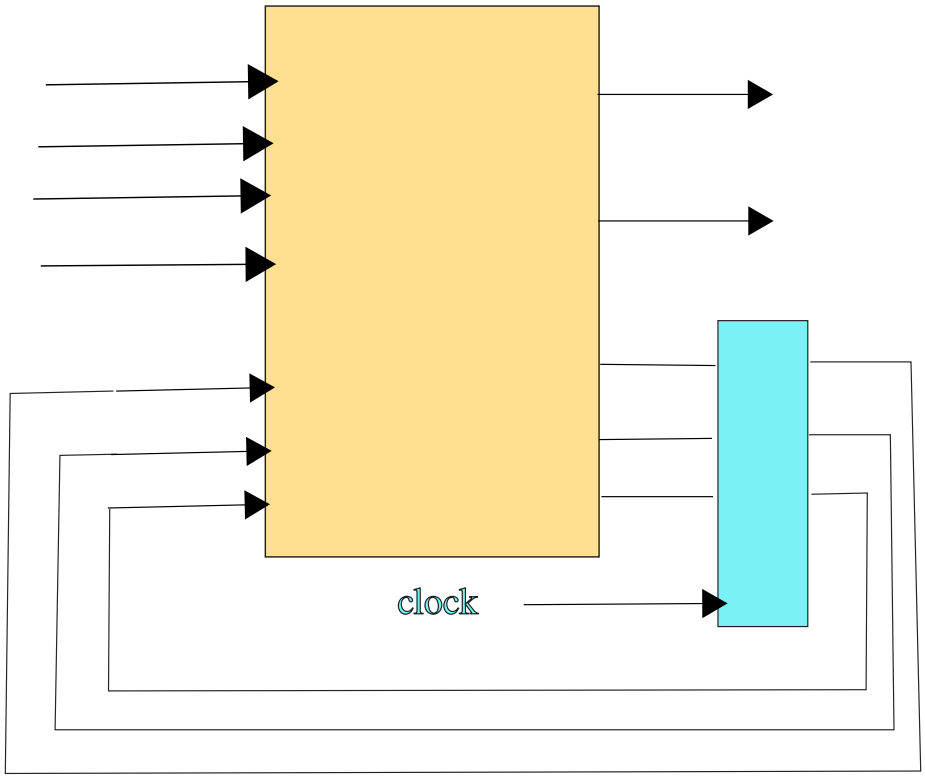


Figure 25: finite state machine

With the introduction of a lock, the state of the system can be thought of as jumping instantly from one value to the next. The number of possible states is limited by the number of bits used to represent the current state, but the number of actual states can be much smaller than that. We call this kind of system "Finite State Machine" (FSM).

One possible representation for FSMs is a table with, for example, one line for each state and one column for each possible input. Each cell would indicate the output as well as the next state. *Digital* has an editor for a very popular graphical representation: each state is shown as a circle and arrows connecting the circles show the possible transitions. The arrows indicate what values the inputs must have to jump to the other state as well as what the values of the outputs should be.

Among the examples supplied with *Digital* we have *rotDecoderMealy.fsm* with *A*

and B as inputs and L and R as ouputs. Each state has a name, but it also has a number from 0 to 6 which are the values that the registers should have for each state. This system can detect if an axis is rotating to the left or to the right.

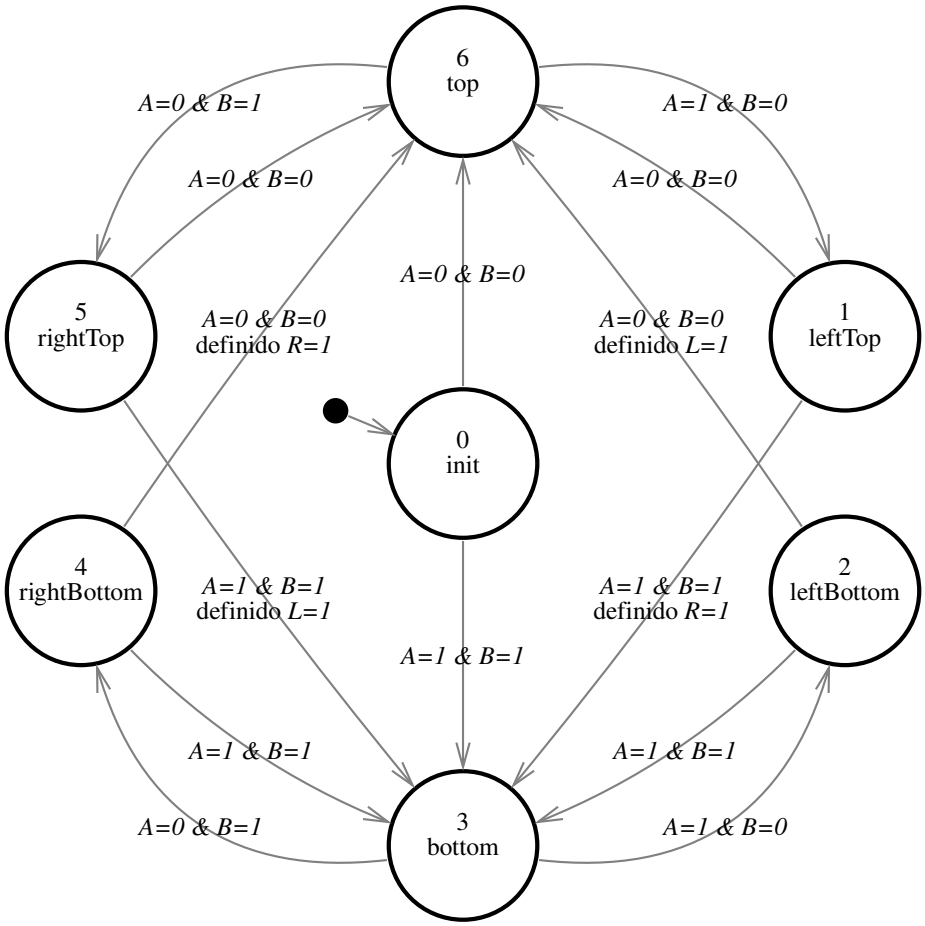


Figure 26: Mealy FSM example - rotational decoder

The "Mealy" in the file name is to indicate that this a Mealy type FSM, which was defined by George H. Mealy in 1955. One feature of this kind of FSM is that the outputs depend not only on the current state but also on the other inputs. This means that the outputs might change between clock edges if the inputs change.

In the FSM editor we have the option of generating the truth table (for the "transitions", since as a sequential system the machine as a whole can't be decribed by a truth table) and from the table we can ask for a circuit to be created. The current state is stored in registers 2n, 1n and 0n and we can see the A and B inputs we well as the combinational circuit in the form of sum of

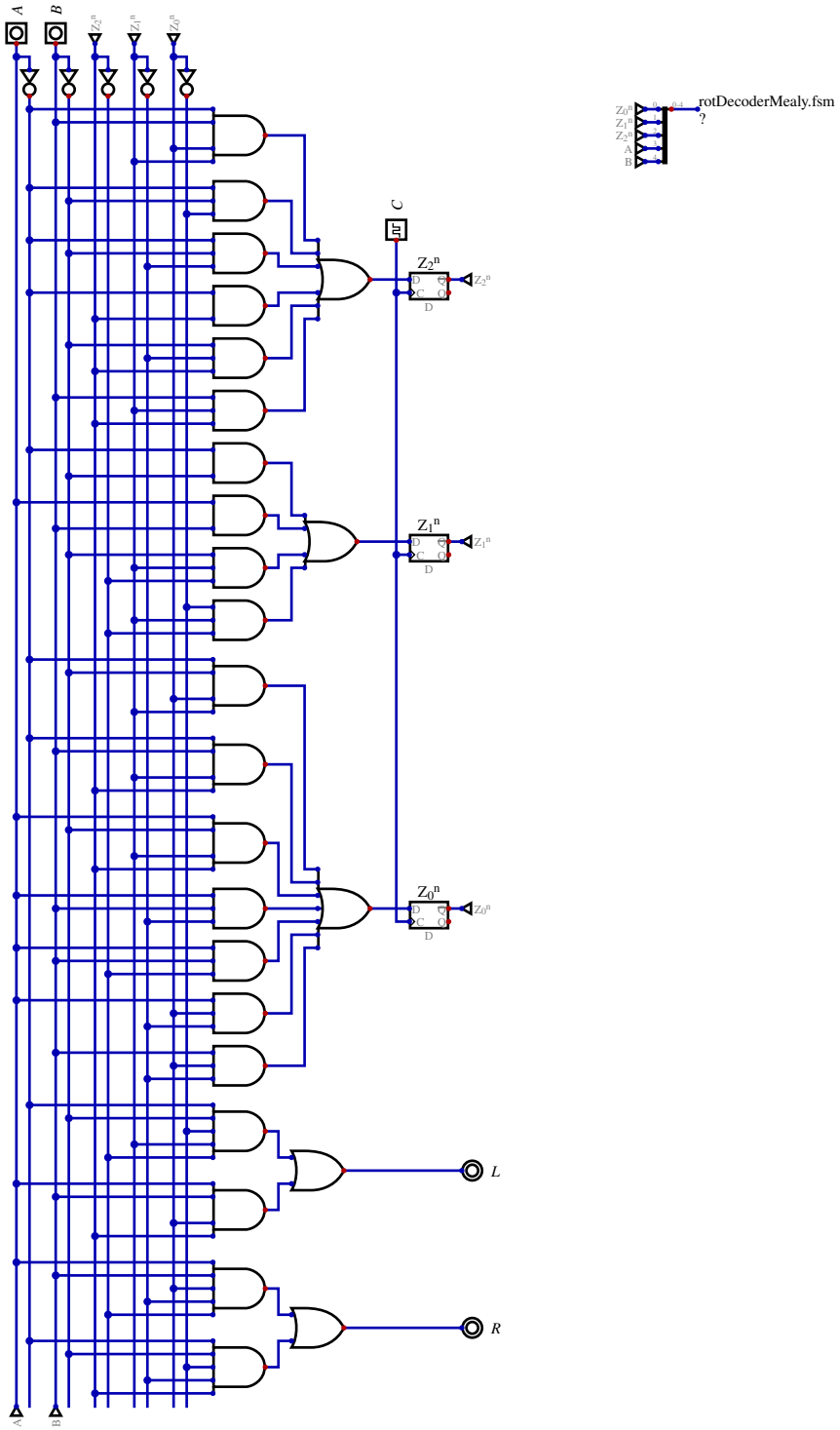


Figure 27: Mealy FSM example - rotational decoder

prodcuts that generated both the value of the next state and the outputs L and R.

The outputs of the registers should have wires going to the inputs of the combinational circuit. *Digital* has a feature called *tunnel* which allows a signal to jump from one place to another (or several others) without crossing over the rest of the drawing, and that was used here.

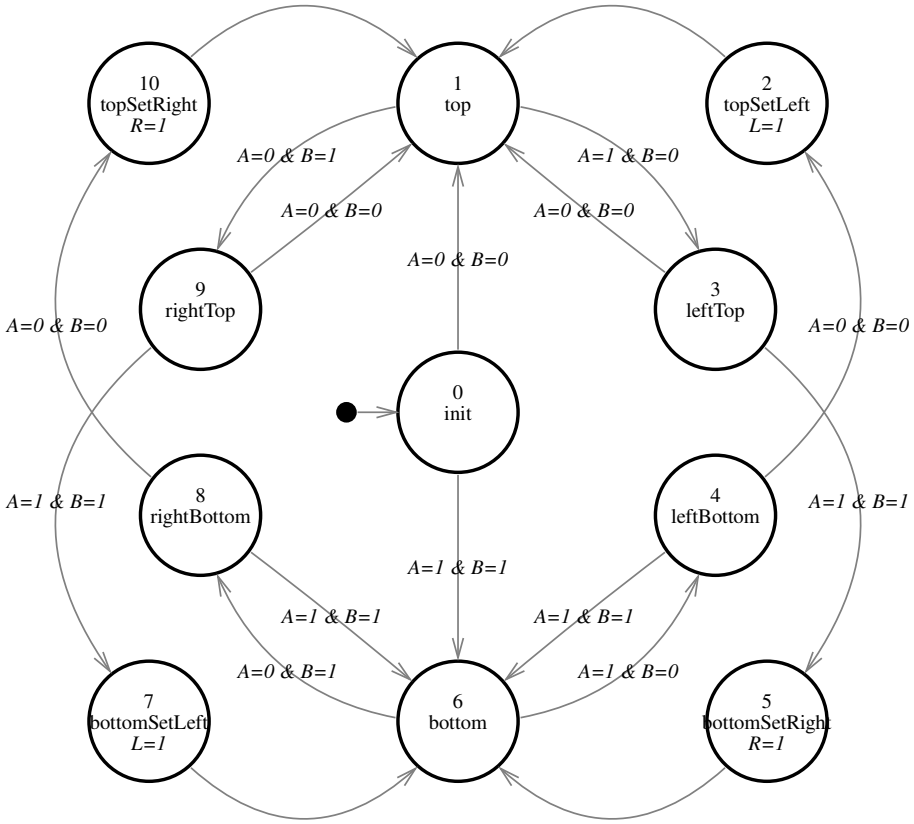


Figure 28: Moore FSM example - rotational decoder

In 1956 Edward F. Moore defined another variation of the FSM where the outputs depend exclusively on the current state. The advantage is that the outputs remain stable between clock edges. In the same example implemented as a Moore type FSM we see that extra states are needed for the same result, in this case.

The most obvious difference between this circuit and the previous one is that outputs L and R don't have any connections with inputs A and B.

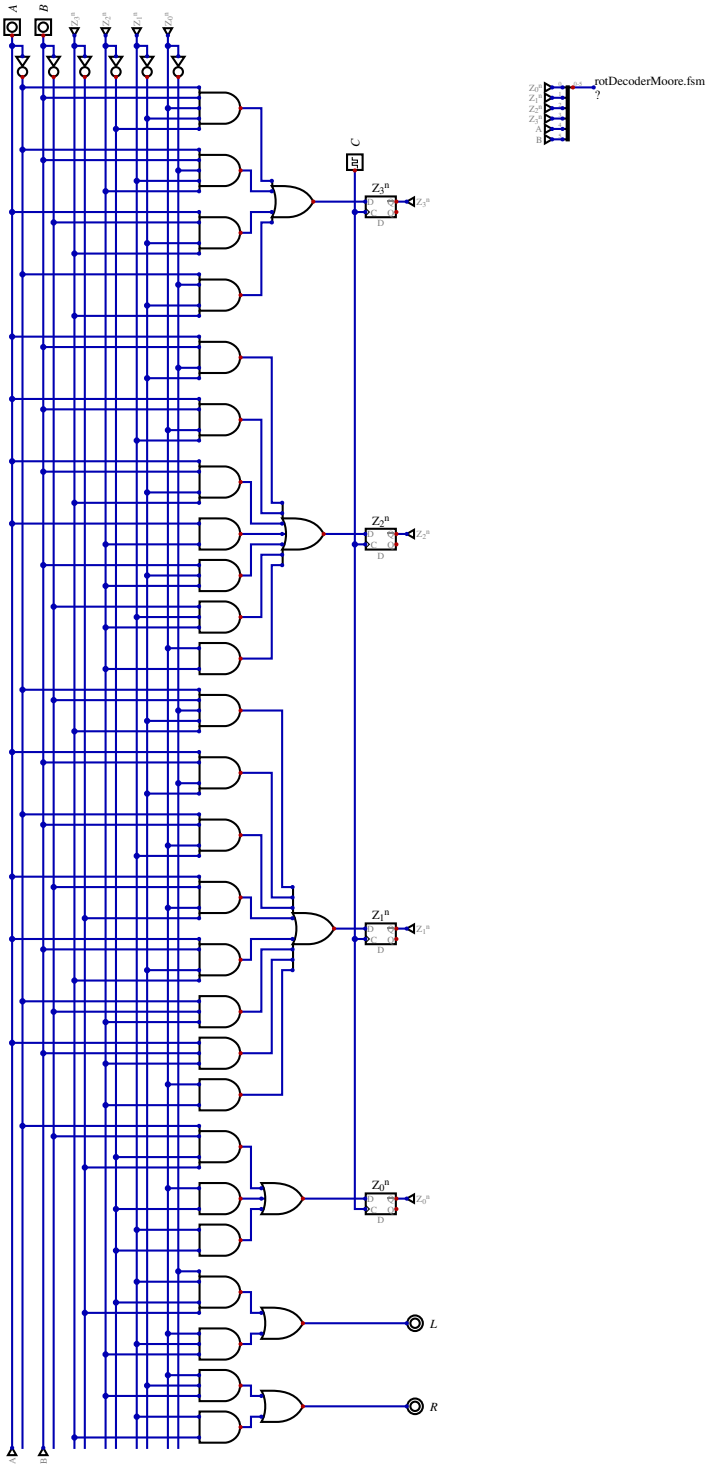


Figure 29: Moore FSM example - rotational decoder

3. Processors

All computational problems can be solved with a FSM. In theory. In practice even relatively small problems might need an absurd number of states and the corresponding machine would be impossibly expensive to build (if not absolutely impossible, for example if it needs more components than there are atoms in the universe).

Imagine a FSM that receives six characters of 7 bits each and which is supposed to print thesse characters in the reverse order. It will have 8_865_353_597_185 (8 trillion) states. The problem is that the system's only memory is the register for the current state and using that to store what characters have already been seen is not efficient. 6x7 = 42 bits to store the characters while 8 trillion states need 44 bits to represent them.

If we use a FSM connected to a small external memory it would be much more reasonable. A memory with 8 words of 8 bits each would be more than enough and a FSM with 12 state would be sufficient to control it to solve the problem.

In his 1936 paper, Alan Turing imagined something even simpler than a memory: he connected his FSM (represented in the paper as a table) to an inifinite tape with individual cells that can hold a single symbol chosen from some alphabet. There is a read/write head that is position on one of the tape's cells. The input to the FSM is the symbol in the current cell and the outputs are a symbol to be written (possibly the same one of we don't wish to change the tape at this time) and optionally a command to move the head to the cell on the left or on the right.

Today this is known as a "Turing Machine". A very interesting simulator which is available on the web is (https://turingmachine.io/) which includes several examples.

Here we have the multiplication of two binary numbers. The same thing as a pure FSM would have a much larger number of state, while here only 21 are needed. But this was created only to study a mathematical problem of the kind "there exists a uring machine capable of..." and not to be something practical. Each problem requires the construction of a new Turing Machine, but towards the end of the paper there is a very interesting proposal: a Universal Turing machine that would receive on the same tape as the input data a representation, in the form of a sequence of symbols, of a Turing Machine that would solve the selected problem. We now call this an "interpreter". Once such a machine was built, changing the tape would modify its behavior. This is what we call "software".

von Neumann and Harvard Architectures

The first computer (or "electronic brain" as the press inicially called them) was ENIAC in 1946 (previous projects were kept secret for many years). Designed

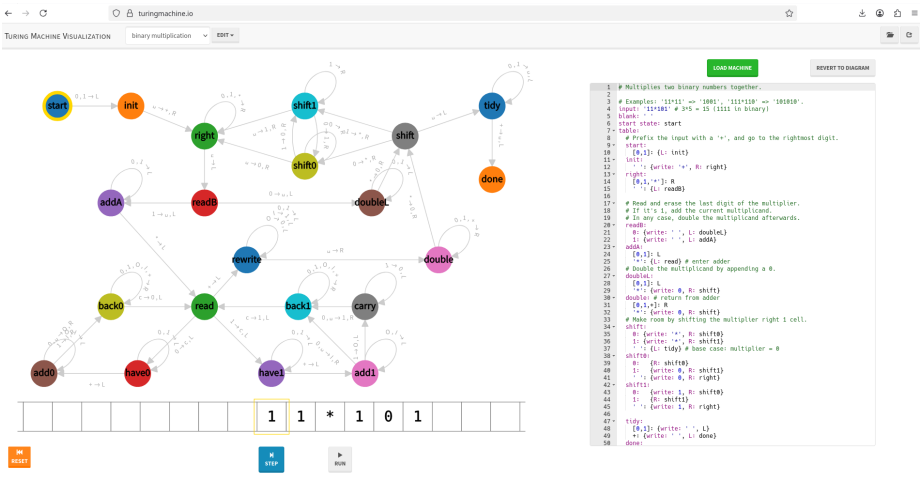


Figure 30: Turing Machine simulator

by John Mauchly e J. Presper Eckert at the University of Pennsylvania, a key limitation of the ENIAC was the need to reconfigura the hardware using patch cable panels for each new problem. So even during its development they started discussing its sucessor, to be called EDVAC.

One of the participants of these debates was John von Neumann and he wrote a detailed report on these ideas. Another participant, Herman Goldstine, ended up distributing the report to several external grupos with von Neumann as the sole author, so this style of computing is known as the "von Neumann architecture" even though it was created by a group.

John von Neumann liked analogies with biology and so called the parts of the computers "organs" and where the data is kept "memory" (others, specially IBM, prefered terms such as "storage" but ended up losing this battle).

The control unit is just a FSM, which we have already seen. The arithmetic and logic unit is a more complicated version of the adder/subtractor which we have also already seen. The input and output are different for each computer, so for now we will ignore them.

The memory stores both data and programas (just like the tape of the Universal Turing Machine). One bit of this memory is like a register, which we have already seen. What we didn't talk about is how to select one bit from many, but the multiplexer is similar to the mechanism used. An interesting feature of the von Neumann architecture is that the central processor (CPU) can be made from a technology that is completely different from that used to make memory. Some examples:

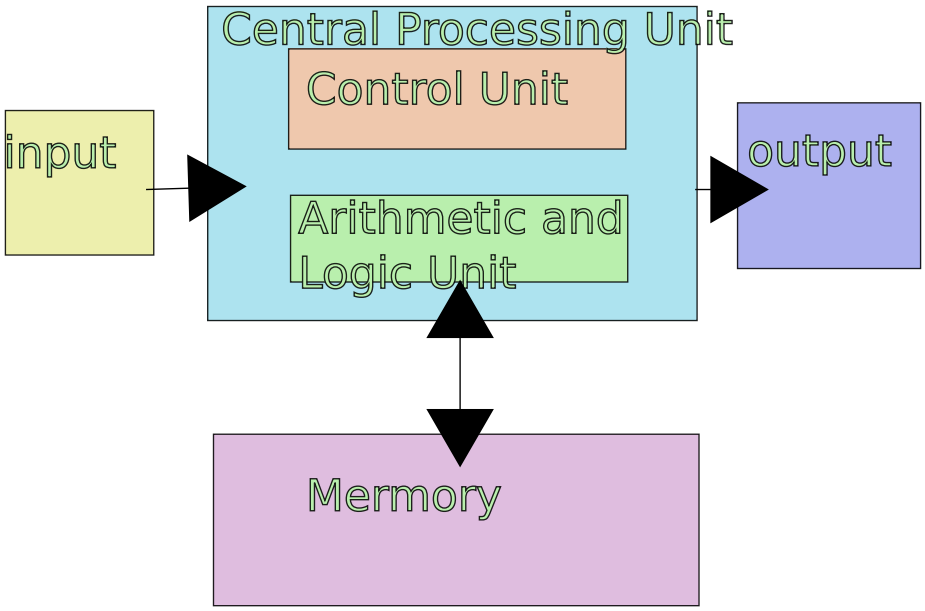


Figure 31: von Neumann architecture

Computer	CPU	Memory
EDSAC	vacuum tubes	mercury tanks
IAS	vacuum tubes	Williams tubes
LGP-30	vacuum tubes	magnetic drum
PDP-8	transistors	magnetic cores
modern PC	digital chips	chips with vertical capacitors

IAS is “Institute for Advanced Studies” in Princeton where John von Neumann and his team built the computer from his report. That is why it is more rarely referred to as the “Princeton architecture”. That is to contrast it with the “Harvard architecture”, which is named after a computer that IBM built for Howard Aiken at Harvard University. The only difference is that the data memory and program memory are separate. The separation allows an instruction and some data to be read at the same time, which can simplify the project. But makes it impossible for a program to modify itself.

Today all computers except for the simplest ones are hybrids: directly connected to the processor we have two tiny memories known as the level 1 instruction cache and the level 1 data cache. These memories only store copies of information that have been recently used by the computer. When the needed information is not there, a level 2 unified cache is acessed. And there might b e a level 3 cache and then finally the main memory as in von Neumann’s drawing. This combines

the hardware advantages of the Harvard architecture with the programming advantages of the von Neumann architecture.

We mentioned that inputs and outputs are specific to each computer. In early computers this was reflected in the instruction set. A computer might have one intruction to read a key and another instruction to write to the tape. Around 1970 the idea of making input/output devices appear as special memory locations began to become popular. This allows the processor to use "normal" instructions for everything. Of the processors current used, only x86 (Intel and AMD) still have special instructions for input and output and all the users use "memory mapped peripherals", which we will do as well.

MCPU16h

In the EDAVC report the idea was put forth of representing a program as a series numbers where the most significant bits would represent an "order" being given to the computer (this is now called the "operation code" or just "opcode) and the least significant bits would be the address of the memory position to be used for this instruction. Many operations need two operands and produce a result that must be stored somewhere. This would require 3 addresses, mas EDVAC inherited from mechanical calculators the idea of a special register called"accumulator" which supplies one of the operands and is the destination of the result for most of the instructions.

How many different instructions will we have? This defines how many bits we will need for the "opcode". It turns out that it is actually possible to do everything with just a single instruction, but that is not a good idea from an educational viewpoint. An example of such an instruction is SUBLEQ which has 3 addresses and subtracts the value at the first address from what is at the second address (saving the result back there) and if the result was negative or equal it jumps to the third address. Programas written for such a computer are nearly as hard to understand as those for a Turing Machine.

A more reasonable alternative is MCPU with its 4 instructions (two bits for the "opcode"). Tim Böscke was inspired by MPROZ and its 3 instructions, but replacing the memory to memory operations with an accumulator in the EDVAC style greatly simplified the project.

MCPU16h has two differences: while the original MCPU is 8 bits wide so only 6 bits are left for the address (pnly 64 bytes, which is sufficient for the simplest examples) MCPU16h, as its name implies, is 16 bits wide and its 14 bit addresses can handle 16 thousand words of 16 bits each (32KB). And the "h" at the end of the name is from Harvard. But like the EDVAC, MCPU16h depends on being able to modify a program while it is running but that is not possible with a Harvard architecture. But *Digital* has a two port memory and we will use that to make it look like there are two separate memories but what is written to one can be read by the other.

The choice of the Harvard architecture was to allow the use of a combinational circuit for the control unit. In a von Neumann architecture (like the original MCPU) reading an instruction happens in one cycle while reading data in a different one. Because of this the control unit has to be a sequential circuit, which is a bit more complex.

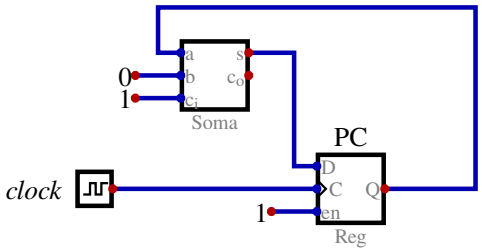


Figure 32: PC do MCPU

With this simple circuit, at each rising clock edge the PC advances to the next word. In this case we connected the "enable" of the PC register to 1 since we never stop changing the PC. But in a more complete project there will be situations where that will happen (if we have to wait for the instruction memory, for example).

So let's add the dual port memory. Port 2 is limited to reading and will be used as the instruction memory. The control signals for port 1 are connected to constant values that won't interfere with the operation (not writing anything, for example).

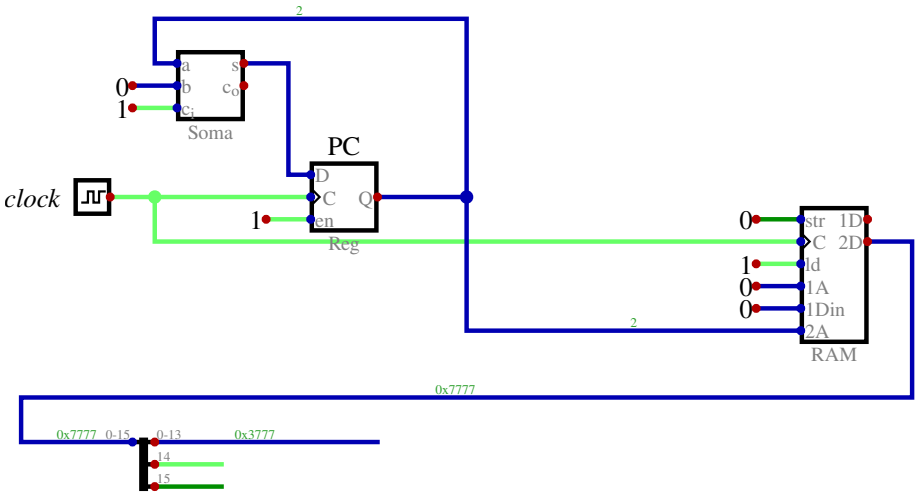


Figure 33: dual port memory

When simulation starts the memory is full of 0s. Later on we will use the contents

from a file to fill in the memory, but for not we are manually editing the values of the first few words every time the simulation starts just to see that something is actually happening. The picture shows what happens after two clock rising edges.

The data arriving from memory (0x7777 indicating that it is the hexadecimal equivalent to the binary 0111011101110111) is divided into a 14 bit address (0x3777) and two "opcode" bits (0 and 1). We are decoding the instructions using only wires.

We shall implement the first of the four instructions: JCC ("Jumps if Carry Clear"). We need to choose an "opcode" for this instruction and we selected 1 and 1.

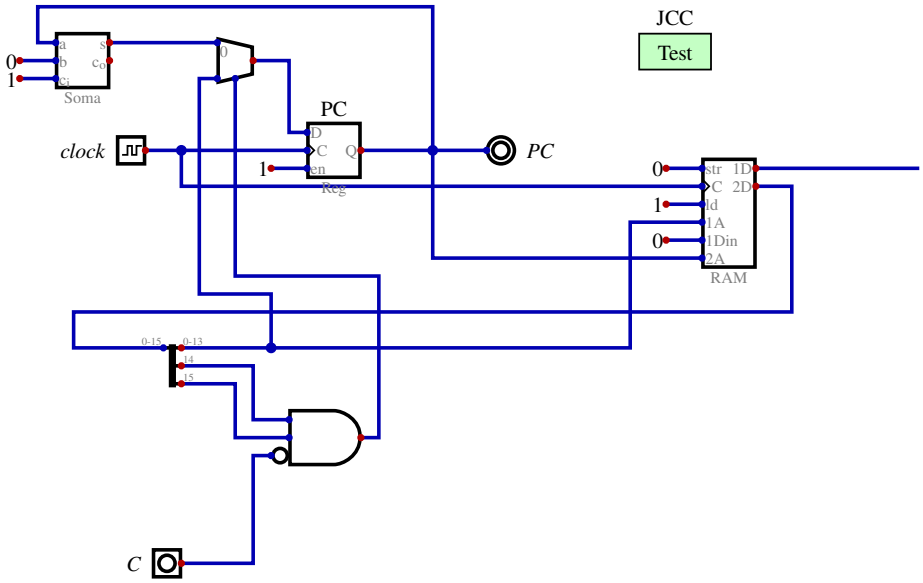


Figure 34: JCC instruction

The multiplexer decides between jumping to the address indicated by the instruction or just adding 1 to the current PC. The first case should only happen if the opcode is that of the JCC and if the C bit is 0. The 3 input AND gate detects this situation and controls the multiplexer.

Here we are creating a single circuit, but normally the AND gate would be part of the control unit while the multiplexer, the PC register and the adder would be part of what we call the "data path". And the memory would be separate from the CPU.

```
clock C PC
program(0x0000, 0xC006, 0xC001, 0xC008)
0 1 0
```

```
C 1 1
C 1 2
C 0 1
C 1 2
C 0 1
C 0 6
```

To test this circuit we need to make PC an output so it can be compared during the test. We initialize the first 4 words of memory with a tiny test program. Instruction 0 will be ignored for now and instructions 1, 2 and 3 are all JCC. The first time that instruction 1 is executed (line 5 of the test) C é 1 and PC is incremented to 2. In lines 7 and 9 instruction 1 is executed againm with C equal to 1 and C equal to 0 respectively. Only in the last case it doesn't go on to 2, but instead jumps to 6.

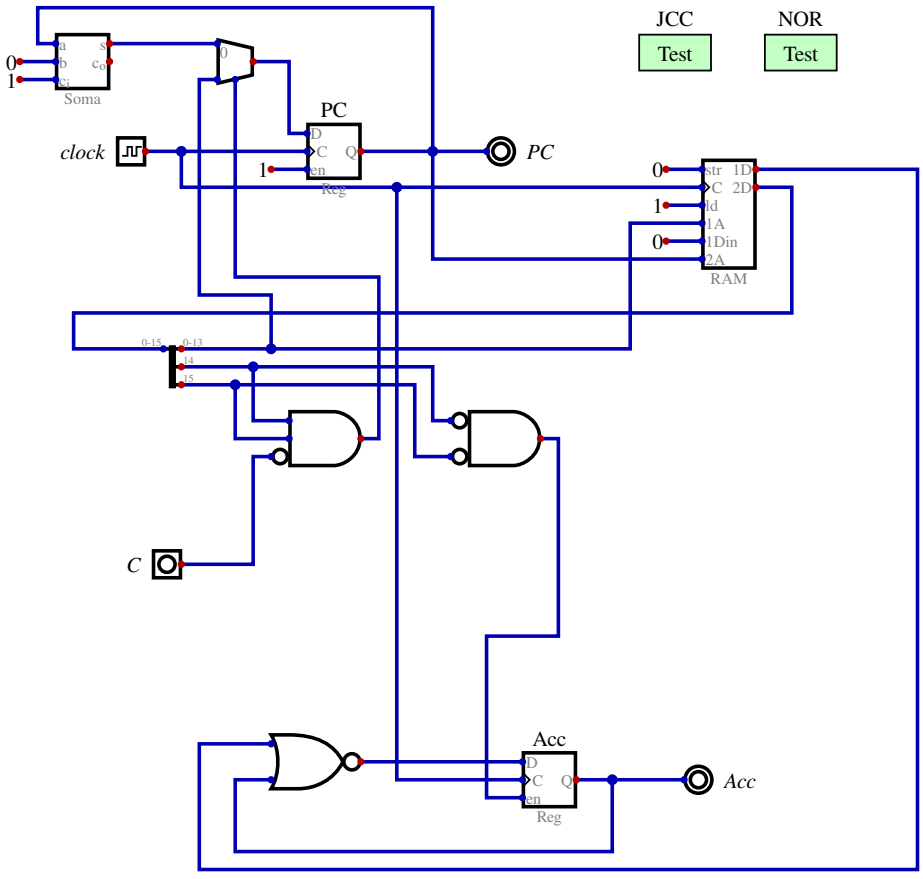


Figure 35: JCC and NOR instructions

The next instruction we will implement is NOR, for which we chose the opcode

0 and 0. We need a new 16 bit register that we will call Acc. We will also have an output with this name to be able to use it in the tests. The 16 bit wide NOR gets one input from Acc and the other from memory, with the result going back to Acc.

First we test JCC to check that it continues to work. Then we create a new test for NOR. The circuit must pass both.

```
clock Acc
program(0x0004, 0x0000, 0x0001, 0x0002, 0xFFFF)
0 0
C 0
C 0xFFFB
C 4
C 0xFFFA
```

Curiously, words 0, 1 and 2 are initially used as instructions but then are used as data.

The STA instruction ("STore Accumulator") will have the opcode of 1 and 0. Acc's output will be the data input for the memory (which was always 0 up to now). And the memory's *str* signal will be activated by this instruction, writing to the indicated address.

After checking that the JCC and NOR tests continue to work, we create a new test for STA.

```
clock Acc
program(0x0004,0x8004,0xC000,0x0000,0xFFFF)
0 0
C 0
C 0
C 0
C 0xFFFF
C 0xFFFF
C 0xFFFF
C 0
C 0
C 0
C 0xFFFF
```

It isn't possible to test STA without also using other instructions. The results of writing data is only visible when you read back that data and for that we need NOR. The test also uses JCC just to keep the code short - a sequence of NOR, STA, NOR, STA, NOR... would be good enough test.

The last instruction, ADD, would be nearly the same as NOR if it wasn't the complication of the carry bit. But as both ADD and NOR write to Acc we need a multiplexer to decide between them.

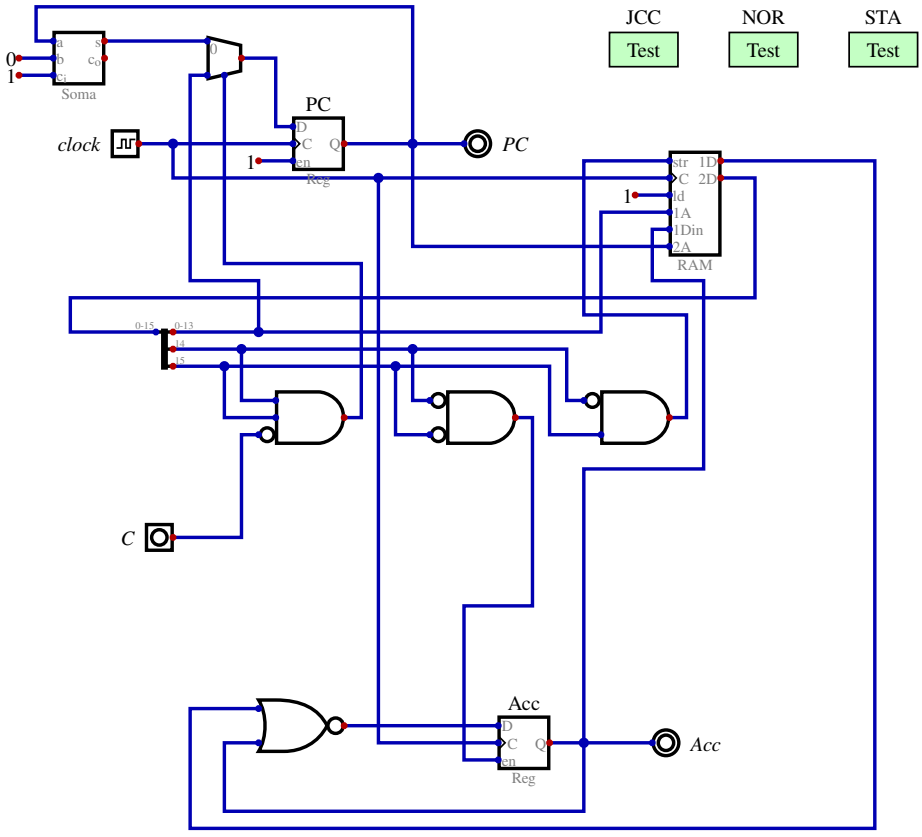


Figure 36: JCC, NOR and STA instructions

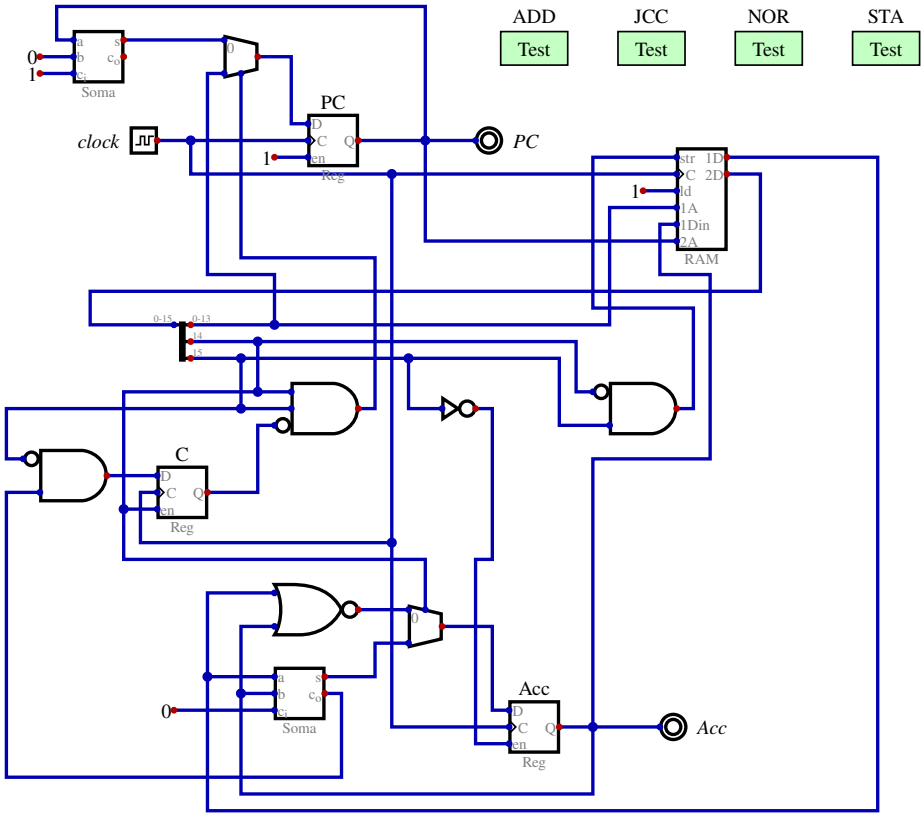


Figure 37: the complete mcpu

Now Acc needs to be enabled both for ADD (opcode 01) as well as NOR (opcode 00), so we can replace the AND gate with the two inverted inputs with a simple inverter. The new adder has the same inputs as the NOR.

The C input is replaced by a 1 bit wide C register. This forces us to change the JCC test. The definition of JCC is that C is cleared independently of whether the jump happens or not. This is used so two JCC in a row will be an unconditional jump (which in other processors is a separate instruction). The carry output of the adder is forced to 0 in the case of a JCC instruction (or a STA, but in that case it doesn't matter).

C needs to be enabled for the ADD (opcode 01) and JCC (opcode 11) instructions, so the lower bit of the opcode can be used directly for that.

With that, MCPU is complete and can execute complex programs.

Software

We have already written small programas for the MCPU16h in the tests we have created. But indicating each instruction directly as a hexadecimal or binary numbers, what we call "machine language", is unfeasible for programs larger than 10 instructions or so. Fortunately, a program called "assembler" can read a text file where each line corresponds to one instruction and some letters indicate the operation ("NOR", "ADD", "STA" and "JCC" in the case of MCPU16h) and generate the corresponding machine language. We can mark certain lines using textual labels and then use the same text as the address field of other instructions so the assembler will take care of the calculation of the actual value associated with each label.

Fancier assemblers, like *GNU as*, allow the definition of "macros" which are texts (possibly with some arguments) which are then exapanded everywhere they are used in a program. In this project we use this feature to get one version of *as* (for the x86 processor, for example) to generate machine code for the MCPU16h. These macros are found in the file *mcpu16.inc*, which has near its start:

```
absStart:
        .macro NOR a
        .word (0x3FFF & ((\a-absStart)/2))
        .endm

        .macro ADD a
        .word (0x3FFF & ((\a-absStart)/2)) | 0x4000
        .endm

        .macro STA a
        .word (0x3FFF & ((\a-absStart)/2)) | 0x8000
        .endm
```

```
.macro JCC a
.word (0x3FFF & ((\a-absStart)/2)) | 0xC000
.endm
```

A program that includes this file can have something like "JCC myLoop" and the corresponding 16 bits will be generated. One complication is that *as* considers a label like "myLoop" to have relative value and can't, in principle, calculate the expression which generates the instruction. The idea is that a program might be organized as several modules and the assembler translates one at a time and a later step links all the modules into the final program. This means that the addresses will only be known after that. But the programs we will write for MCPU16h will be relatively simple and we will not have more than one module. This allows us to define "absStart" as the first thing in the program so the expresison "myLoop-absStart" can have a value that is known to the assembler.

Besides the 4 instructions that the hardware actually understands, we can use macros to define new instructions as short sequences of these instructions. For example: CLR (clears the accumulator to zero), LDA (loads the accumulator with a value), LDP (loads a pointer into the accumulator), LDI (loads into the accumulator indirectly), NOT (inverts the accumulator), JMP (jumps unconditionally), JCS (jumps if C is 1), SUB (subtract the accumulator from a value) and CALL (calls a subroutine). The use of these "pseudo-instructions" generates programas that are somewhat large. Programas for MCPU16h are around 10 times larger than for more reasonable processors.

The pseudo-instructions IN and OUT read and write to the last address in memory and it is up to the circuit to detect this and read and write from a peripheral instead of memory. Two additional macros, STARTCOUNT and COUNT, were defined to control a circuit used to measure speed so we can compare different processors. In this project we will not implement this circuit so these pseudo-instructions will just access the next to the last word in memory without any side effects.

The command "make ALLHEX", among other things, generates the .hex files we will use to initialize memory in our simulations. These are saved in the *hex* directory using the .S source files in the *soft* directory. Programs *as* and *objcopy* are used by *make*. Later on we will use *riscv32-unknown-linux-gnu-as* instead of *as* to handle programs for RISC-V. We use .s (lower case) in the source file names in this case so *make* can decide which assembler to use.

The new circuit is in the lower left side. An AND with 14 inputs indicates when the last memory address (0x3FFF) is being addressed and in this case the new multiplexer selects the data coming from the keyboard for the NOR and ADD instructions (while STA and JCC ignore this data) instead of memory, which is selected for any other address. Two AND gates separate the accesses to the last address for the write (STA) and read (any other instruction) cases. Writes go to the terminal and reads come from the keyboard.

The memory can be configured as "Program memory" and in "Circuit specific

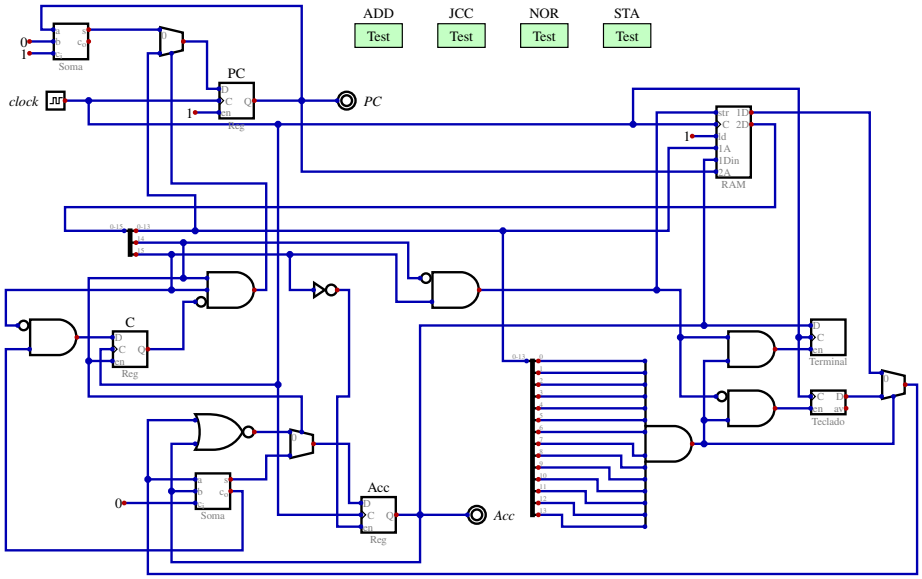


Figure 38: MCPU16h with a terminal and a keyboard

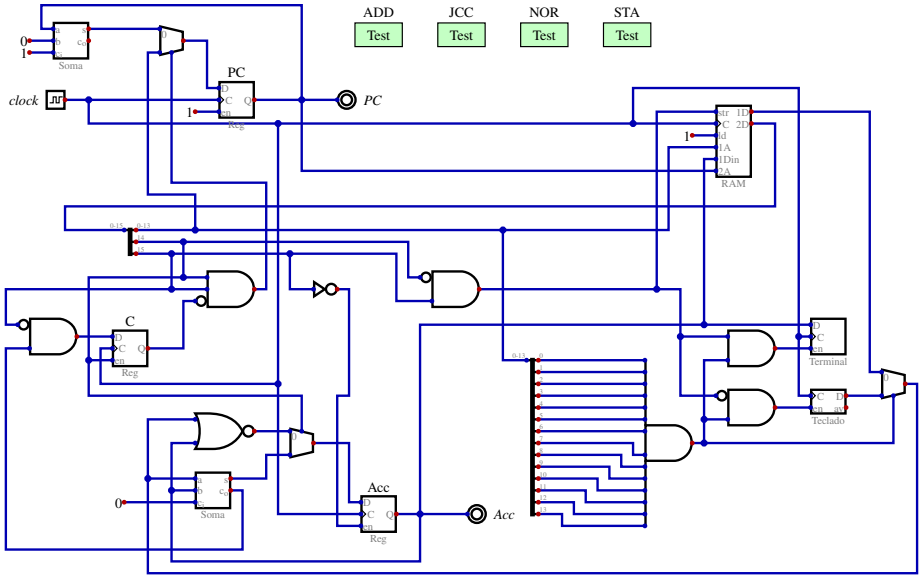


Figure 39: MCPU16h com terminal e teclado

configurations" we can indicate the file *hex/3.01.testTerminal.hex* as "Program file". This was generated from

```
.include "mcpu16.inc"

loop:
        LDA cp
        ADD adInst
        STA 0f
        CLR
0:      .word 0  /* will be replaced */
        STA char
        ADD minusOne
        JCC halt    /* zero terminated string */
        LDA char
        OUT
        LDA cp
        ADD one
        STA cp
        JMP loop

halt:   JMP halt

text:   .string16 "Hello world!"
cp:     .word (text-absStart)/2
char:   .word 0
```

We can step through the execution of this code by repeatedly clicking on the *clock* input. If *Digital* opens a window and prints the expected text in it we can consider that the processor is completely working (the NOR instruction is not seen in this listing but it is used by pseudo-instructions such as LDA).

For a second execution of this same programa we can configure the *clock* input to pulse in real time.

Replacing the memory initialization with *hex/3.02.sine.hex* we can see a sine wave drawn textually on the terminal. The CORDIC algorithm avoids the use of multiplications.

A third example is the interactive game 2048. The terminal shows a 4 by 4 matrix of numbers that have be pushed up, left, down or right by pressing keys in the small window with the title "Keyboard". The file to initialize memory is *hex/3.03.term20248.hex*.

drv32h

In 2010 a researh group at the University of California in Berkeley was developing a circuit which needed to include a processor. After evaluating the available

alternatives they decided that the best option would be to create their own design. After all, this same group had created the RISC processor in 1982 and RISC II in 1983. SOAR ("Smalltalk On A RISC") from 1984 and SPUR from 1988 had continued this tradition. So this new processor would be the fifth in this lineage: a RISC-V.

They defined a very small set of instructions but with room for optional extensions. The base can be 32 bits wide (RV32I or RV32E), 64 bits (RV64I or RV64E) or even 128 bits (RV128I). The "I" and "E" variants are identical except for "I" having 32 register and "E" only 16.

An example of an extension is the "M" one which adds multiplication and division instructions. The basic instructions are sufficient to implement these functions, but having dedicated instructions cana accelerate some applications significantly at the cost of more expensive hardware. Most extensions are like this one where something that was already possible becomes faster. An example of an extension that adds functionality is "A" which introduces atomic operations. Without these new instructions it would not be possible for multiple processors connected to a single memory to coordinate their activities.

With the increase in interest in the project by companies and researchers outside of Berkeley an independent foundation was created to organize the standardization and evolution of the architecture.

In drv32h (RISC-V in Digital with a RV32I base and a Harvard architecture) we will implement only the base instructions and no extensions.

Instructions

Details about RISC-V instructions can be found in the official specification or in the slides of this 2022 course.

The basic RISC-V instructions are 32 bits long, but the bottom two bits are always 1 and 1. The other 3 combinations are used by the "C" extension which defines 16 bit wide instructions to make executable code more compact. The 7 bottom bits are the opcode and without the "C" extension we have 32 possible combinations:

	00...11	01...11	10...11	11...11
..00011	LOAD	STORE	MADD	BRANCH
..00111	LOAD-FP	STORE-FP	MSUB	JALR
..01011	cust0	cust1	NMSUB	reserved
..01111	MISC-MEM	AMO	NMADD	JAL
..10011	OP-IMM	OP	OP-FP	SYSTEM
..10111	AUIPC	LUI	reserved	reserved
..11011	OP-IMM32	OP32	cust2	cust3
..11111	48 bits	64 bits	48 bits	>=80 bits

We will only implement the individual instructions JALR, JAL, AUIPC and LUI and the instruction groups LOAD, STORE, BRANCH, OP-IMM and OP. These are only 9 of the 32 possiblities for the main opcode. The groups of instructions use helper opcodes to select the individual instruction.

The instructions MADD, MSUB, NMSUB and NMADD and the groups LOAD-FP, STORE-FP and OP-FP are for the floating point extensions: "F" (32 bit floating point), "D" (64 bit floating point) and "Q" (128 bit floating point). We won't use them in this project.

The 5 bottom bits are all 1 got instructions longer than 32 bits, but none has been defined so far. The groups OP-IMM32 and OP32 allow RV64I and RV128I to also operate on 32 bits, but in a RV32I these instructions are not used.

Those marked as "reserved" will be used by official extensions while those marked as "custX" are what non official extensions are supposed to use.

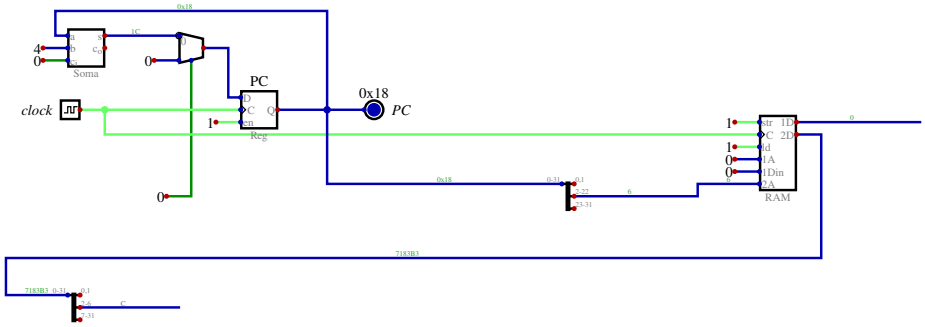


Figure 40: PC in drv32h

Eliminating most of the circuito from MCPU16h we go back to having little more than the PC. For drv32h we increase the PC from 14 to 32 bits (as well as the multiplexer and adder) and we replace the value being added with 4. Before sending the value of PC to be used as a memory address, we discard the bottom two bits which are used to select a byte from a 32 bit word. The width of the memory was also increased to 32 bits but the address only went up to 21 bits (which gives us a total of 8MB). When PC addresse byte 0x18 the memory reads word 6. As the top 9 bits are not connected, the memory will repeat 512 times over the 4GB space.

We initialize memory with *hex/3.04.sine.hex* which is the RISC-V version of the same program that MCPU16h ran as *hex/3.02.sine.hex*. While the latter is a file with 6874 bytes, the RISC-V version is only 373 bytes long.

The instruction is split into the two lowest bits (which should always be 1 and 1), the 5 bits of the main opcode corresponding to the above table and the remaining bits.

The first instruction to be implemeented is JAL ("Jump And Link") which is

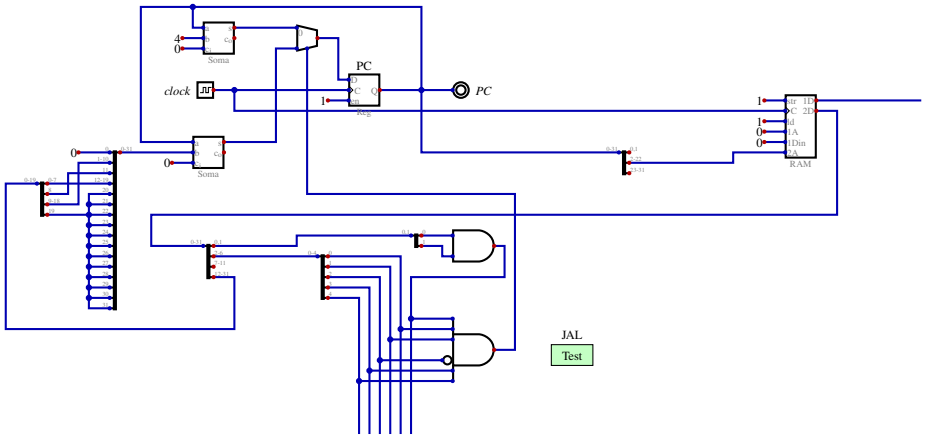


Figure 41: JAL instruction

used for unconditional branches and subroutine calls. There is a field with the destination register but as these don't exist yet this field is simply not connected to anything. An immediate value of 20 bits to be added to the PC is encoded in a somewhat complicated way. A second adder is being used to add the immediate value to the PC and to send the result to the multiplexer that was already there from previous projects. An idea that seems viable would be to have the multiplexer at an input of the original adder, selecting between the constant 4 and the immediate vaue. But this would only work due to JAL still being incomplete. The value to be saved to the destination register is exactly PC+4 and the only way to have both that and PC+immediate is to use two separate adders.

A 6 input AND gate combined with a 2 input one detect that the bottom 7 bits of the instruction correspond to JAL. Their output is used to control the multiplexer.

The test corresponds to:

```
loop:   nop
        jal x31, loop
```

PC alternates between 0 and 4.

Next, we will look at the very similar groups of instructions: OP and OP-IMM.

group	instruction	funct7		funct3	function
OP	ADD	0000000	rs2	000	rd := rs1 + rs2
OP-IMM	ADDI	imm	imm	000	rd := rs1 + imm

group	instruction	funct7		funct3	function
OP	SUB	0100000	rs2	000	rd := rs1 - rs2
OP	SLL	0000000	rs2	001	rd := rs1 « rs2
OP-IMM	SLLI	0000000	imm	001	rd := rs1 « imm
OP	SLT	0000000	rs2	010	rd := rs1 < rs2
OP-IMM	SLTI	imm	imm	010	rd := rs1 < imm
OP	SLTU	0000000	rs2	011	rd := rs1 < rs2 (unsigned)
OP-IMM	SLTIU	imm	imm	011	rd := rs1 < imm (unsigned)
OP	XOR	0000000	rs2	100	rd := rs1 XOR rs2
OP-IMM	XORI	imm	imm	100	rd := rs1 XOR imm
OP	SRL	0000000	rs2	101	rd := rs1 » rs2 (unsigned)
OP-IMM	SRLI	0000000	imm	101	rd := rs1 » imm (un-ssigned)
OP	SRA	0100000	rs2	101	rd := rs1 » rs2
OP-IMM	SRAI	0100000	imm	101	rd := rs1 » imm
OP	OR	0000000	rs2	110	rd := rs1 OR rs2
OP-IMM	ORI	imm	imm	110	rd := rs1 OR imm
OP	AND	0000000	rs2	111	rd := rs1 AND rs2
OP-IMM	ANDI	imm	imm	111	rd := rs1 AND imm

The only difference between the two groups is the lack of a SUBI, but since the immediate value is signed this instruction is unnecessary.

Two new 6 input AND gates indicate the OP and OP-IMM groups. The *PC* output was replaced by a *PC* probe. It takes up less space yet the tests can use

it just the same. We also added probes for *JAL*, *OP* and *OP-IMM* so we can test these signals.

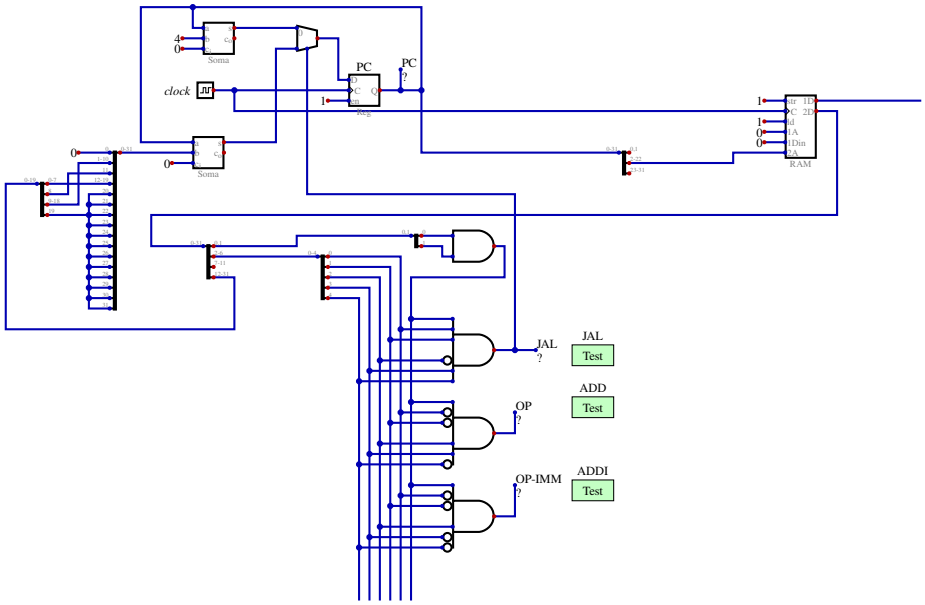


Figure 42: probes

To work correctly one of the things these instructions need is a register bank. RV32I has 32 registers or 32 bits each, but register *x0* (also known as *zero*) ignores all writes to it and always returns 0 when read. After building the register bank by repeating 4 times a block of 8 registers, register *x0* was deleted and where its output used to be a constant with value 0 was inserted.

With the registers connected to inputs and outputs it is possible to test them manually independently from the rest of the circuit.

Writting programas directly in hexadecimal is very tiring and error prone. To develop the tests "as" for RV32I is run directly on the command line so that it reads from the standard input and generates the output when control-D is typed in. The output can be copied as comments in the test. Even so it is each to make mistakes when converting from the "little endian" of the listing to 32 bit numbers.

The 32 bits of the instruction are split into fields as defined in the RISC-V standard. 3 fields of 5 bits each are addresses for the register bank. A 3 bit field can be used directly to control a multiplexer which selects which circuit will produce the result. The first input comes from an adder but the second input of the adder can be inverted for substraction. The 7 most significant bits of the instruction can have only two patterns in the instructions we are implementing: 0000000 or 01000000. The second one (combined with the indication that this

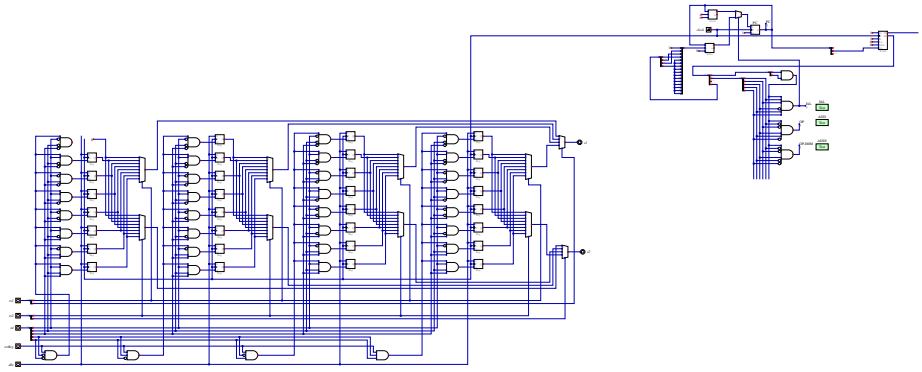


Figure 43: registers

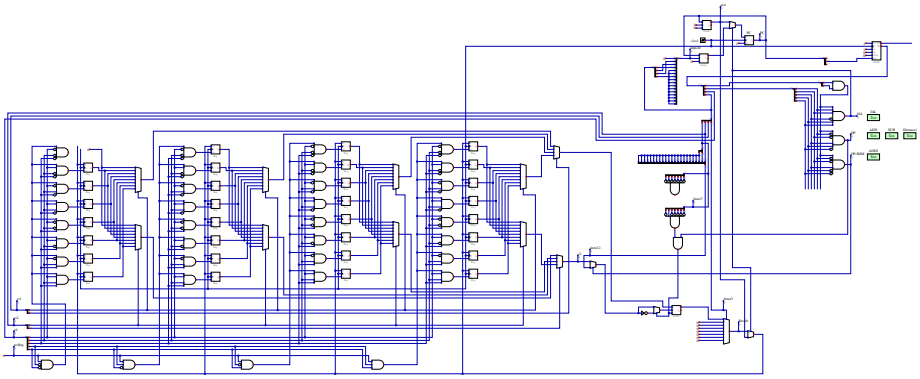


Figure 44: addition and subtraction

is OP and not OP-IMM since these bits can happen as part of an immediate value) transforms the addition into subtraction.

A multiplexer selects the value to be written to a register between the calculated result and the PC+4 adder (for the JAL instruction).

With these instructions it is already possible to have a test program that calculates the Fibonacci numbers.

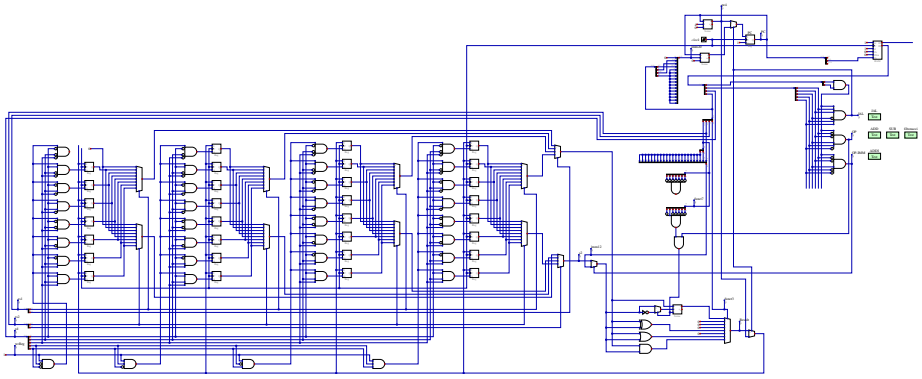


Figure 45: logic operations

With all this infrastructure in place, adding the six logic operations (3 OP and 3 OP-IMM) is very simple: they are 3 gates that are 32 bits wide in the lower right.

For the comparison between two 32 bit nnumbers there are two possible results: if the two numbers are unsigned or if the two numbers use the two's complement representation. It doesn't make sense to compare numbers with different representations.

Here are the results of subtracting two 3 bit numbers at the binary level (with the carry from bit 2 shown to the left before a comma and the carry from bit 1 shown to the right in parenthesis):

	000	001	010	011	100	101	110	111
000	1,000 (1)	1,001 (1)	1,010 (1)	1,011 (1)	1,100 (1)	1,101 (1)	1,110 (1)	1,111 (1)
001	0,111 (0)	1,000 (1)	1,001 (1)	1,010 (1)	1,011 (0)	1,100 (1)	1,101 (1)	1,110 (1)
010	0,110 (0)	0,111 (0)	1,000 (1)	1,001 (1)	1,010 (0)	1,011 (0)	1,100 (1)	1,101 (1)
011	0,101 (0)	0,110 (0)	0,111 (0)	1,000 (1)	1,001 (0)	1,010 (0)	1,011 (0)	1,100 (1)

	000	001	010	011	100	101	110	111
100	0,100	0,101	0,110	0,111	1,000	1,001	1,010	1,011
	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)
101	0,011	0,100	0,101	0,110	0,111	1,000	1,001	1,010
	(0)	(1)	(1)	(1)	(0)	(1)	(1)	(1)
110	0,010	0,011	0,100	0,101	0,110	0,111	1,000	1,001
	(0)	(0)	(1)	(1)	(0)	(0)	(1)	(1)
111	0,001	0,010	0,011	0,100	0,101	0,110	0,111	1,000
	(1)	(0)	(0)	(1)	(0)	(0)	(0)	(1)

In the case of unsigned comparison, the equivalent results are:

	0	1	2	3	4	5	6	7
0	=	>	>	>	>	>	>	>
1	<	=	>	>	>	>	>	>
2	<	<	=	>	>	>	>	>
3	<	<	<	=	>	>	>	>
4	<	<	<	<	=	>	>	>
5	<	<	<	<	<	=	>	>
6	<	<	<	<	<	<	=	>
7	<	<	<	<	<	<	<	=

Comparing the two tables we see that "unsigned less than" (LTU used in the SLTU and BLTU instructions) is just the NOT of the bit 2 carry.

For signed comparisons we have these results:

	+0	+1	+2	+3	-4	-3	-2	-1
+0	=	>	>	>	<	<	<	<
+1	<	=	>	>	<	<	<	<
+2	<	<	=	>	<	<	<	<
+3	<	<	<	=	<	<	<	<
-4	>	>	>	>	=	>	>	>
-3	>	>	>	>	<	=	>	>
-2	>	>	>	>	<	<	=	>
-1	>	>	>	>	<	<	<	=

It is a lot more complicated to see which cases are signed less than, but if we look closely we can see that carry for bit 2, carry for bit 1 and bit 2 of the result always have an odd number of 1s in these cases. The XOR of the two carry bits is available on many processors as the V ("overflow") flag that indicates that the bits produced as the result are actually wrong. With three bits it is not

possible to represent all reauls. If we add 5 and 5 unsigned the answer will be ten and that would need 4 bits to be represented. If we add 2 and 3 signed the answer wiil be five, but in two's complements this will be understand as -3. A positive V indicates these cases saying the software should discard the result. A XOR between V and the sign of the result indicates the "unsigned less than".

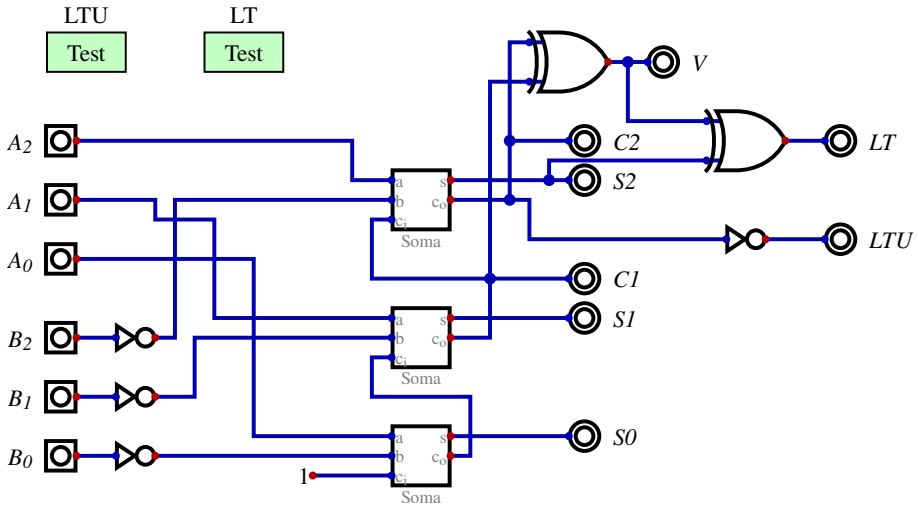


Figure 46: test of signed and unsigned comparisons

Normally we would make an effort to reuse the circuito for the SUB instruction to also implement SLT, SLTU, SLTI and SLTIU (besides the conditional branches, which we haven't seen yet). But here we use a separate adder which handles the lowest 31 bit and another adder just for the most significant bit, which generates all the signals we need for the comparisons.

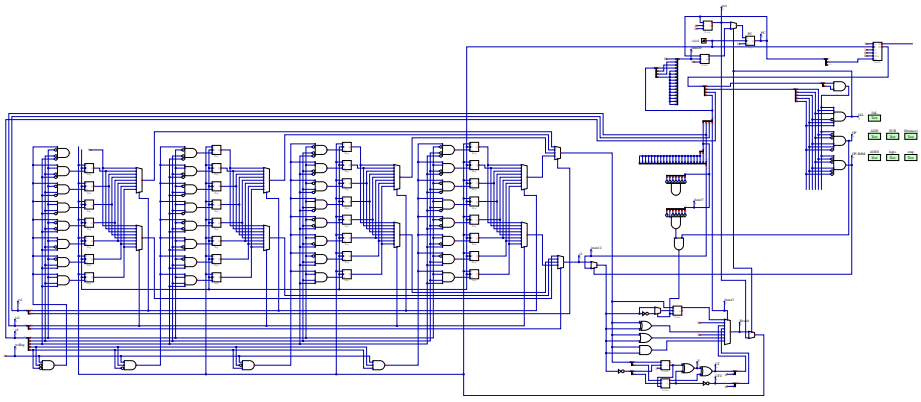


Figure 47: instruções de comparação

The shift instructions need a lot of additional hardware. To shift to the left by a single bit it is enough to just add the value to itself. But to shift to the right we can use a multiplexer to select a shifted version of the value (this is just wires). The rightmost digit will simply disappear and we need to decide what to put into the leftmost digit. For a unsigned number the new digit will be 0 while for a signed number the correct thing is to copy the previous value of the same bit.

Many processors either only had instructions to shift a single digit or allowed any number of digits but took one clock cycle per digit. The first option is not compatible with the RISC-V standard while the second doesn't fit into the style of the other drv32h instructions which all execute in a single cycle.

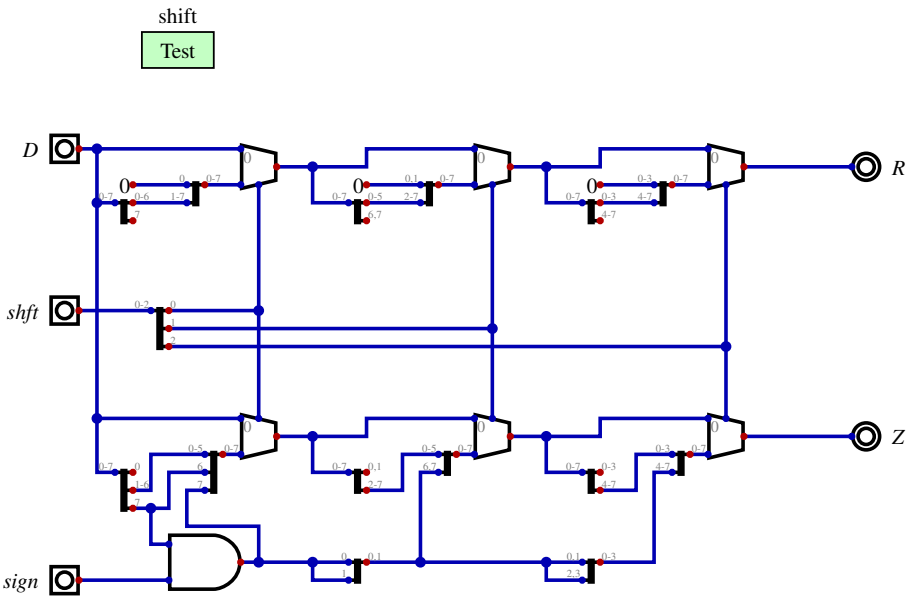


Figure 48: shift tests

One way to shift by any number of digits in a single clock cycle is to replace the single shifter through which the data passes N times with N shifters in series. But for 8 bits we would need 8 shifters (multiplexers) and for 32 bits it would be 32 shifters, which is a huge and slow circuit.

A more elegant solution is to make some of these shifters move the bits more than one digit at a time. The first shifter would move it by 1 digit, the second by 2 digits, the third by 4 digits, then 8, 16 and so on. Something interesting happens in this case - each bit of the second operand can directly control one of the shifters. So if the operand is 5 (00101 in binary) we shift by 1, but not by 2, we do shift by 4 but not by 8 or 16. The circuit is greatly simplified. In the test circuit we use 3 shifters in series for 8 bits.

The top half of the test circuit is the left shift. At each stage the multiplerxer

allows the data straight through if the corresponding bit of the second operand is 0 or it selects a shift created entirely with just wires.

On the bottom half the shift to the right is a little more complicated since instead of the value to be inserted being always 0 (but with twice the number of bits at each stage) it is bit 7 of the input data or (via an AND gate) 0, depending on whether we want a SRL or a SRA instruction. Expanding this idea to 5 stages of 32 bits we will have the remaining instructions for OP and OP-IMM.

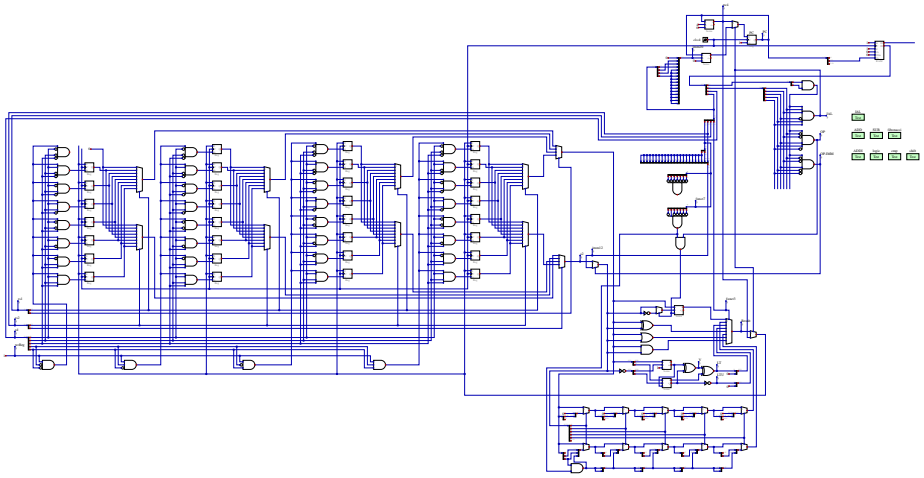


Figure 49: shifts

The next instruction is JALR, which is halfway between JAL and the various OP-IMM. In reality we will want to be like OP-IMM for JALR„ LOAD and STORE so we will modify the signal controlling the multiplexer that selects between R[rs2] and imm12. In the same way we will modify the signal to the multiplexer that selects between the result and PC+4 to be written to the register bank. A new multiplexer selects between the calculation that JAL was already doing and the result from the ALU (which will be an addition since in the JALR instruction funct3 is 0).

In the test for JALR we already can have a small subroutine which is called using the JAL instruction and which returns using JALR. Another use for JALR is to follow function pointers and also to implement calculated jumps, like for the switch/case structure in programming languages.

Still in jumps, we can implement the conditional branches to finish this area. These compare two registers and if the comparison is what was asked for then the immediate value (also 12 bits but taken from different bits than OP-IMM and JALR) is added to the PC, but if the comparison doesn't work out then the instruction doesn't do anything. We have already implemented 4 of tthe 6 necessary comparisons as part of SLT and SLTU. Missing is being able to test that two numbers are equal. Generally this is done by checking if the subtraction

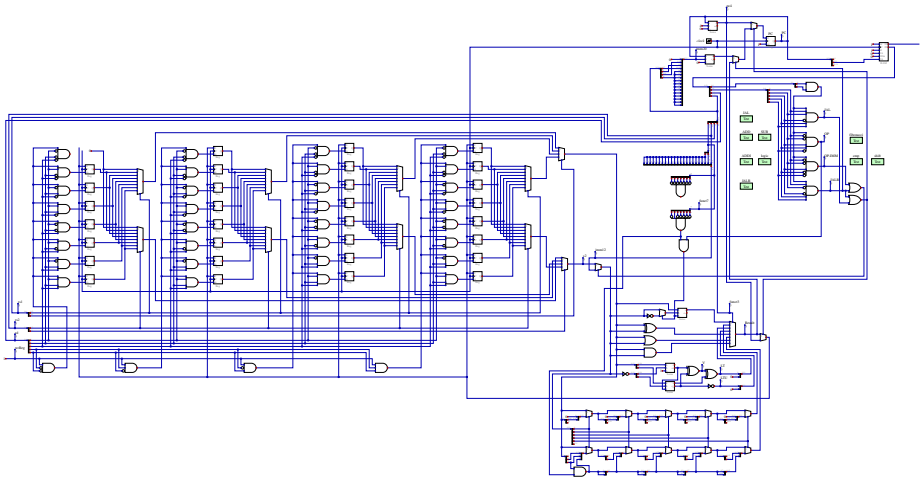


Figure 50: JALR instruction

(which we are already doing as part of the other comparisons) had a zero result. But we are also doing a XOR of the two operands and if any bit (tested with a 32 input OR) is 1 then the numbers are no equal. The same funct3 that selects from the ALU results can control a new 8 input multiplexer to choose the comparison to be done. Two of the possible comparisons are not defined (we hardwired them to false) and the others are different, LT and LTU as well as their inverses (equal, GE and GEU).

Conditional branches are the first instructions we are implementing that don't store a result into the register bank (STORE is the other one we will see). We use the *branch* signal before it is combined with the comparison since we avoid writing to a register independently of whether we branch or not.

We can make use of the adder JAL uses with a multiplexer to select between the imm20 that it adds to the PC and the special immediate for branches. The new immediate is choosen when we have a branch and the condition is true. The OR with JAL and JALR is expanded to 3 inputs so this signal can also select PC+imm instead of PC+4.

Upon the creation of the branch test we detected a mistake that was there from the begining: the *str* signal for the memory was always active, so every single instruction was also writing 0 to address 0 with the second memory port. Since the branch test was the first to try to execute the instruction at 0 a second time it was the first time the problem was encountered (and now fixed). This is, unfortunately, typical. The tests are written to try to observe the problems we can imagine, but the problems we don't imagine and don't directly test for exist.

The next two instructions (LUI and AUIPC) are not necessary, but they are convenient. Most other processors don't have something equivalent. The use of

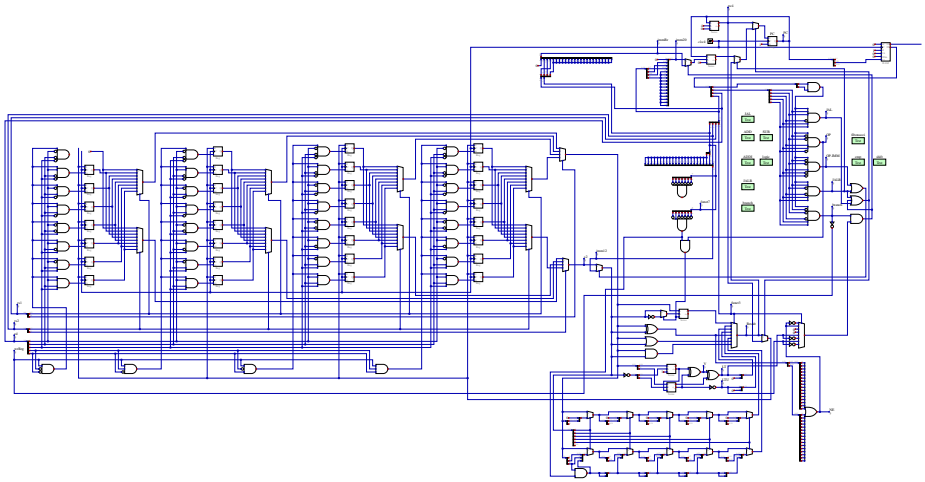


Figure 51: branch instructions

immediates that are only 12 or 20 bits seems to be a huge limitations, but the already implemented instructions can be used to load an arbitrary 32 bit value (0x12345678, for example) in a register:

```
addi x6,zero,(0x12345678>>22)&0x7FF
slli x6,x6,11
addi x6,x6,(0x12345678>>11)&0x7FF
slli x6,x6,11
addi x6,x6,0x123445678&0x7FF
```

Implementing LUI doesn't create any new functionality, but it reduces the above code to just two instructions:

```
lui x6,0x12345678>>12
addi x6,x6,0x12345678&0x0FFF
```

The assembler offers the "li x6,0x12345678" pseudo-instructions that expands to the above code. It also eliminates the LUI if the constant happens to fit into 12 bits. It also takes into account that the immediate value for ADDI is signed and adjusts the immediate value for LUI if necessary so the final result is correct.

Since we can load any 32 bit value into a register, the JALR can jump to anywhere in memory and the limit of the JAL instruction isn't really a problem. But JAL is a relative jump, which allows a code fragment to be copied to a different address and still work (this is needed if we want to join several modules into a single executable). We can calculate a relative destination at the cost of an extra register:

```
li x6,0x01000000
jal x7,next
```

```
next:   addi x6,x6,x7
        jalr zero,-4(x6)
```

The first 3 instructions can be replaced with a AUIPC and then the JALR doesn't need to compensate for the PC+4 of the JAL but instead it supplies the bottom 12 bits of the relative destination (and the immediate value of the AUIPC might need to be adjusted if the JALR's immediate is negative).

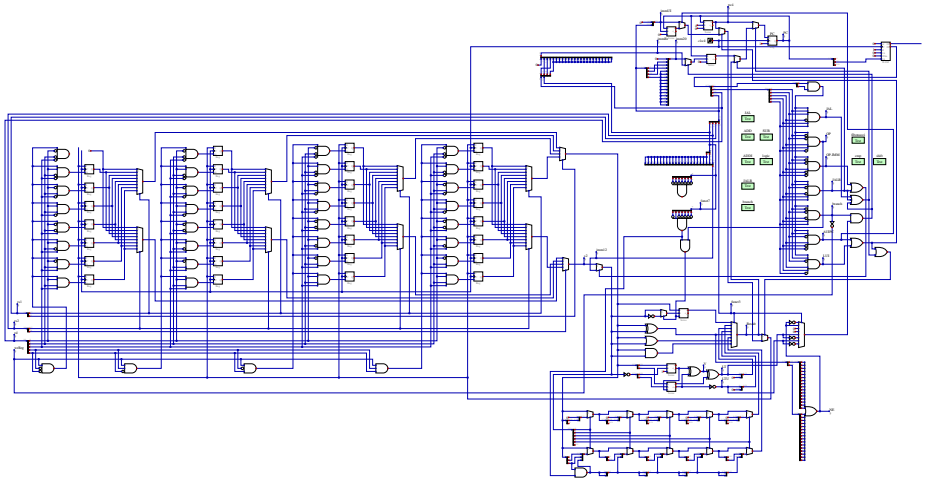


Figure 52: LUI and AUIPC instruction

To check that we are on the right path:

instruction	PC :=	reg[rd] :=
OP	PC+4	Result
OP-IMM	PC+4	Result imm12
JAL	PC+imm20	PC+4
JALR	Result imm12	PC+4
cond branch yes	PC+immBr	x
cond branch no	PC+4	x
AUIPC	PC+4	PC+immUI
LUI	PC+4	immUI
LOAD	PC+4	dMem (rs1+imm12)
STORE	PC+4	x (rs1+immS)

Based on this we introduced yet another adder (for PC+immUI) and two multiplexers: one to select between PC+4 and the second one, which in turn selects between immUI and PC+immUI. Two more OR gates expand the cases which two control signals are set to include the new instructions.

The table includes the LOAD and STORE instruction groups. Without them everything works but we can use only 31 data words of 32 bits each in our program. The term “RISC” is somewhat ambiguous: is it “computer with a reduced set of instructions” or “computer with a set of reduced instructions”? An alternative is to call it a “load/store architecture” to call attention to the fact that only special instructions can access memory. The MCPU16h processor has both a reduced set of instructions (only 4) and its instructions are very simple. But NOR and ADD not only do what their name says but also access memory, so it isn’t a load/store architecture. The drv32h processor we are finishing here certainly is.

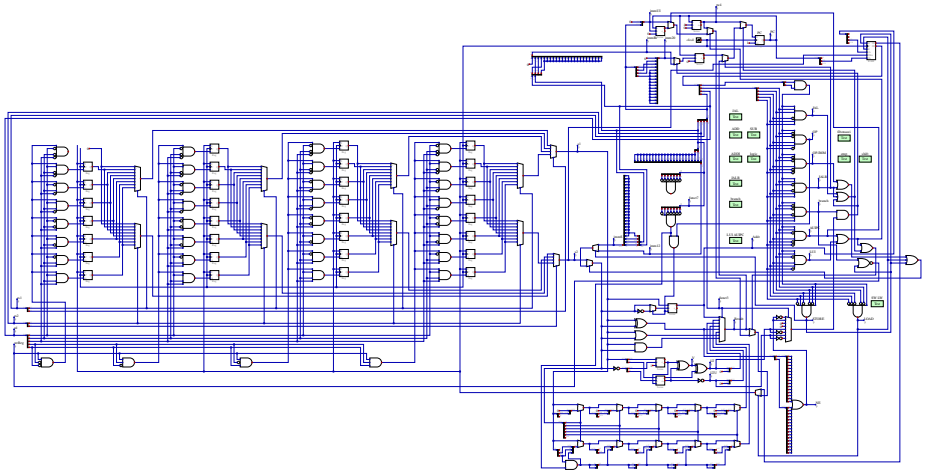


Figure 53: SW e LW instructions

The STORE signal can directly connected to the *str* signal of the memory, while LOAD can be connected to *ld* in the memory. The memory data input can come from the *s2* output from the registers and the address can come from the adder in the ALU because we will use funct3 to select between variations of LOAD and STORE so the ALU result might not be what we want.

The NOT that was used to stop the register bank from being written to by the conditional branch instructions was replaced by a NOR so that STORE doesn’t write either, as indicated in the above table.

Load, on the other hand, does write to the register bank but not the ALU result nor PC+4. We need a new multiplexer to receive data coming from the memory.

Now we can read from and write to memory, but only whole words at a time. It might sometimes be more convenient to work with 16 bit information or even access individual bytes (8 bits). The funct3 field indicates the width of the data to be transferred. For reading from memory we can just add a circuito which uses the bottom two bits of the address and the width information to extract

the relevant bits and convert the result to a 32 bit number to be written to a register.

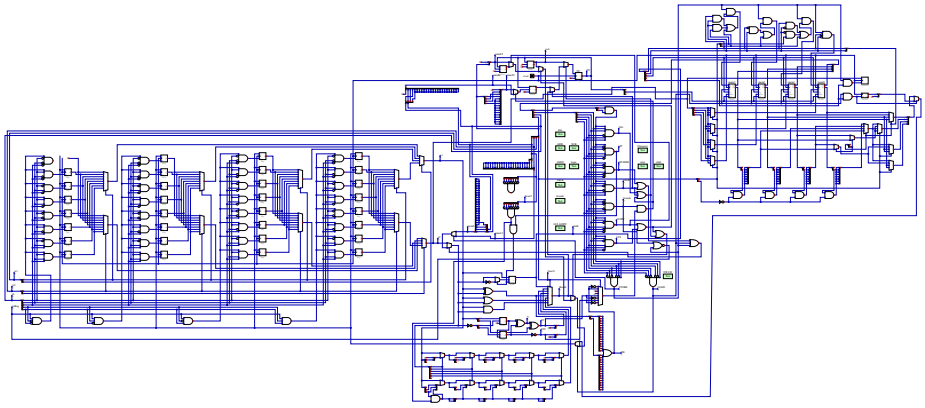


Figure 54: complete drv32h

Writing just part of a word is more complicated. One option is to read the whole word, change the bits we want and write back the whole word. Instead of that we will use 4 separate memory circuits, each 8 bits wide. Now we have 4 separate *str* signals that we can control individually.

funct3_1	funct3_0	A_1	A_0	str_3	str_2	str_1	str_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0
0	0	1	1	1	0	0	0
0	1	0	0	0	0	1	1
0	1	1	0	1	1	0	0
0	1	X	1	X	X	X	X
1	0	0	0	1	1	1	1
1	0	X	1	X	X	X	X
1	0	1	X	X	X	X	X
1	1	X	X	X	X	X	X

These signals are all 0 if the instruction isn't STORE. Besides the *str* it is necessary to reorganize the data. If we are writing a byte, for example, then bits 7 to 0 must go to all 4 memories. This is achieved with 4 multiplexer with 4 inputs of 32 bits each and controlled by the bottom two bits of funct3.

This is all supposing that only aligned accesses are allowed. Bytes can be read or written at any address while 16 bit words only at even addresses and 32 bit words only at addresses that area multiple of 4.

For reading we also use multiplexers:

funct3_1	funct3_0	A_1	A_0	byte3	byte2	byte1	byte0
0	0	0	0	sign0	sign0	sign0	mem0
0	0	0	1	sign1	sign1	sign1	mem1
0	0	1	0	sign2	sign2	sign2	mem2
0	0	1	1	sign3	sign3	sign3	mem3
0	1	0	0	sign1	sign1	mem1	mem0
0	1	1	0	sign3	sign3	mem3	mem2
0	1	X	1	X	X	X	X
1	0	0	0	mem3	mem2	mem1	mem0
1	0	X	1	X	X	X	X
1	0	1	X	X	X	X	X
1	1	X	X	X	X	X	X

The most significant bit of the address is used to select the peripherals (keyboard and terminal). Reserving a whole 2GB for that is very wasteful but the focus of this project is simplicity and not efficiency.

4. FPGAs and Shin JAMMA

5. Video and Audio

6. Pegasus 42

A. History

From 1977 to 1991 Brasil had a "reserved market policy" for small scale computers. At the time nearly all imports were forbidden, from automobiles to chocolate including computers. But for other products a foreign company like Volkswagen or Nestlé could open factories in the country and sell locally. The reserved market didn't allow this for the case of mini and microcomputers.

In times of high inflation and no economic growth, the computer sector was one of the few that was greatly expanding. This attracted investment from non related areas, specially due to the opportunity of copying foreign products without having to deal with competing with the originals. It was in this context that a group from the financial sector contacted the lawyer Amaro Moraes e Silva Neto for ideas, and Amaro told them that one thing nobody was making was a children's computer. He said he knew a computer genius who could develop such a product.

He was thinking of Fábio Cavalcante da Cunha, who was officially an electronic engineering student at the Polytechnical School of the University of Sâo Paulo

Figure 55: Amaro Moraes e Silva Neto

(Poli-USP) but was then more focused on his job at a computer store.

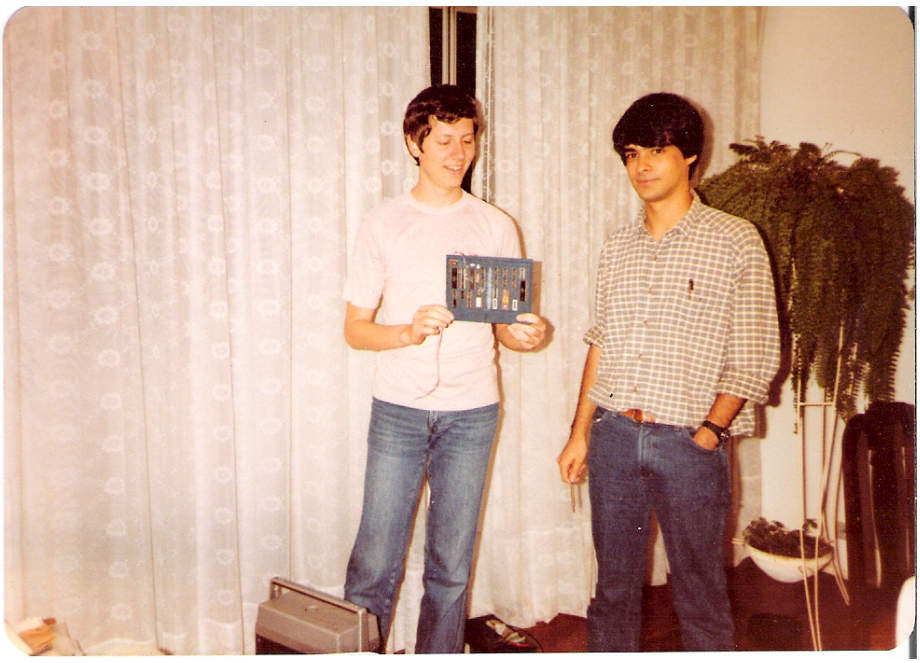


Figure 56: Jecel and Fábio with the prototype's front side

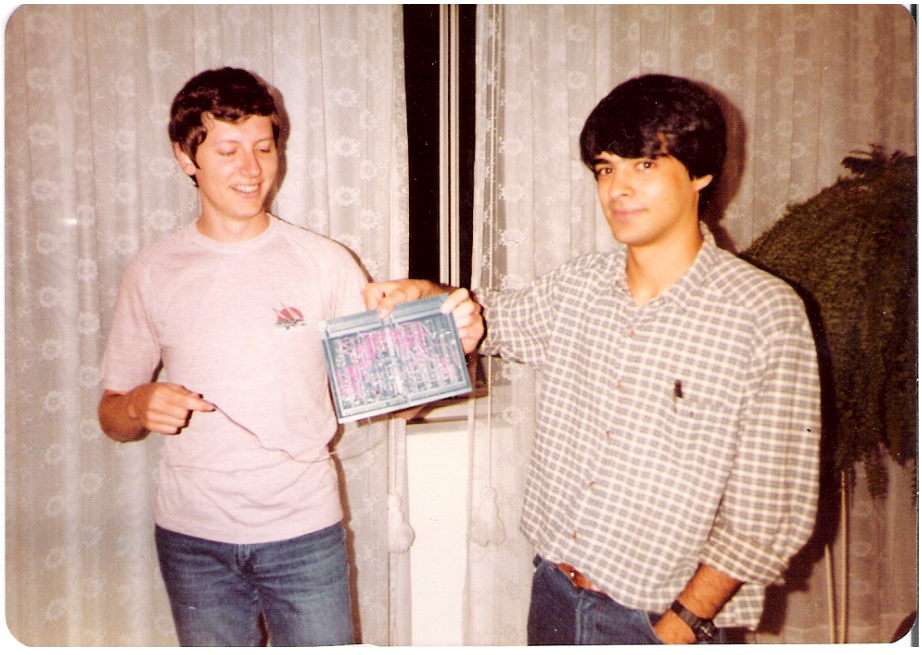


Figure 57: Jecel and Fábio with the prototype's wiring

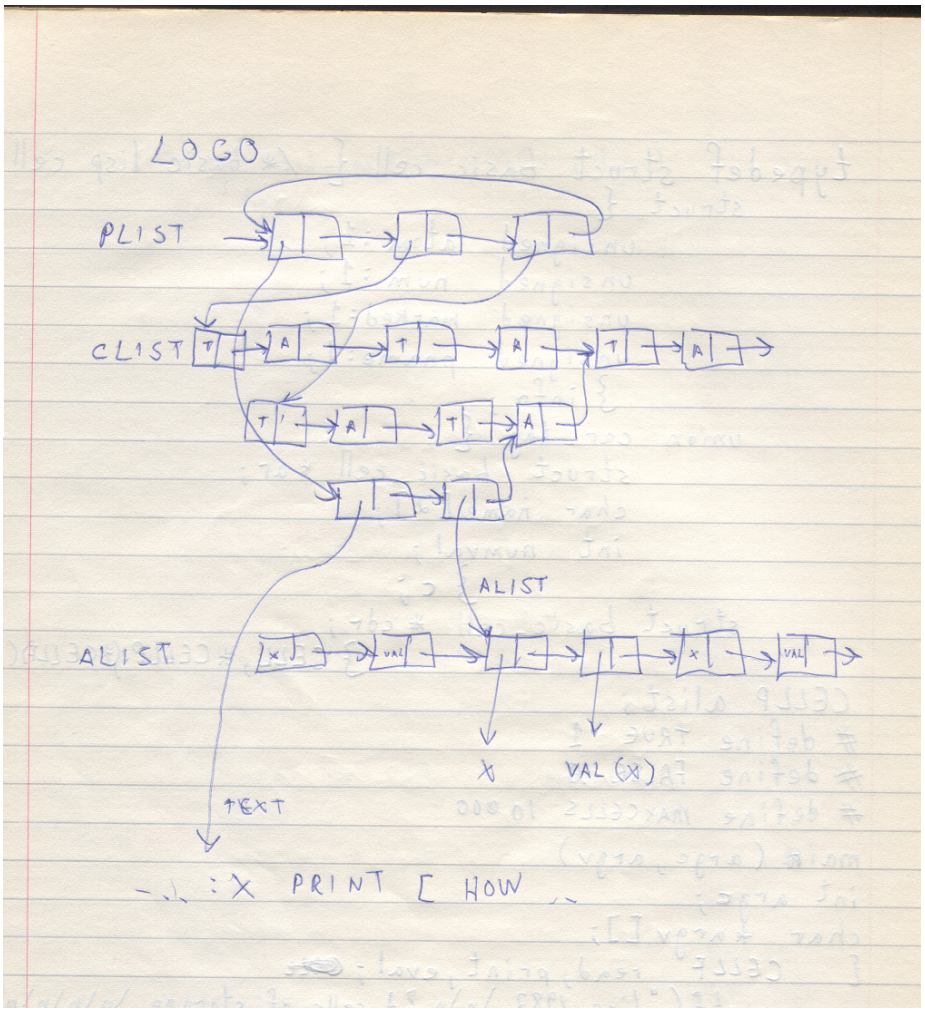


Figure 58: Logo's structure

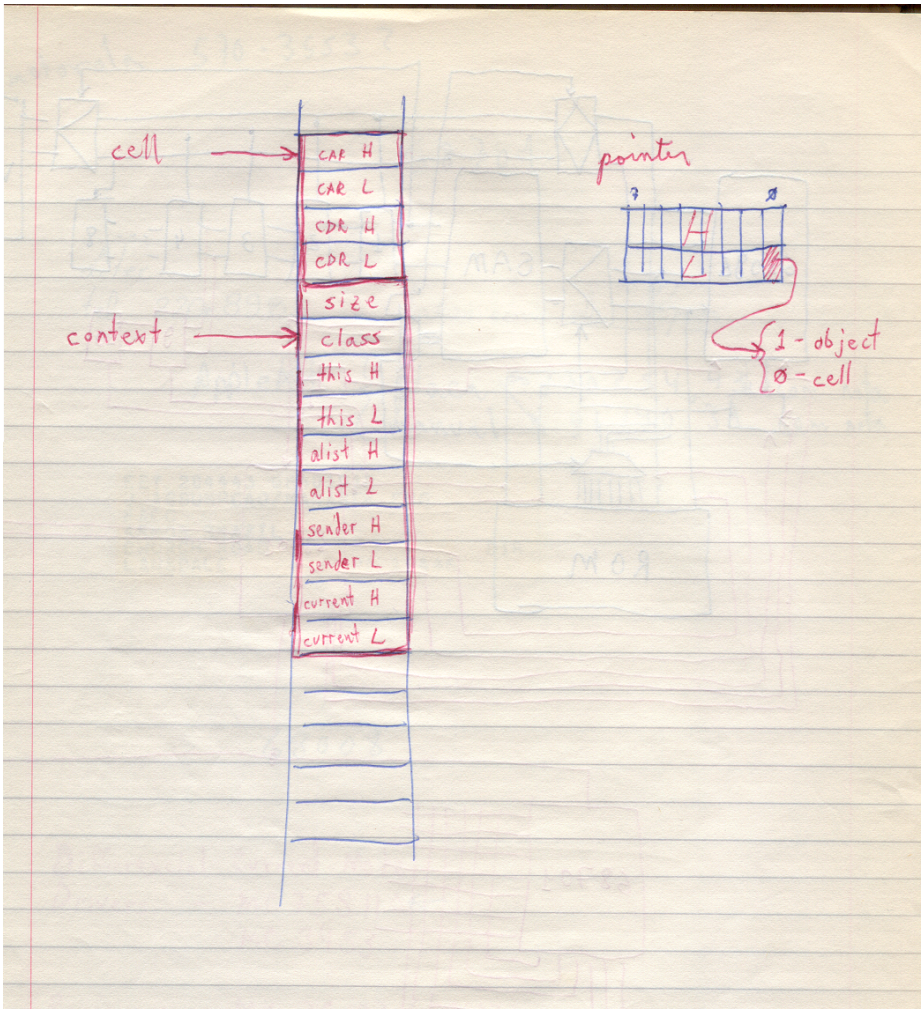


Figure 59: object format

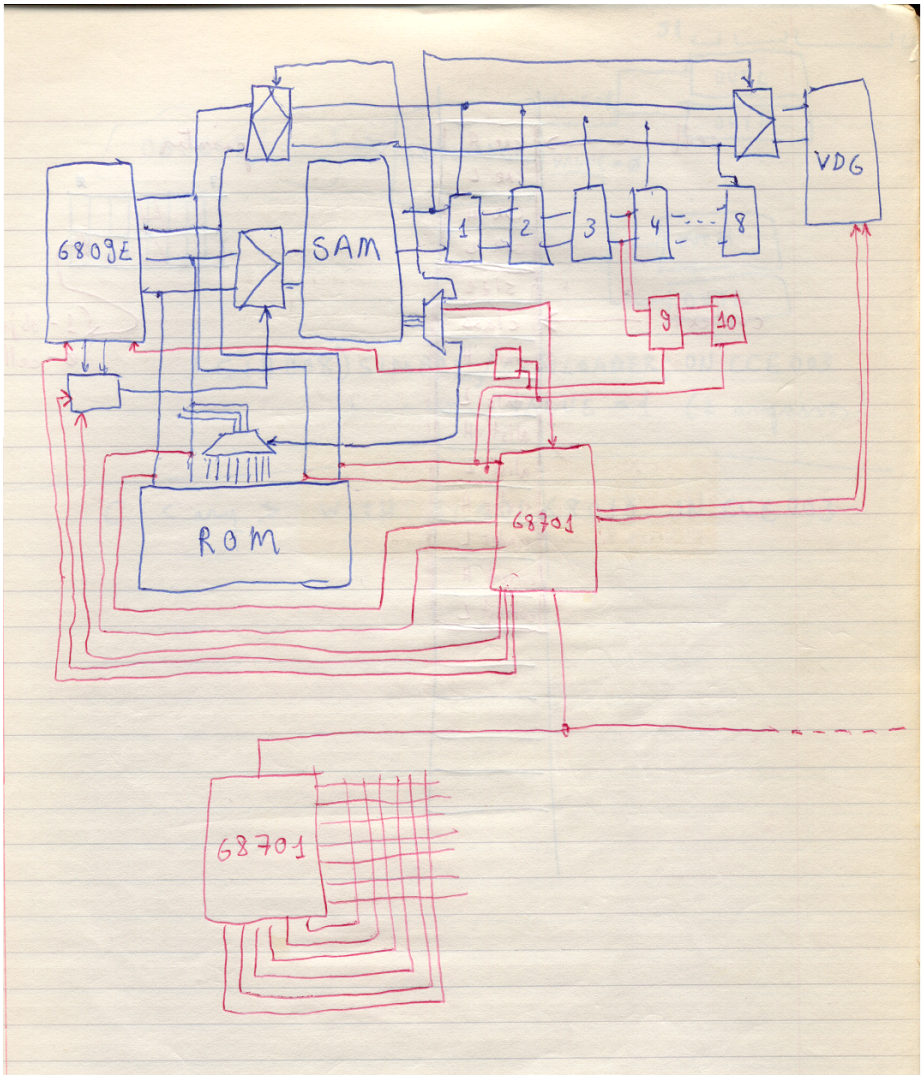


Figure 60: Pegasus schematic

Com linguagem Logo, a máquina de Fábio é inédita

19 anos, aficcionado, Fábio inventa a nova eletrônica

Com seu computador, ele se torna um pioneiro

JOSÉ EDUARDO MENDONÇA

Influenciado pelo pai, um rádio-amador, Fábio Cavalcanti resolveu, aos sete anos de idade, desmontar um rádio para ver como as válvulas funcionavam. Ganhou uma bronca, e o pai um rádio que nunca mais funcionaria. Além disso descobriu, com a experiência, a paixão pela eletrônica que o levou agora, aos 19 anos, a fabricar seu próprio computador.

Aos 13 anos, Fábio fez um curso de computação e o "arquivou na cabeça", devido a interesses que julgava mais apropriados no momento, como seu autorama e os aeromodelos. Mais tarde, no colégio, começou de novo a "ler revistas especializadas e montar pequenos circuitos". Tentou fazer um curso mais avançado e os que encontrou "limitavam demais o conhecimento". Mais uma vez, esqueceu temporariamente sua paixão para enfrentar o cursinho e entrar na Politécnica no curso de engenharia eletrônica, depois de já ter conseguido uma vaga na universidade de São Carlos, para onde migrou abandonando sua cidade natal, Londrina.

Seu computador, o Projeto Pegassus, para a linguagem Logo (ver página "Computadores"), começou a nascer com o exercício de programação em calculadoras e a busca constante de mais informações. Na época, "78, 79, ainda não existia este clima de hoje com os computadores, só mexia com eles o pessoal de processamento de dados em grandes computadores". Fábio aprendeu "tudo que podia saber uma calculadora, depois de fuçar durante um ano", e optou pelo caminho natural: em 1983 foi para os Estados Unidos, mais especificamente para o coração da indústria de computação, o laboratório do Massachussetts Institute of Technology, na Universidade de Stanford, em pleno Vale do Silício.

Um de seus encontros lá foi tão emocionante quanto o encontro, digamos, de um guitarrista daqui com Frank Zappa. Fábio trocou horas de papo com seu guru Marvin Minski, um dos papas da inteligência artificial, e conseguiu provar para si mesmo "que podia estar lá". A mágoa foi não ter conseguido uma bolsa de graduação e ter de "assumir o caminho de volta".

"Se eu tinha de fazer alguma coisa aqui, senti que teria de fazer como sempre fiz: sozinho." Armado de dezenas de livros, estimulado por dois ou três colegas de escola também aficcionados pelos computadores, utilizando seu TRS-80 da Radio Shack, Fábio partiu para seu projeto. Antes, outra decepção: ofereceu seus serviços à Poli, disse que tinha estado no MIT e que gostaria de colaborar. Ouviu como resposta que suas idéias não tinham aplicação comercial imediata. "Parece que dentro da escola não acreditam muito no valor dos alunos, e assim nós temos de criar fora dela."

O computador de Fábio será, diz ele, "possivelmente o primeiro do mundo a vir já com a linguagem Logo, e não com a Basic." A idéia: "Aproximar as crianças dos computadores. Tem cor, som, pode até mesmo programar jogos e serve para crianças a partir de 4 anos até a adolescência, ou mais. Há também excelentes recursos para aplicações mais maduras." O projeto Pegasus ainda não foi enviado para a necessária avaliação e aprovação da Secretaria Especial de Informática, mas, segundo um de seus autores (o outro é o colega Jacel Mattos de Assumpção, também um autodidata, e "um dos poucos gênios" que Fábio conhece), "a máquina já está muito bem definida".

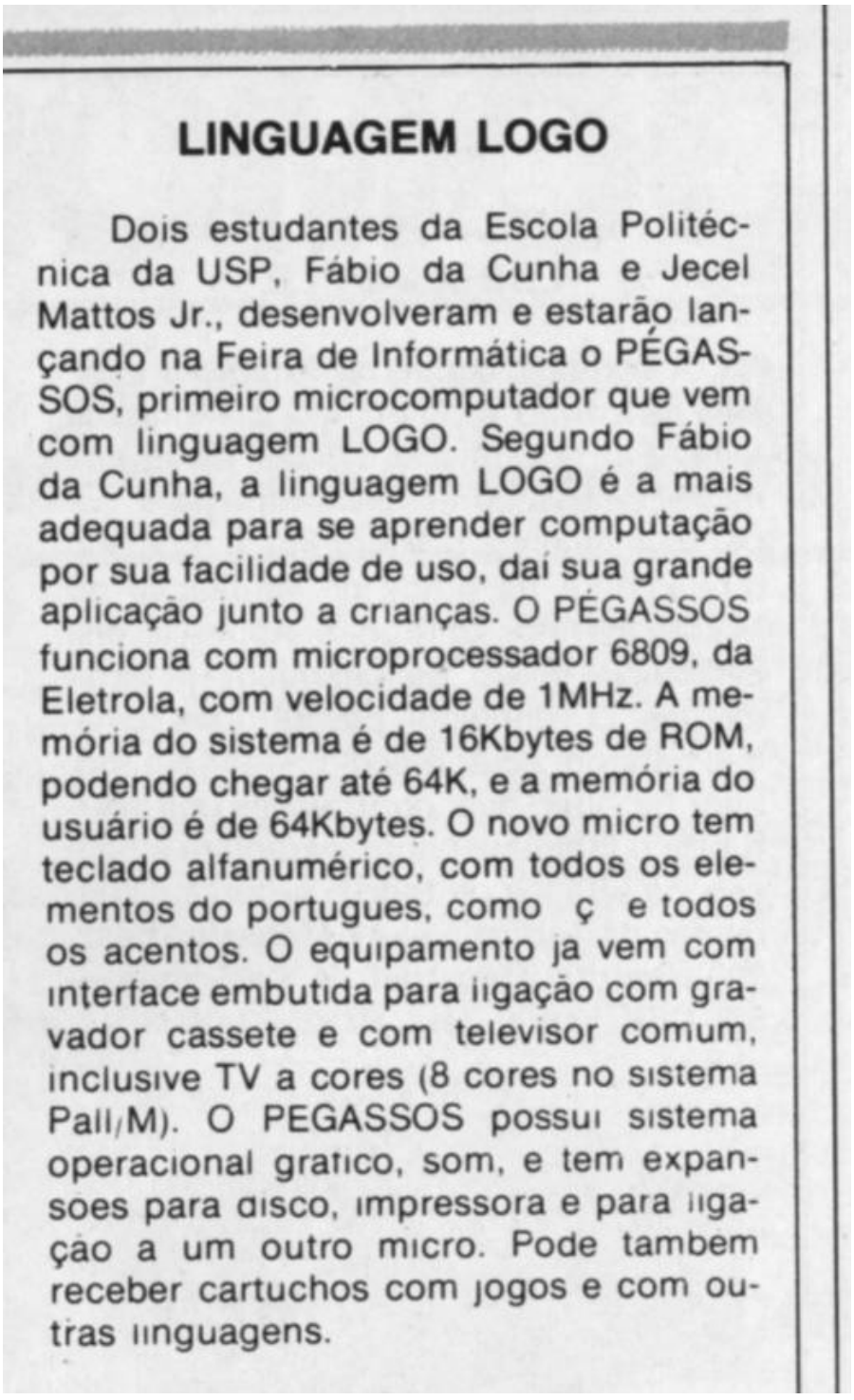
O projeto Pegasus deverá ser comercializado por cerca de 300 mil cruzeiros, tem 64 k bytes de memória, 8 bits e utiliza um chip 6809 da Motorola. Fábio enfatiza que seu computador não é de forma nenhuma um "game", passatempo que critica: "O videogame pode estimular, mas para uma posição errada. As pessoas podem aprender a utilizar o computador como brinquedo, e apenas assim. Aí ele não estará servindo como ferramenta de aprendizagem."

Além de estudar na Poli e dar aulas de computação na Emarés para crianças, Fábio Cavalcanti "troca figurinhas" com grupos de usuários de computadores Apple e TRS-80. Há, no entanto, poucos jovens. "Isto está apenas começando. Os mais novos aprendem diferente. Os mais velhos vai falar como o computador funciona, e a garotada vai perguntar como é que eu faço."

Como profissionais de outras áreas, Fábio e seus colegas têm sua galeria de ídolos e seus livros e passatempos prediletos. Exemplos de pessoas quentíssimas: Wolzniack, um dos inventores do Apple (mais "fera" que o outro, Steven Jobs, tido pelos mais puros como comerciante), Scott Adams, genial escritor de softwares para games em computador; Leo Chrispopherson, programador de animações e Nolan Bushenell, inventor do Atari.

Todos lêem Sherlock Holmes por seu gênio de dedução, os livros de Tolkien por sua magia, o já clássico "Godel, Escher e Bach", de Douglas Holfstadter, um estudo comparativo entre o matemático, o arquiteto e ilustrador e o músico. Adoram assistir "Star Trek" (Jornada nas Estrelas) e cultuam em forma de livros ou filmes a série "The Twilight Zone" (Além da Imaginação). Jogam horas seguidas o jogo de computador chamado "Adventure", tentando sempre superar suas próprias marcas. Com uma cultura semelhante a esta em todo o mundo, estes jovens aficcionados dos micros e da robótica começam a criar a tecnologia do futuro.

Figure 61: news item in Folha de São Paulo



LINGUAGEM LOGO

Dois estudantes da Escola Politécnica da USP, Fábio da Cunha e Jecel Mattos Jr., desenvolveram e estarão lançando na Feira de Informática o PÉGASSOS, primeiro microcomputador que vem com linguagem LOGO. Segundo Fábio da Cunha, a linguagem LOGO é a mais adequada para se aprender computação por sua facilidade de uso, daí sua grande aplicação junto a crianças. O PÉGASSOS funciona com microprocessador 6809, da Eletrola, com velocidade de 1MHz. A memória do sistema é de 16Kbytes de ROM, podendo chegar até 64K, e a memória do usuário é de 64Kbytes. O novo micro tem teclado alfanumérico, com todos os elementos do portugues, como ç e todos os acentos. O equipamento ja vem com interface embutida para ligação com gravador cassete e com televisor comum, inclusive TV a cores (8 cores no sistema Pall/M). O PEGASSOS possui sistema operacional gratico, som, e tem expansoes para disco, impressora e para ligação a um outro micro. Pode tambem receber cartuchos com jogos e com outras linguagens.

Figure 62: news item in Microsistemas magazine

Brasil coleciona casos de sucessos e fracassos

A saga de Steven Jobs e Stephen Wozniack não contagiou apenas jovens americanos. A idéia de afrontar uma grande companhia, tomando como ariete apenas o cérebro, um punhado de chips e um micro doméstico, ganhou entusiastas adeptos no Brasil, e aqui, como nos EUA, os herdeiros de Jobs e Wozniack podem ser divididos entre vitoriosos e fracassados. Ivan Nazarenko, 21, por exemplo, que estará concluindo o curso no Instituto Tecnológico da Aeronáutica (ITA) em 1986, pode ser perfeitamente enquadrado no primeiro caso. Em 1983, quando os micros fabricados no país ainda não tinham caracteres específicos da língua portuguesa, como o cedilha e acentos, Nazarenko, com apenas 18 anos, criou uma placa que "ensinava português" aos computadores nacionais, batizada de Ivanita.

"Foi um invento simples, mas que não tinha precedentes no mundo", explica Nazarenko, que já vendeu através de sua microempresa 1.700 placas, que hoje são comercializadas a 17 ORTNs. "Não deu para ganhar muito dinheiro, mas alimentou certos luxos que tenho. Sem a Ivanita, eu não poderia ter um Passat 1.8, que adoro, e seria obrigado a me contentar com o modelo 1.6", diz. No momento, a empresa de Nazarenko assiste uma queda de vendas da Ivanita —que teve seu ápice de comercialização no começo deste ano—, mas deve voltar à carga no final de 86. "Estou projetando um equipamento de controle para automação industrial, como trabalho final de curso, que deverá ser o próximo produto da empresa".

Christovão Manuel Baptista, 32, diretor da Blump, também começa a vislumbrar o retorno de vinte anos investidos em pesquisa. Criando seu primeiro robô aos doze anos —"com lata de talco no corpo e cabeça de tampa de aerosol"— Baptista começa a engatinhar num mercado milionário: o de braços mecânicos e robôs industriais. Dos Cr$ 50 mil iniciais, abocanhados no programa do Sílvio Santos com sua primeira engenhoca, Baptista pode saltar para Cr$ 100 milhões —o preço estimado de seu modelo mais simples de robô industrial.

Já Fábio Cavalcanti, 24, e Jecel Mattos de Assumpção Jr.,24, criaram há dois anos atrás um computador baseado na filosofia educacional Logo, que prevê o uso de micros em salas de aula, batizado de **Logo Machine**. Na época, quando os micros domésticos ainda não tinham cor, a máquina de Fábio e Jecel, após seis meses de pesquisa, operava com oito cores e sintetizador de sons. Resultado: ninguém quis fabricar a **Logo Machine**. "Nos antecipamos às necessidades do mercado", reconhece Fábio.

Atualmente, Fábio e Jecel têm um projeto bem mais ambicioso: estão projetando um micro de 32 bits voltado à linguagem **Small Talk** —uma poderosa filosofia de programação criada pela Xerox dos EUA, que não requer qualquer conhecimento de computação. "Agora estamos no caminho certo", diz Jecel. "Conceitos de **Small Talk** são usados no Macintosh da Apple, no sistema operacional Unix, mas queremos criar uma máquina puramente **Small Talk**".

Vidal Cavalcante

Fábio (esq.), Jecel e a placa de um microcomputador frustrado

Flávio Ca

A Ivanita, de Ivan Nazarenko, "ensinou português" aos micros brasileiros

Figure 63: news item in São Paulo two years later