**8-Bit CPU Documentation**

*Compiled and Documented by Mukul Malik (21UCS133), Parshva Gang (21ucs144), Mridul Jindal (21ucs132)*

***THE BEGINNINGS***

The very initial stage of building this CPU included the decision of instruction size/word length as the circuits/modules would be based on that. Since it was our first ever project exposing such a task, we **went with the decision** of keeping everything **simple as it could be** while achieving a lot!

Ultimately, to simplify things, we chose 8-bit bus size, 8-bit word length and instruction size. This limits what we can do. But we believe we can still achieve a lot of the functionality that **16-bit instructions** provide if we could tweak its programming and control circuits. At the end, we are able to perform a lot of functions. Whilst, the functions are still limited as it’s based on 8-bit instructions, this design *possibly can be transitioned into a 16-bit CPU with enough tweaking.*

With an 8-bit instruction size, only one option could be thought of as a perfect balance between OPCODE size and OPERAND size **which came to be at 4-bit Opcode and 4 -bit for TWO operands**. The instructions, therefore, will contain at **maximum two operands which will be registers.** Therefore, to address a register, only 2 bits are permitted. Therefore, we could have only four registers in our register file. **So, only Register-Register addressing is permitted with this ISA design.**

**The final bifurcation looks like this:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7: OPC | 6: OPC | 5: OPC | 4: OPC | 3: RA | 2: RA | 1: RB | 0: RB |

(*Instruction Bits)*

7-4 denote OPCODE. 3-2: R(A), 1-0: R(B)

**Details on Register names and addresses are provided in the ISA documentation file.***Instruction bifurcation is flexible for some instruction types. For example, for conditional jumps, operands are not required so all 8 bits form the opcode.*

**The next thing to be done was the Instruction Set Architecture design.**

We had the following instructions in our mind to implement in the circuit: ALU Operations (ADD, NOT, OR, AND, CMP, SHL, SHR), Data Operations (DATA), Movement Operations (LOAD, STORE), Unconditional Jump Operations (JUMP\_REGISTER, JUMP\_ADDRESS) and Conditional Jumps (16 Conditions).

The ALU Operations require **7 instructions with two register addresses.**The Data Operations require **1 instruction with one register address.**  
The Movement Operations require **2 instructions with two register addresses.**The Unconditional Jumps require **1 instruction with zero addresses, 1 instruction with one register address.**The Conditional Jumps require **16 conditions with zero addresses.** *(For jumping, these utilize* ***indirect memory addressing*** *as a constraint on instruction size. Refer to ISA documentation on how these instructions actually work.)*

**This concludes the following requirements:**

9 Instructions with two register addresses,  
2 Instructions with one register addresses,  
17 Instructions with zero register addresses.

To encode the first requirement, we use 4 bits, leaving 4 for register addresses.  
0000 RA, RB  
0001 RA, RB  
0010 RA, RB  
0011 RA, RB  
0100 RA, RB  
0101 RA, RB  
0110 RA, RB  
0111 RA, RB  
1000 RA, RB

We still have 7 opcodes left. Use them to cover the 2nd requirement.

1001 xx RB  
1010 xx RB

We still have 5 opcodes left.

1011  
1100  
1101  
1110  
1111

For 16 conditional jumps, the operand bits **can be used to decide the condition fulfilment.**

Now, to allot opcodes, we chose simplicity (**keeping in mind that decisions made here should be the best ones to keep the CONTROL UNIT simple**).

**Thus, it was decided that all ALU Operations will begin with the bit ‘1’.** WE HAVE 7 OPCODES AVAILABLE that begin with ‘1’ and there are seven ALU operations so this seemed to be the perfect choice.

Thus, if the 7th bit is 1, we classify it as an ALU Operations. The 6th, 5th, 4th bits can determine the type of ALU Operations i.e., the ALU Opcode to be passed to the ALU.

So, the ALU Operations looked like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **ALU OPERATION** | **7th bit** | **6th bit** | **5th bit** |
| ADD | 0 | 0 | 0 |
| SHR (Shift Right) | 0 | 0 | 1 |
| SHL (Shift Left) | 0 | 1 | 0 |
| NOT | 0 | 1 | 1 |
| AND | 1 | 0 | 0 |
| OR | 1 | 0 | 1 |
| CMP | 1 | 1 | 0 |

**The Opcode 1000 was assigned to the ADD Instruction as that makes the ALU Opcode ‘000’ for addition which is by default selected when the ALU is not in operation. This serves a special purpose. This is used to automatically increase instruction addresses by one during the fetch cycle to store in the Program Counter without passing ALU Opcodes as it is by default on addition mode.**

For Movement Instructions, we began from the start: allot 0000 RA RB to LOAD ***and*** 0001 RA RB to STORE.

Then next opcode 0010 xx RB was allotted to DATA instruction.

Then, next **two opcodes (0011, 0100) were allotted to unconditional jumps.** 0011 (JUMP REGISTER), 0100 (JUMP\_ADDRESS).

What’s left are the conditional jumps. To encode these, we used a simple format to make it easy to implement in the Control Unit,

7th – 4th Bits: CONDITIONAL JUMP OPCODE (0101)  
3rd – 0th Bits: Condition in format: C A E Z  
*where C stands for Carry flag, A stands for A larger, E stands for Equal Flag, Z stands for Zero flag.*

Let’s say the 3rd to 0th Bits are 0101. This will imply that the JUMP will be performed **IF the Equal Flag is Set OR the Zero FLAG is Set.** Other flags will not be checked.

0110 xx xx is allotted to **Clear Flags.**

**Therefore, the opcodes 0111 and 1111 are left unused.**

**This completes Instruction Encoding and Architecture.**

**DESIGNING THE CIRCUITS**

There are inbuilt modules for memory (RAM, ROM, etc.) in TkGate. However, we did not choose to use any of them and built **our own modules from scratch (i.e., the gates!).** The reason for this was triggering issues and clocking issues.

The inbuilt modules in TkGate all work on **Edge-Triggered** ClockInputs. **It was difficult to generate a clocking sequence** that would work with the initial idea of the Control Unit. We wanted to use level-triggered, simple modules that would work if the ENABLE was high and not work if ENABLE was low.

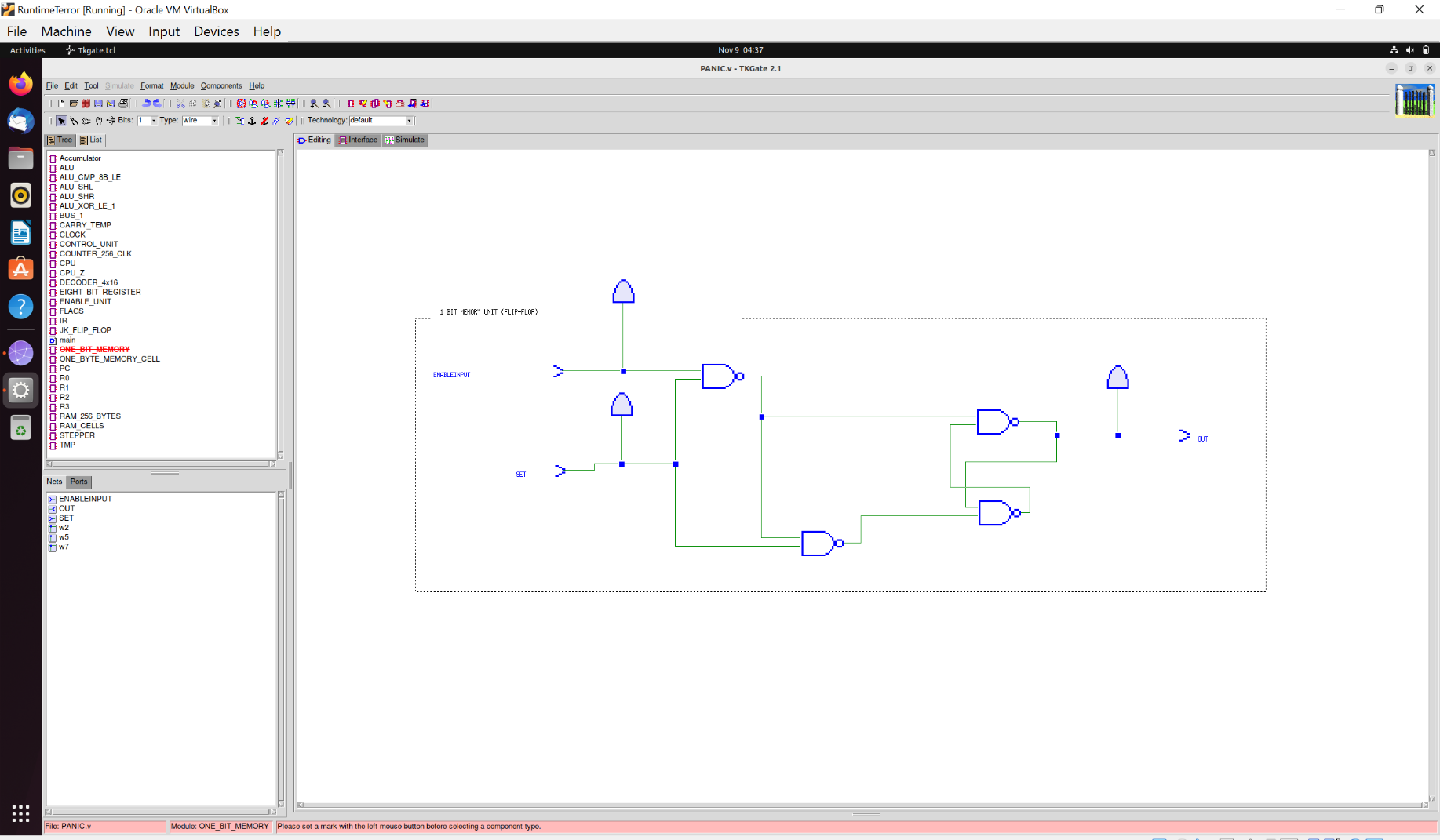
**The Control Unit is a logic unit of combinations of SETS and ENABLES sequences as per our understanding. A clocking sequence is required to operate it. Level-triggered modules paved a way for further development.**

Firstly, we began designing **MEMORY MODULES.**

**MEMORY**

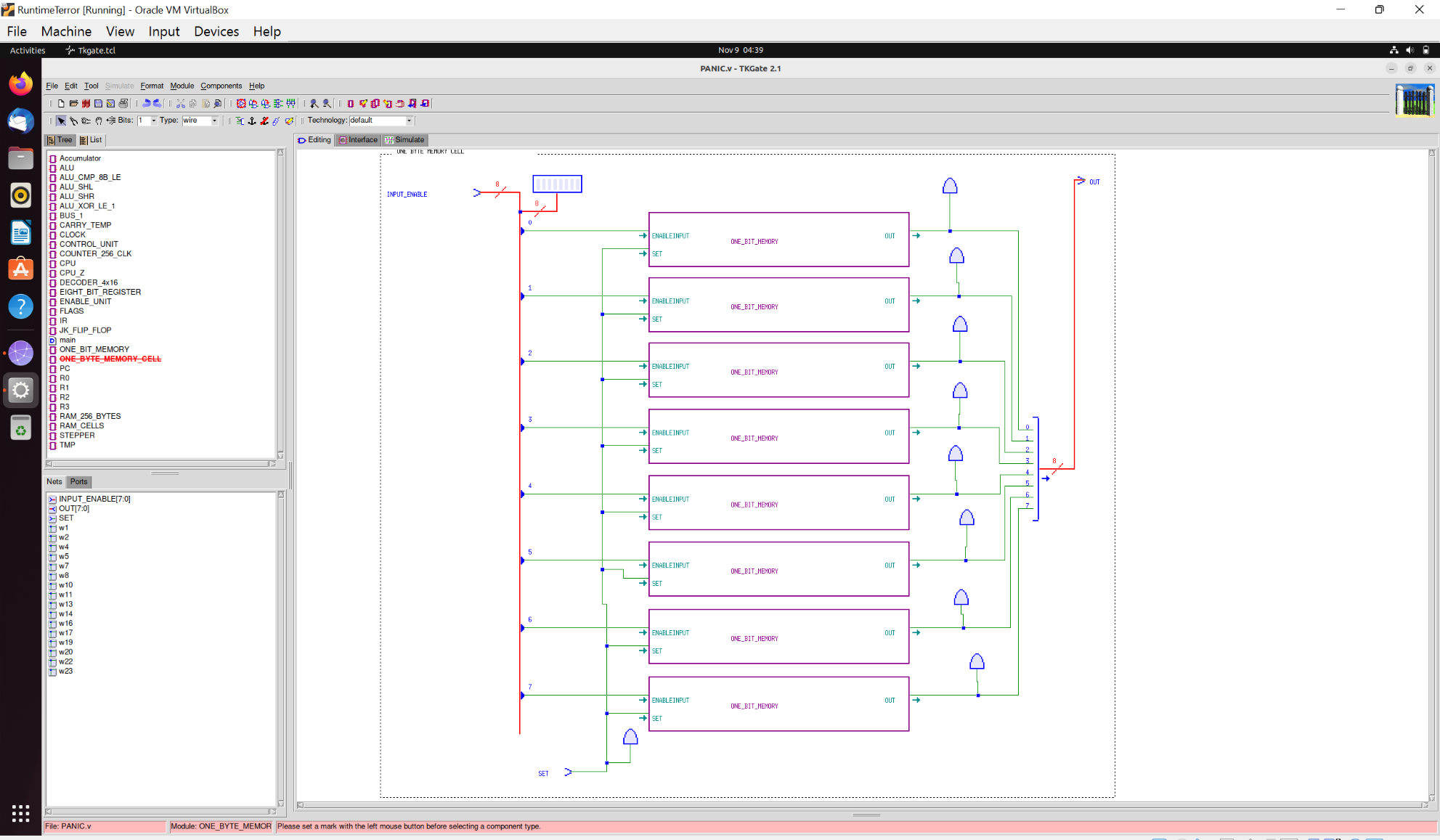
The simplest thing is a 1-bit Memory Unit.

It is a **HIGH-Enabled S-R Latch** with 0-0 and 1-1 conditions eliminated. The input is passed as it is when Set/Enable is ON and nothing changes if Set/Enable is OFF.

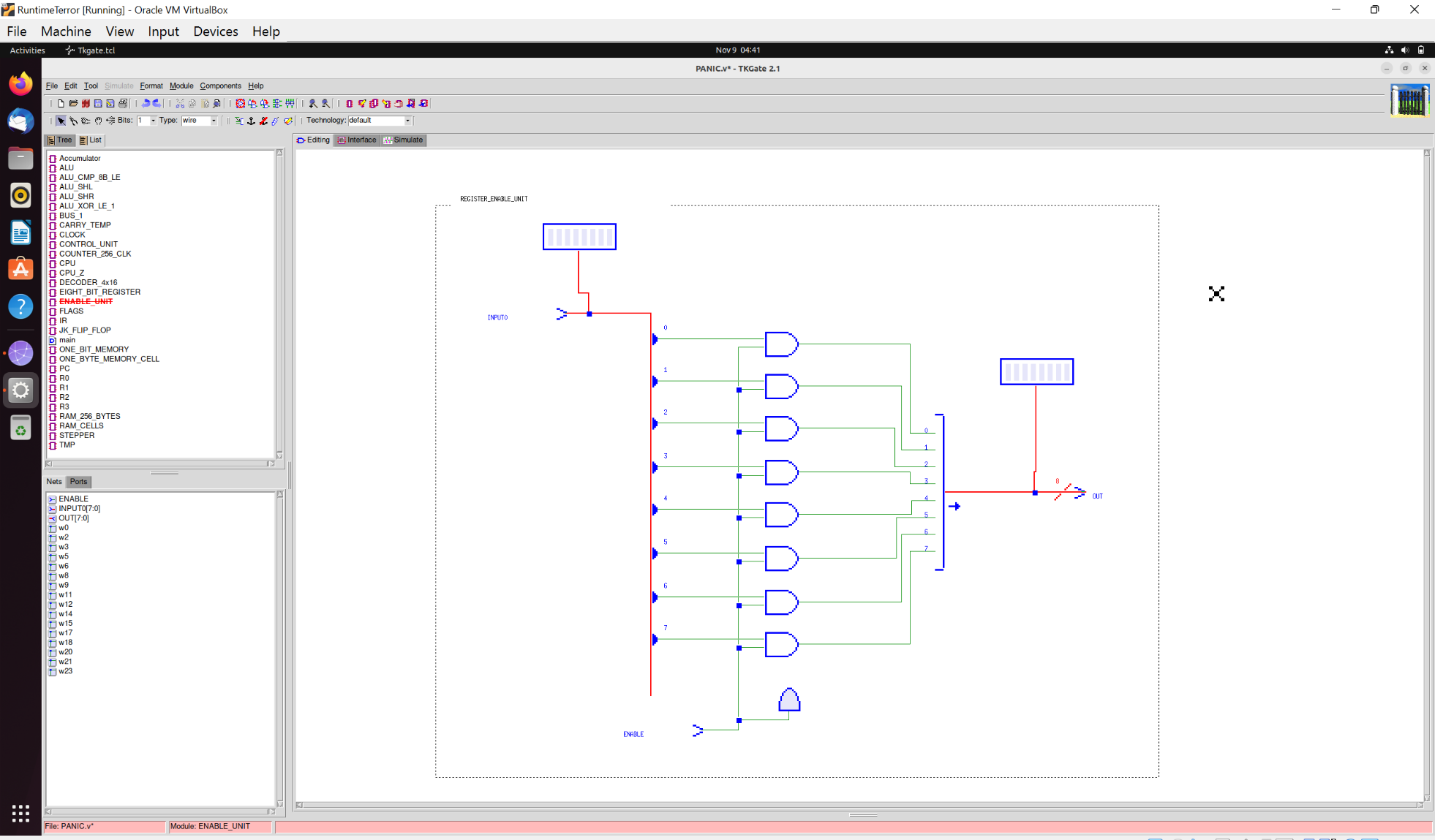


*\*The LEDs were only used to test the circuit and serve no purpose as such in functioning.*

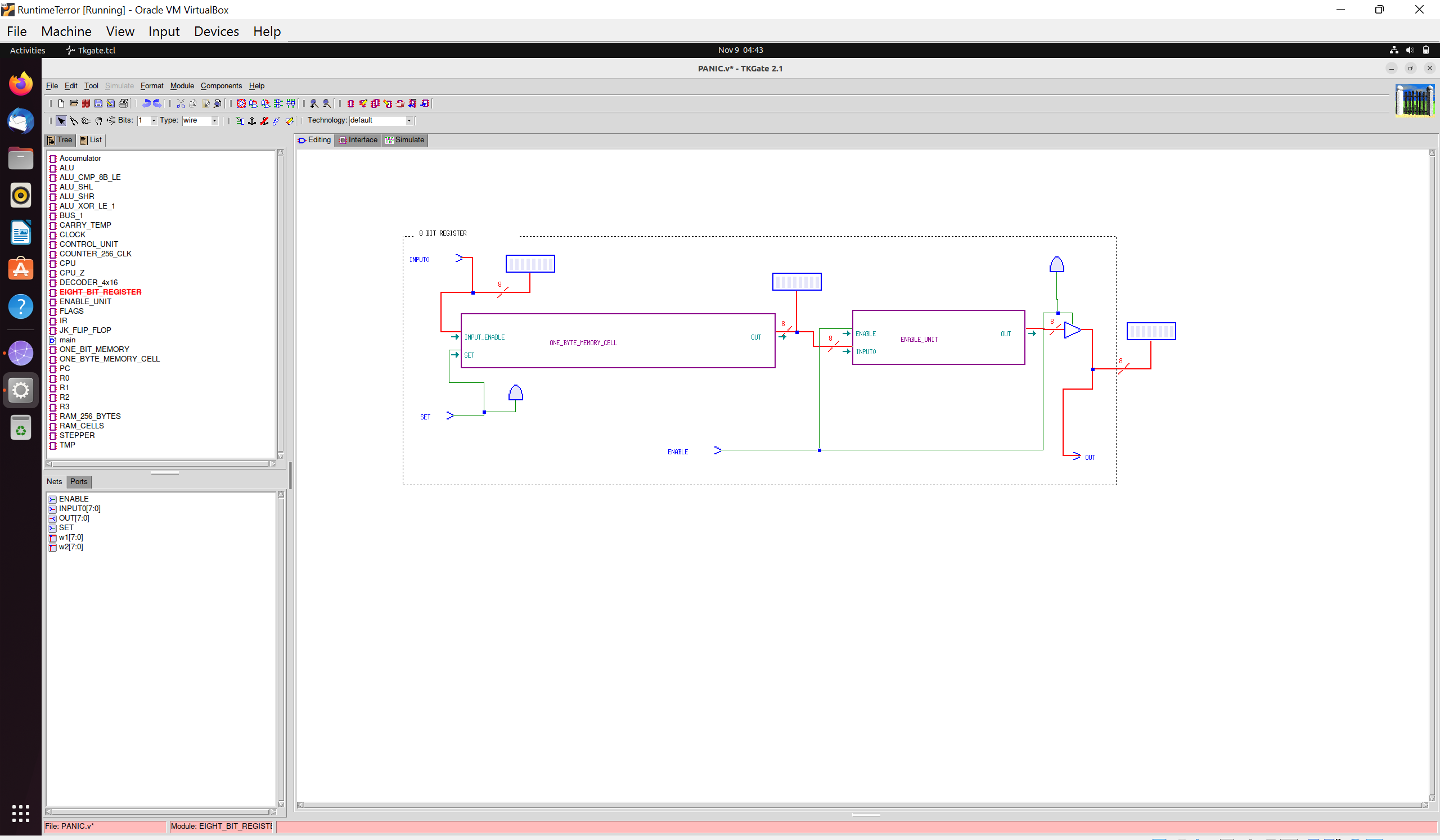
We combined 8 such units to form 1-byte Memory Unit.



What came next was to design **a register** to control the output of such a unit. To do that, an ENABLE unit was added that only lets the output pass if Enable Input is HIGH. The Enable unit simply ANDS the output and ENABLE signal, nothing else.



The Register module looks like this:



This makes the 8-bit Register ready for use!

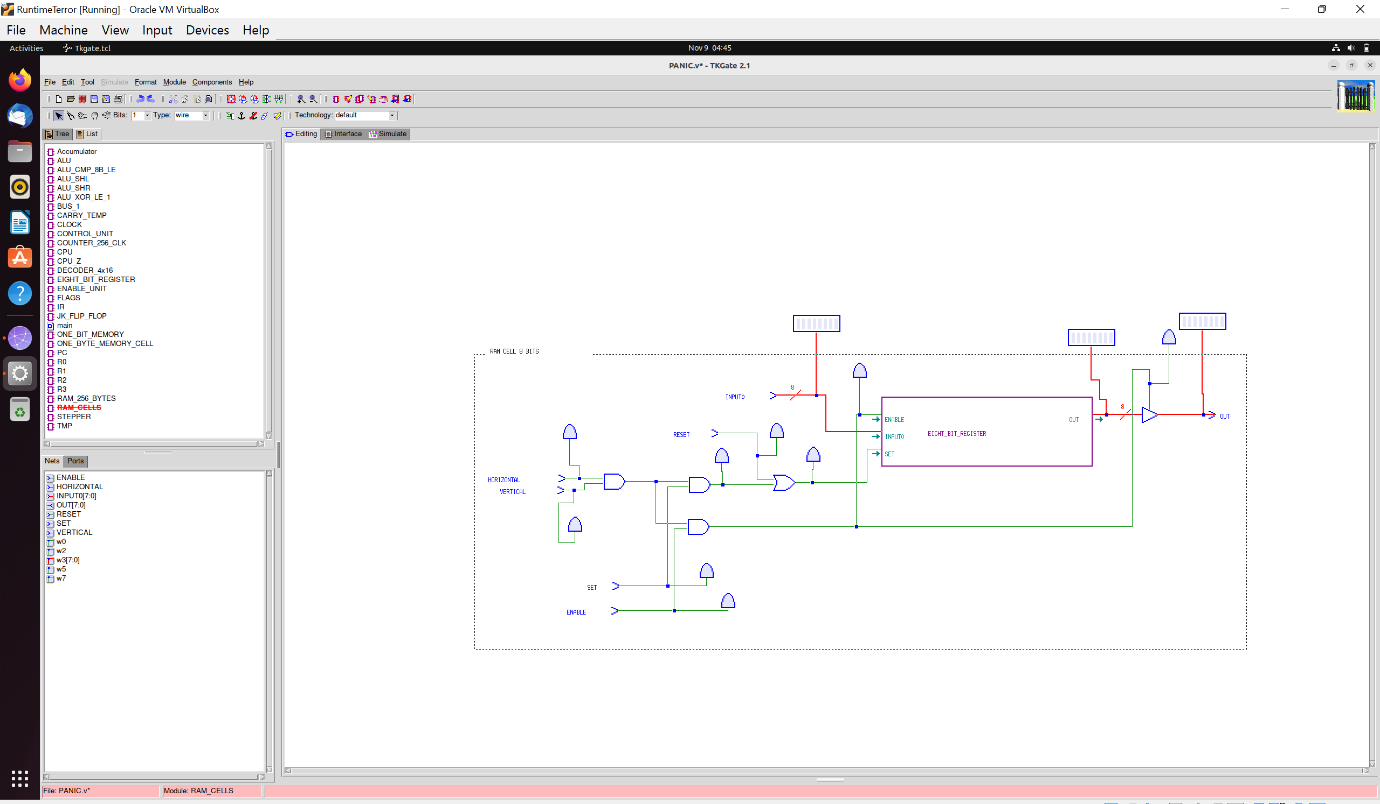
**Next came up the challenge to design a 256-byte RAM. For such, 8-bits would be required to address it. Instead of using a 8x256 Decoder, we chose to use 2 4x16 Decoders.**

We divided RAM’s 1 byte cells into 2 selection criterion:

* **HORIZONTAL ROWS (deciphered by first 4 bits of passed address. Decoded into 16 Rows).**
* **VERTICAL COLUMNS (deciphered by last 4 bits of passed address. Decoded into 16 Columns).**

Thus, each RAM 1 byte unit also required a Horizontal and a Vertical selection pin. The idea was that if both Horizontal and Vertical Signals on a cell were HIGH, then it would be the target unit of the passed address. *A lot of thought went into this but in the end, it worked perfectly.*

MODIFIED REGISTERS (RAM CELLS):



**RAM Designs:**

*(PROPOSED)*

4-7

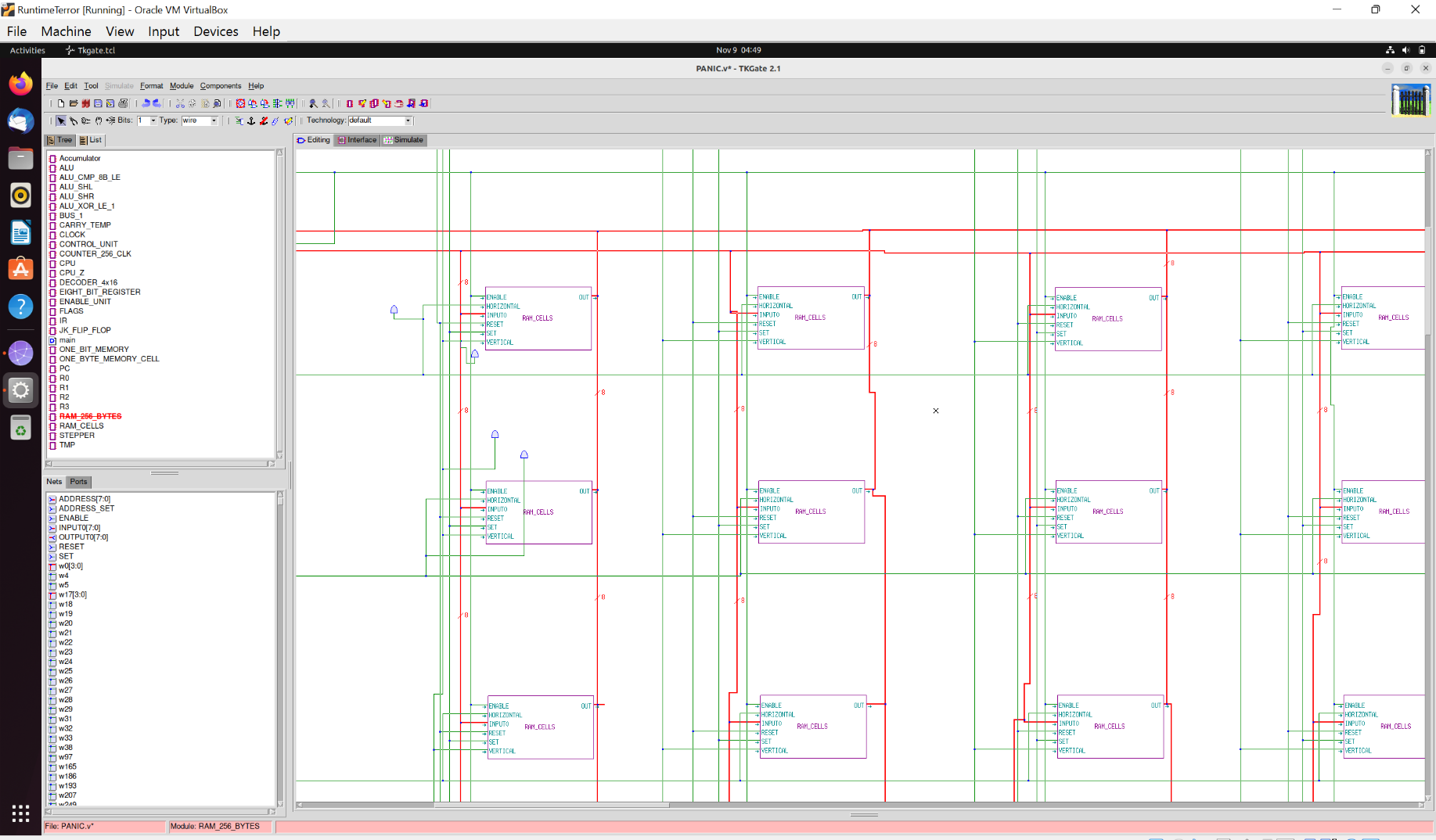
**decoder**

**MAR**

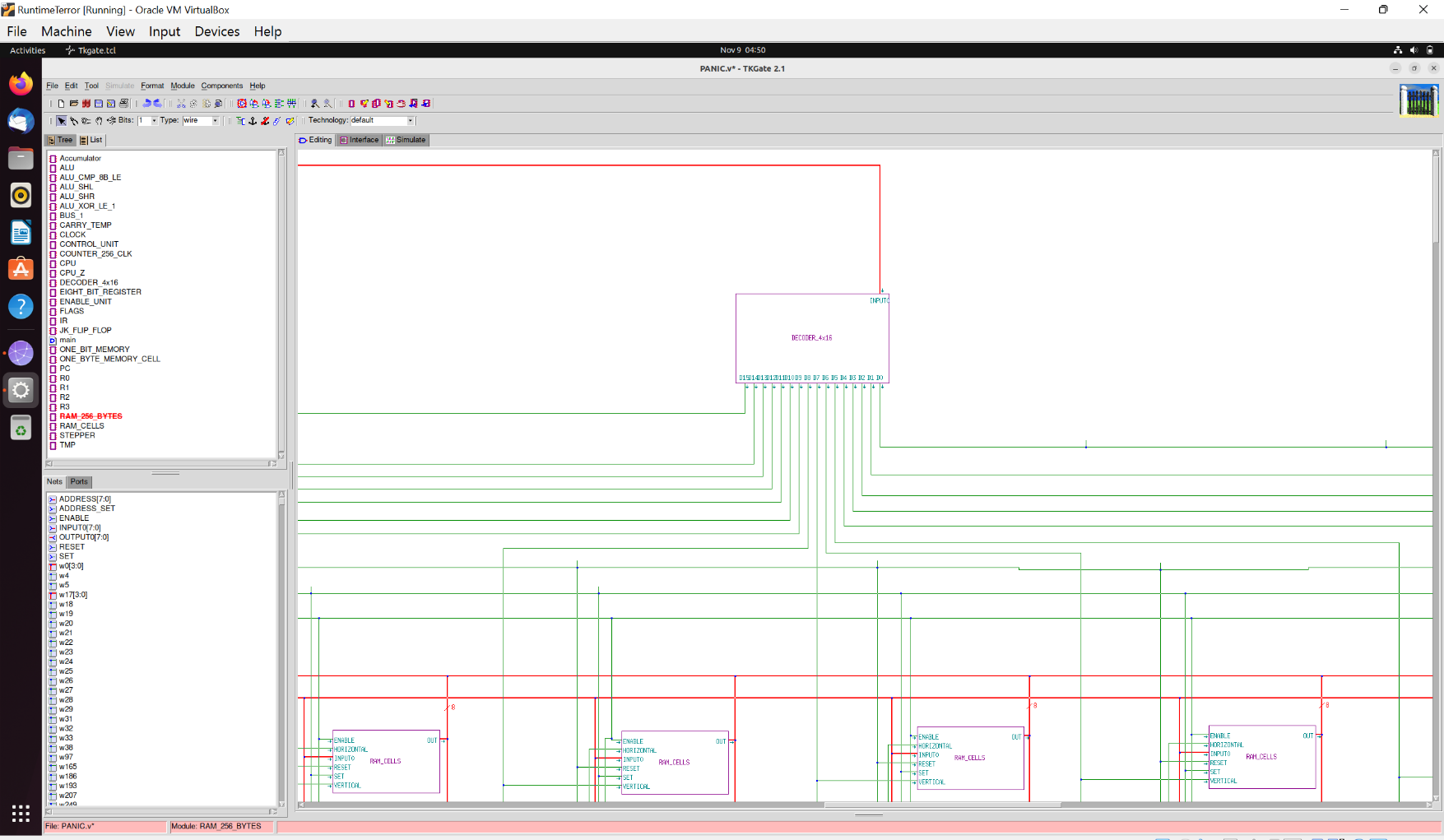
0-3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

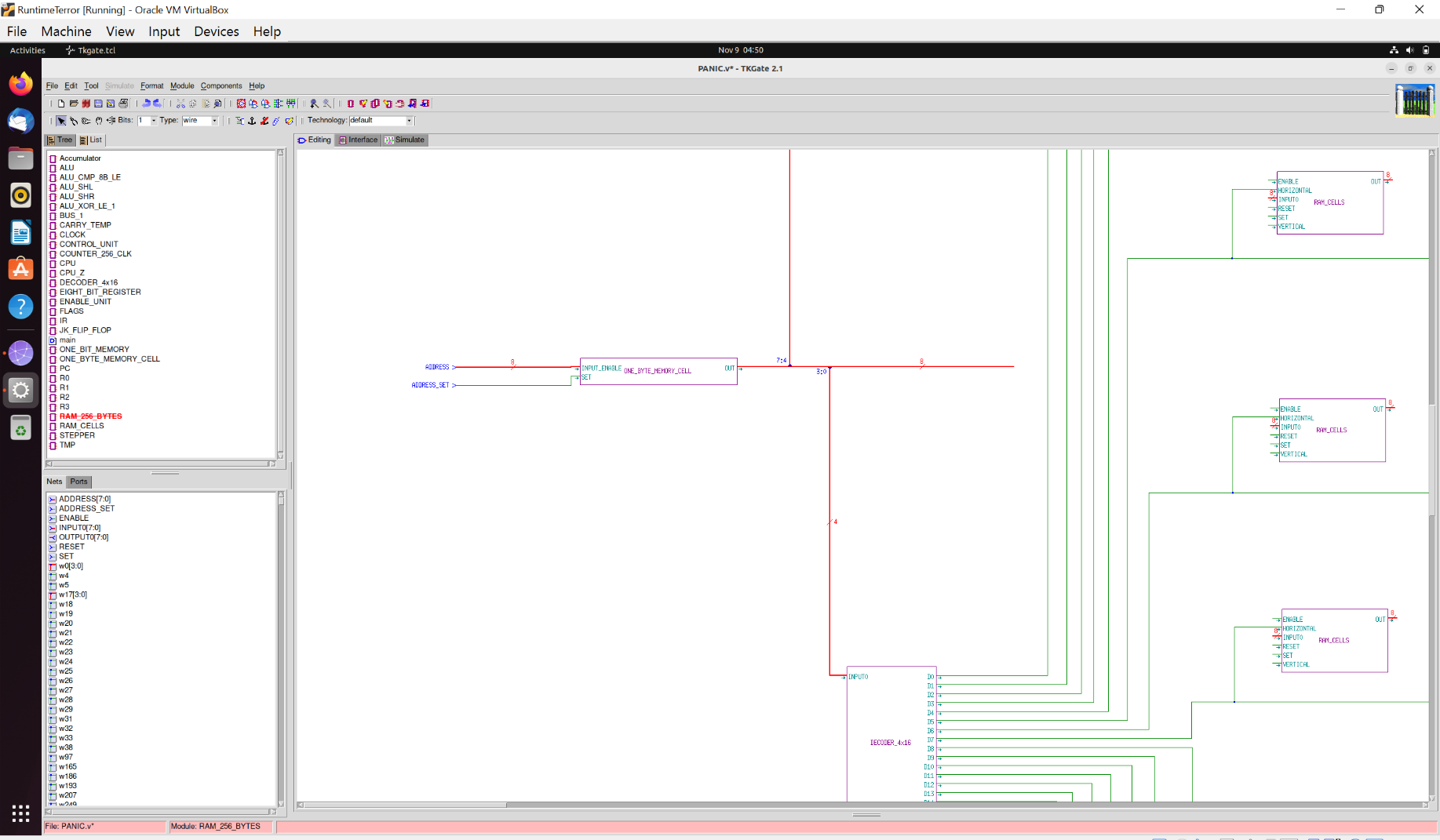
*(ACTUAL)*



256 such cells were connected which is beyond the scope of a single screenshot.

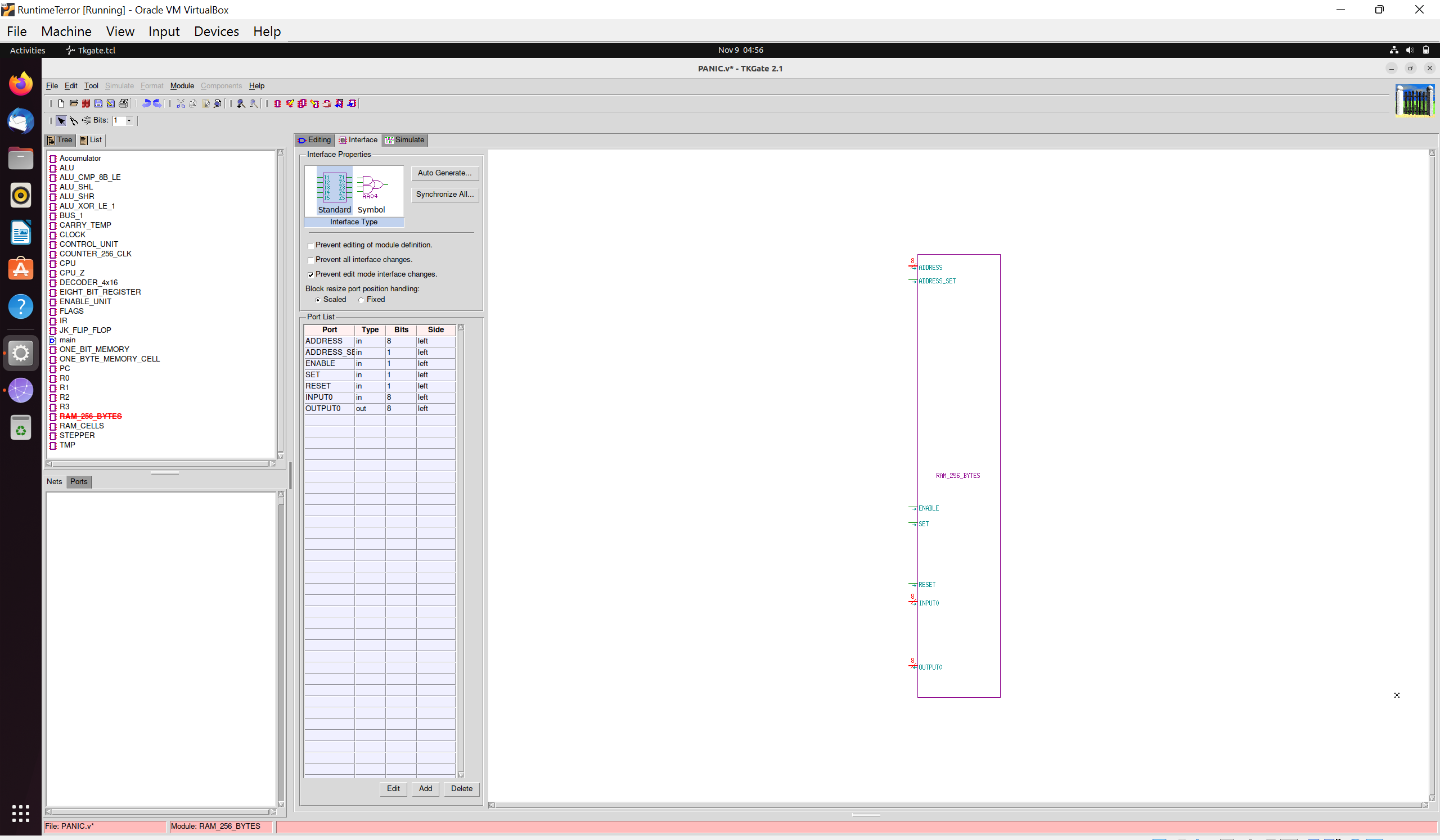


The Vertical Decoder



*(The MAR Unit (Memory Address Register) – Combined with the RAM Module.)*

**That finishes up our talks on the RAM Module.** The final module looks like this (oriented with the Control Unit for pin positioning):

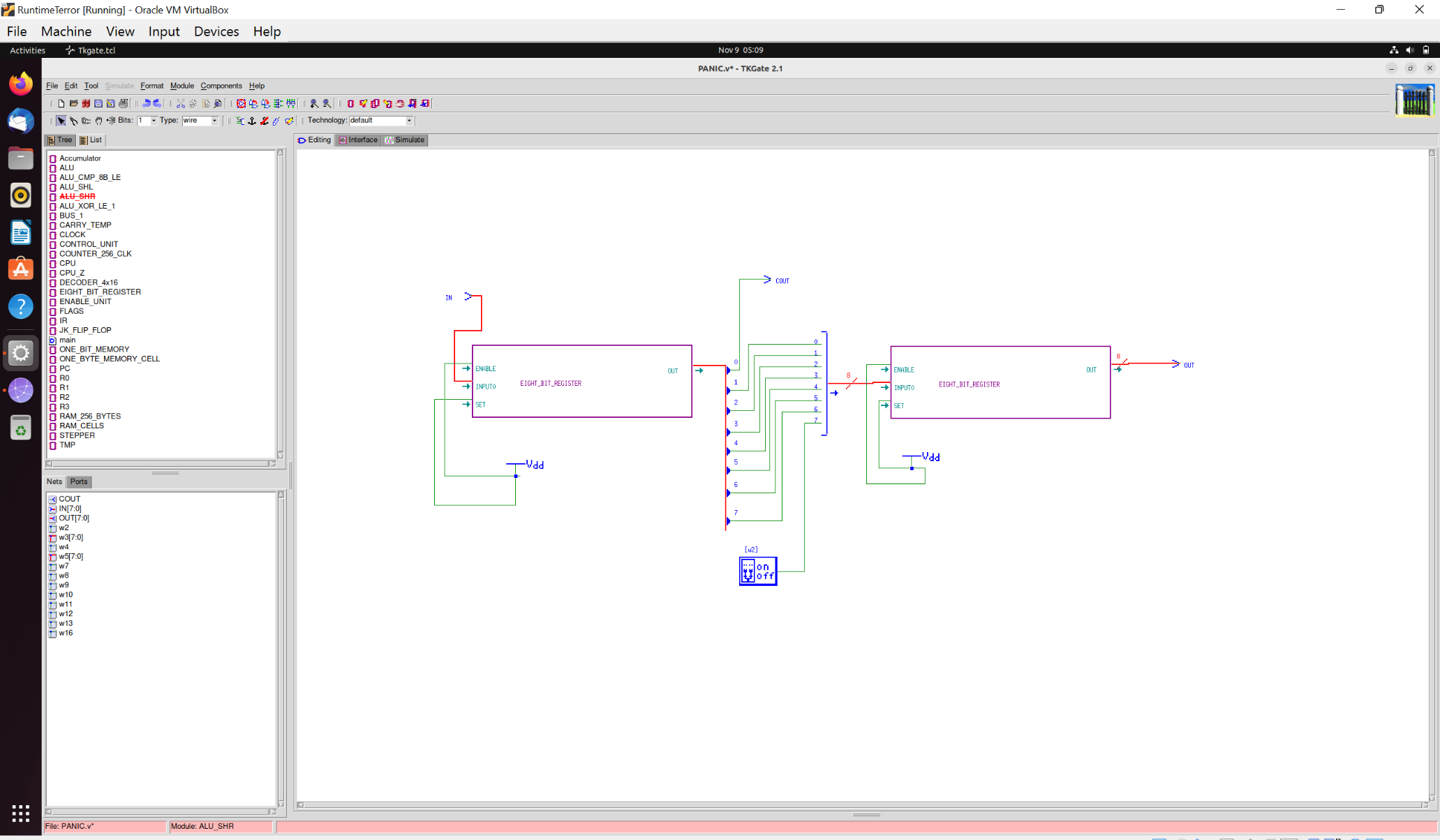


Once, the memory was completed. **Special-Purpose Registers were created:** Accumulator, PC (Program Counter *also referred to as “Instruction Address Register”* at places), IR (Instruction Register), Temporary Registers, Four Main Registers (R0, R1, R2, R3).

**ALU** (Arithmetic Logic Unit)

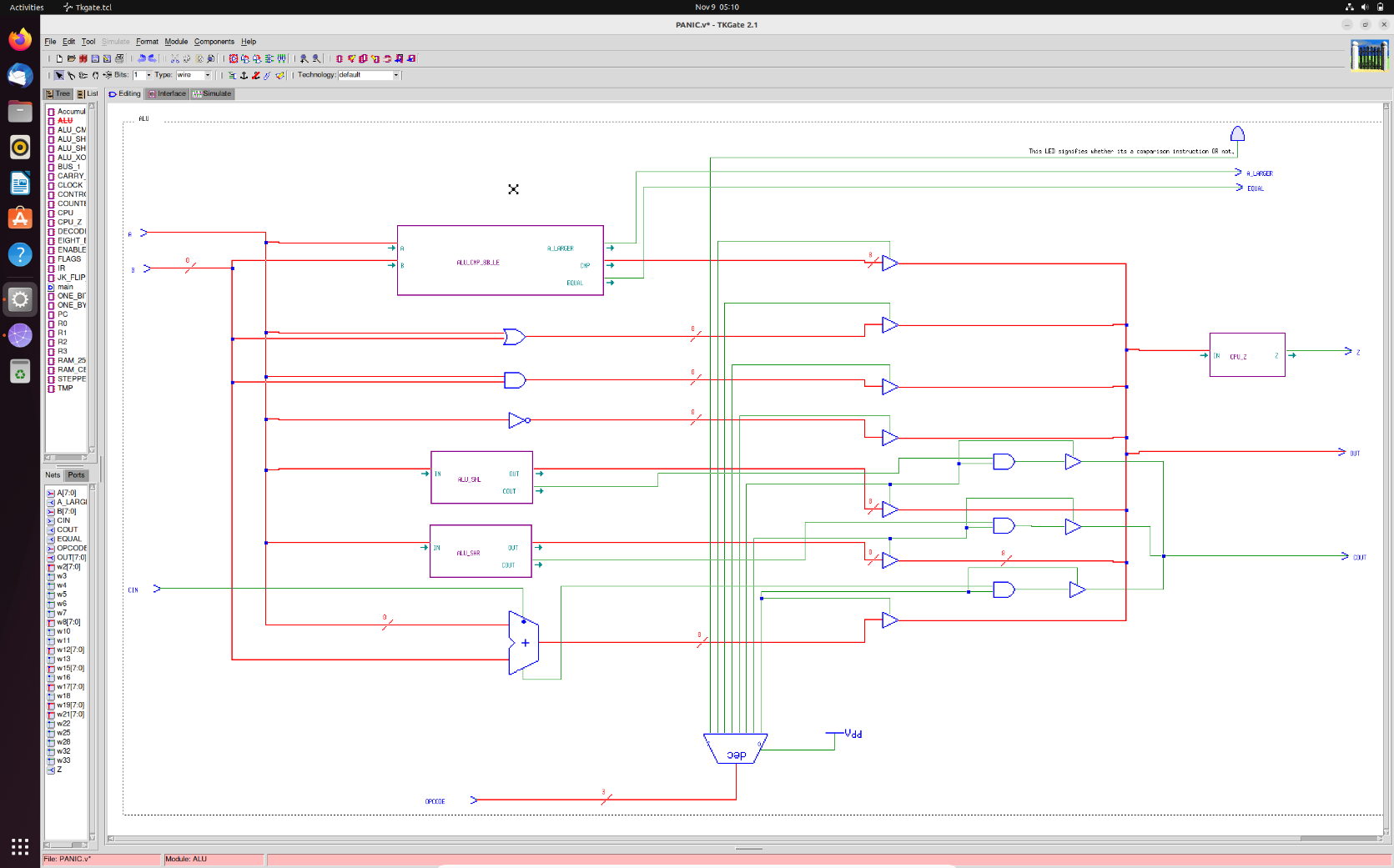
ALU is an integral part of the CPU. We had seven operations to implement. A 3x8 Decoder was used to decipher the ALU Opcode arriving from the Control Unit.

The Comparator, Shift Left and Shift Right functions were implemented by us ourselves. **Tri-State Buffers help in opcode expression.**



*(The Right Shifter)*

**The Complete ALU Design (in all its glory!):**



**CLOCK / STEPPER**

For the circuit to function, we noted a **simple task and observed its performance. That is move data from a register R1 to R2.**

IN

ENABLE

SET

OUT

R1

SET

BUS

IN

ENABLE

OUT

R2

To achieve this, a Clock enable signal would be needed to enable R1’s Enable and at the same time, a Clock set signal would be needed to set R2’s Set pin high. The set is always encased in Enable. **They occur together**.

At the same time, for first sequence, R1’s Enable and R2’s Set are needed. Multiple such sequences can help us determine the **flow of sets and enables to execute instructions.** The solution to that was found to be a **STEPPER.** A 6-step stepper was found to suffice our needs.

**The Ideal Clocking Pulse**

We’re using three clock signals: CLOCK (to control the stepper), CLOCK\_ENABLE (to set Enables), CLOCK\_SET (to set Sets signals). The CLOCK\_ENABLE signal should encompass all of these, as before any further operations, something needs to be enabled for further transportation.

Therefore, between the master Clock\_In, a clock module is required to generate the three **required clock pulses.**

Therefore, the required set of such pulses will be:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MASTER CLOCK SIGNAL** | 1->0 | 0->1 | 1->0 | 0->1 |
| CLK | 0 | 1 | 1 | 0 |
| CLK\_E | 1 | 1 | 1 | 0 |
| CLK\_S | 0 | 1 | 0 | 0 |

Next Step Needed!

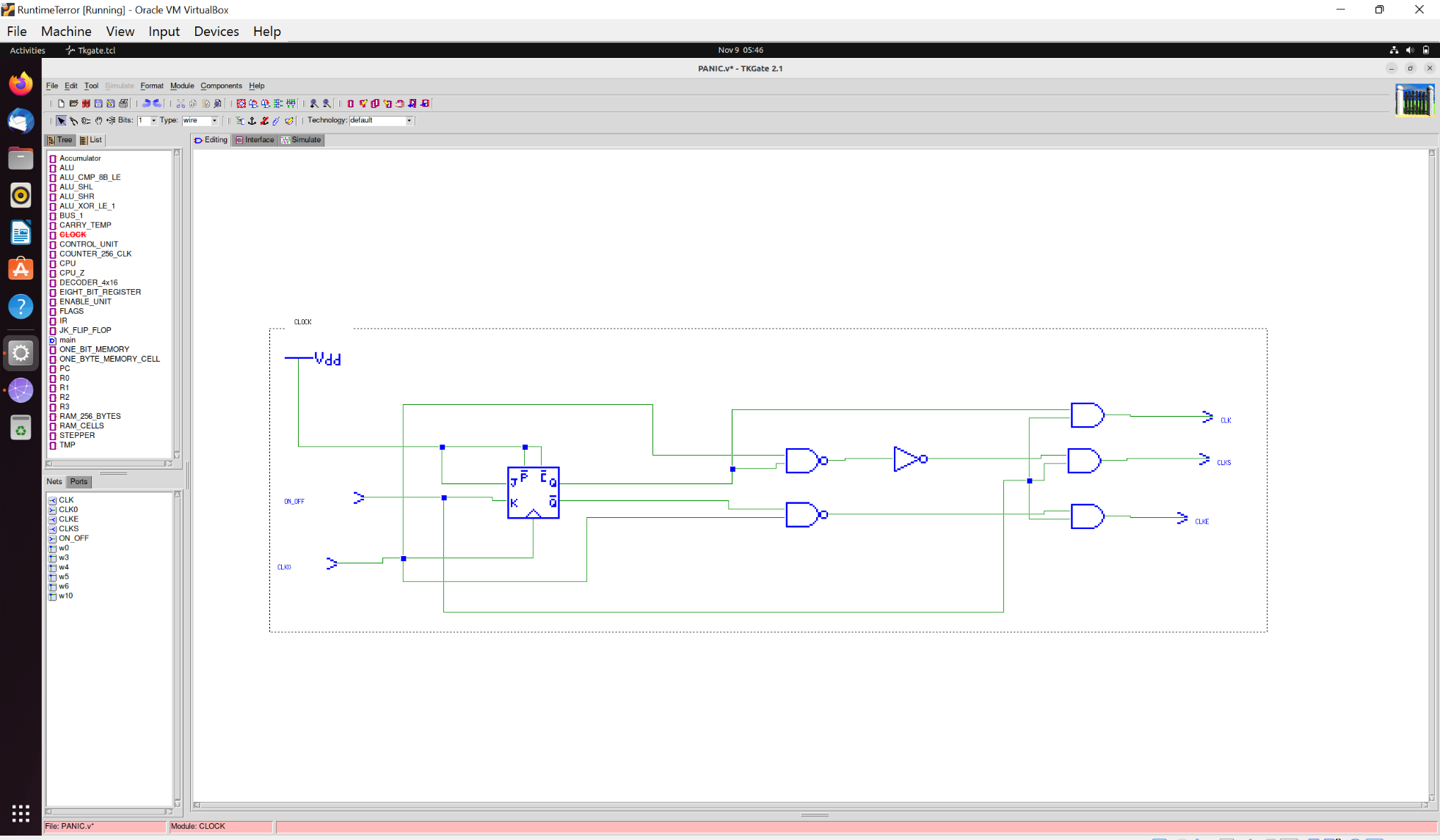
*(Master Clock Input)*

*(CLK to control stepper; triggered by rising edge of master clock input. Implemented using a Edge-triggered JK Flip-Flop with J = K = 1 i.e., the toggling condition).*

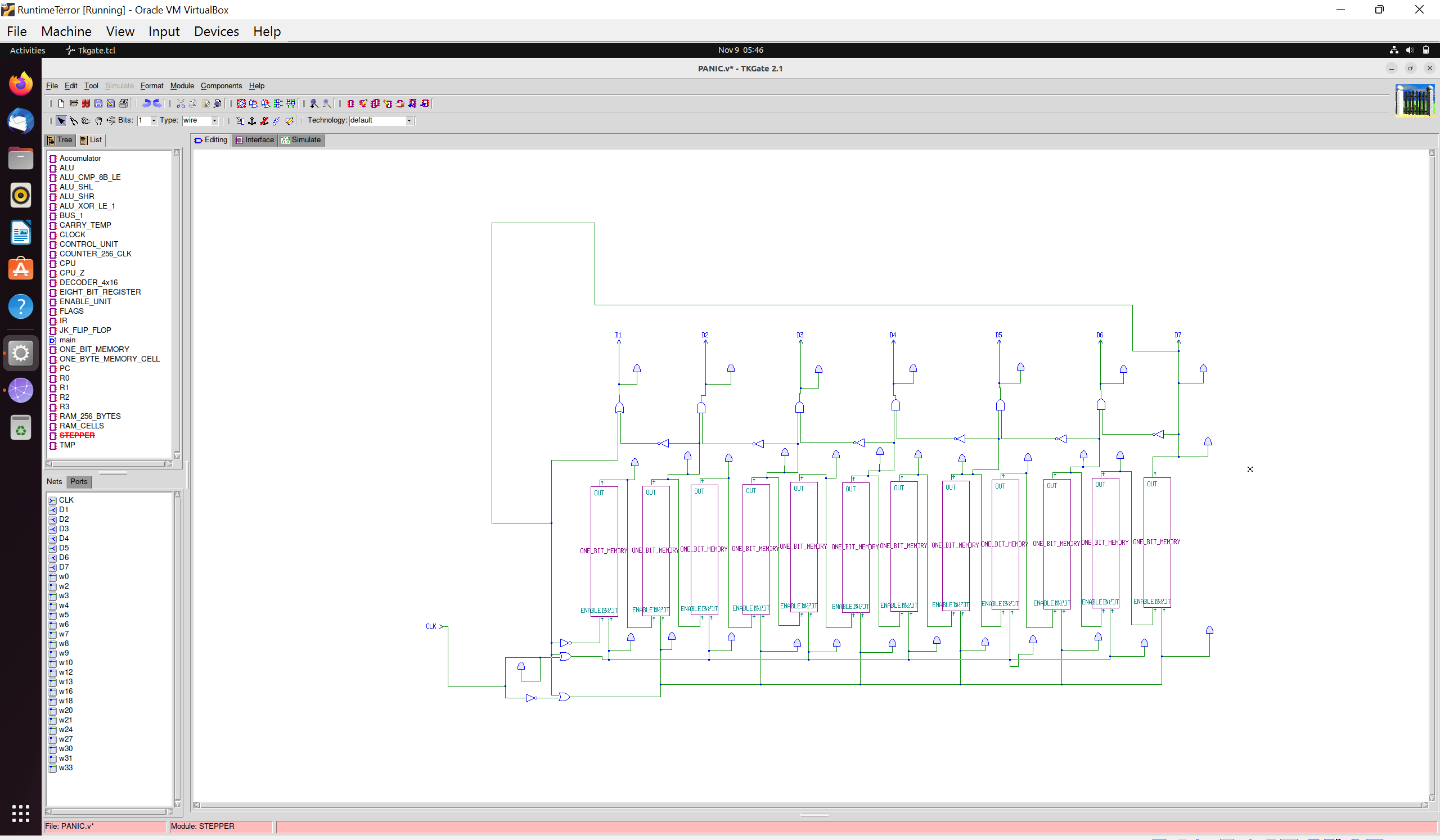
*(CLK\_E which encompasses all other signals. Obtained by ANDing the complement of CLK with the Master Clock Input).*

*(CLK\_S which is encompassed by all other signals. Obtained by ANDing CLK with the Master Clock Input and inverting the output of this AND operation).*

The clock module uses a JK Flip-Flop (edge-triggered; rising edge) set on toggling condition to generate the CLK function from the Master CLK function. The generation of CLK\_E, CLK\_S is mentioned in the graphs above and is evident in the module circuit below.



Now, if we compare it with the signal table on previous page, then it is evident that the stepper must step on **falling edge** of the CLK signal (**1 -> 0**). Such a stepper can be obtained with the following circuit:



**X**

That is all for clocking of the CPU.

**CONTROL UNIT**

It is the beating heart of the CPU. A simple analogy we found that works to visualize the Control Unit is to imagine a little man sitting in the Control Unit. His job is to change the circuit connections as per different instructions (different sequences of Enables, Sets). Now this is not possible so we replace the little man with logic circuits to do the task.

**The first thing to be done was to put the stepper and clock pulses into the CU.** Up next, instruction must be fetched!

**THE FIRST THREE STEPS OF THE CONTROL UNIT FETCH THE INSTRUCTION. THE NEXT THREE STEPS EXECUTE THE INSTRUCTION.** This is the *rule of thumb* we used to design the Control Unit as we found that three steps are enough of our clocking cycles to do job.

**The Fetch Cycle** *(Steps 1 – 3)*

The instruction address will be in Program Counter. Step 1 will first involve enabling of Program Counter Output i.e., instruction address will be output on the BUS and at the same time MAR\_S (RAM’s set address signal) will be set high after that within the Step 1 cycle. The RAM will therefore select the cell containing the instruction.

Step 2 will involve first enabling of RAM to put the instruction onto the bus. Then setting of IR to capture the instruction and pass it to the Control Unit for execution in further steps.

Step 3 involves setting address of next instruction into the PC. That is **current address + 1**. Step 1 also invokes a BUS\_1 signal that passes a strict 1 to ALU as an operand and during step one, current instruction address is already onto the bus, thereby feeding it to the ALU as well. By default, the ALU will be on ‘000’ opcode at that time and perform addition i.e., output the next instruction address. So, during step 1, we will also set the accumulator. During Step 3, we enable the accumulator and set the PC to capture next instruction address.

This **completes the fetch cycle.**

**The Execution Cycle** *(Steps 4 – 6)*

The ALU Operations were implemented first i.e., we check the 7th bit. As you can see in the diagram, RA and RB parts are connected to decoders leading to enable pins, set pins with different setting, enabling conditions (3-4 pins).

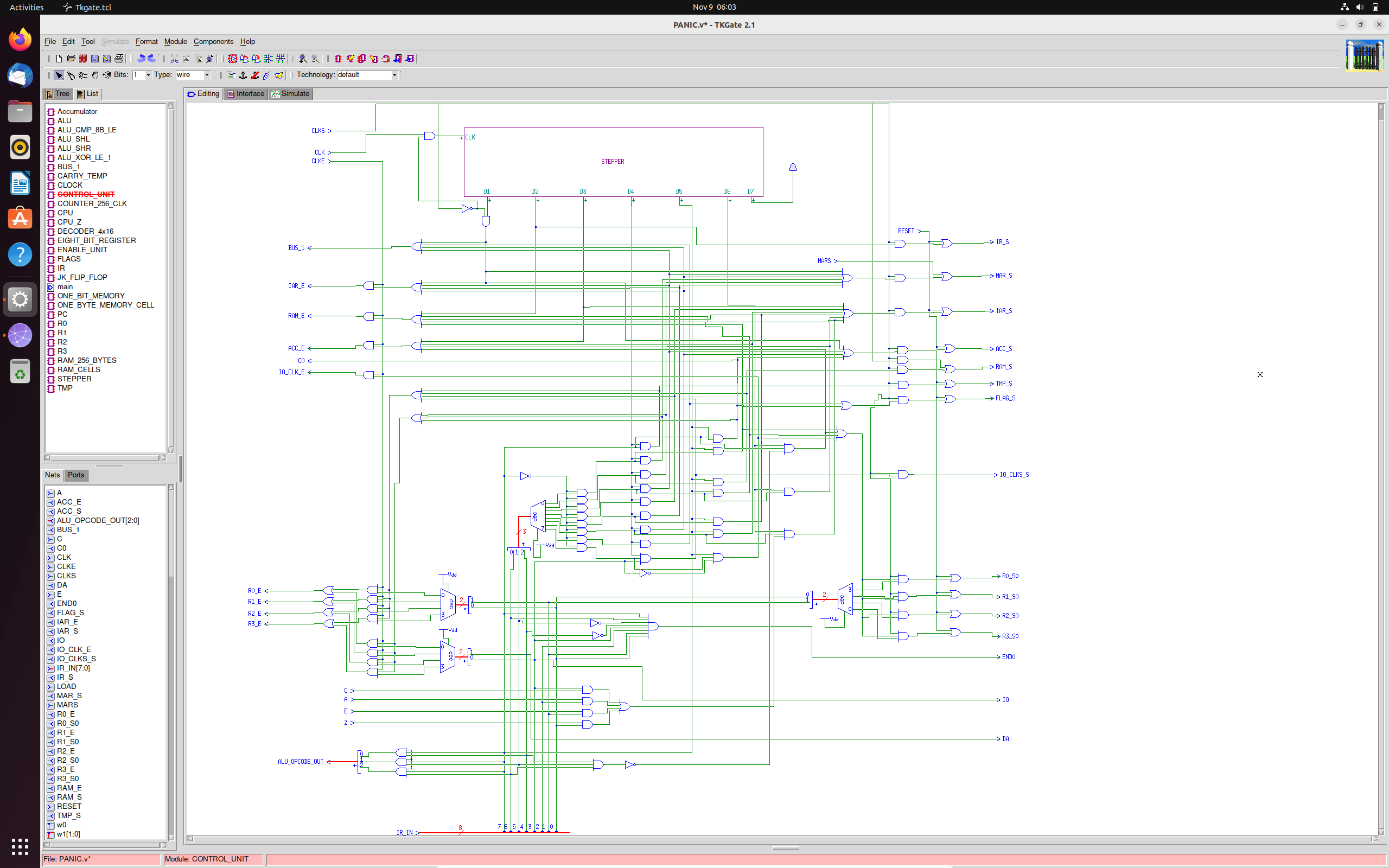
**We’re giving example of one such instruction: ADD 00 01 i.e., 1000 00 01**

Since the invert of 7th bit is 0, no other decoding of opcode will work and hence only ALU-related circuits will work for this instruction.

Step 4: 000 is passed as opcode to ALU for addition. R0\_ENABLE is set high and TMP\_REG\_SET is set high. Since BUS\_1 is zero here, TMP’s data passes to ALU as B operand.

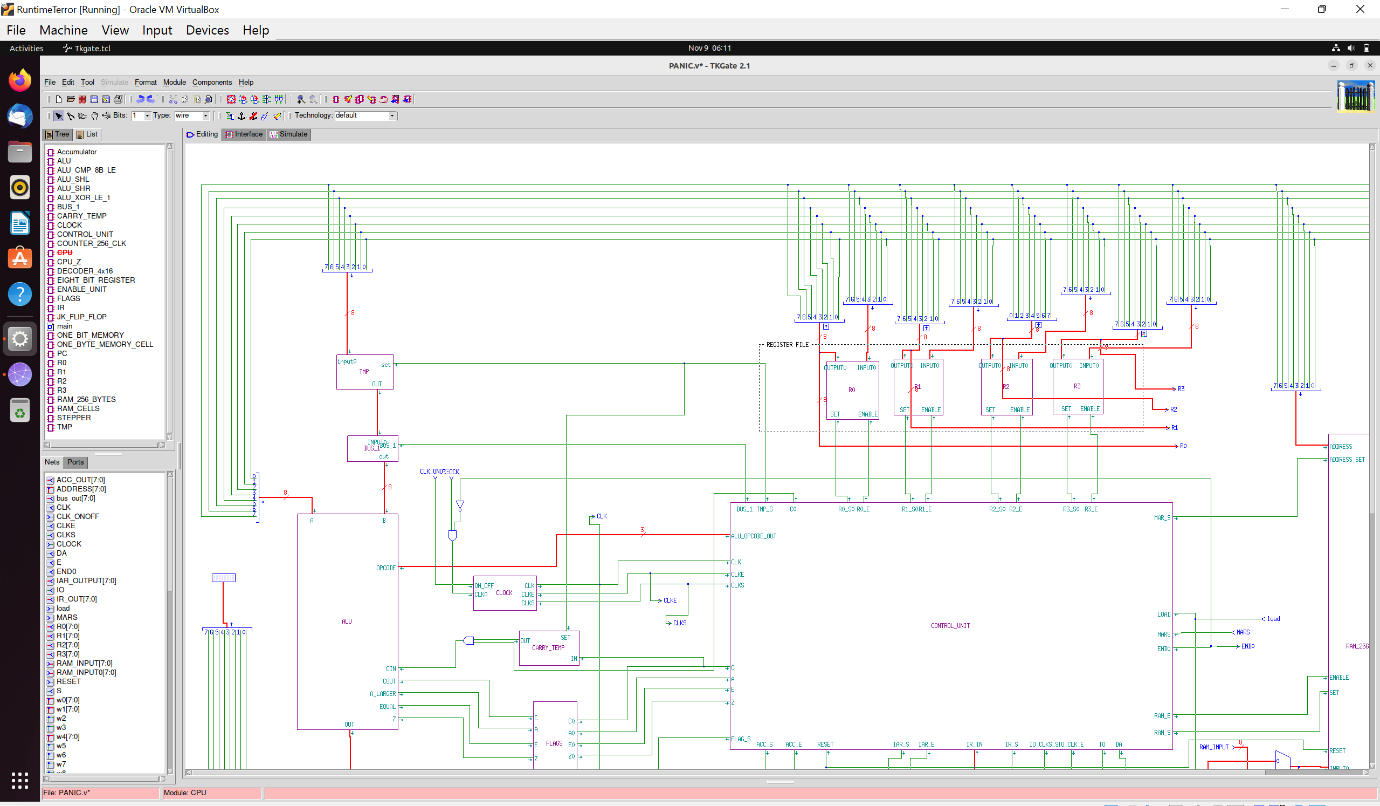
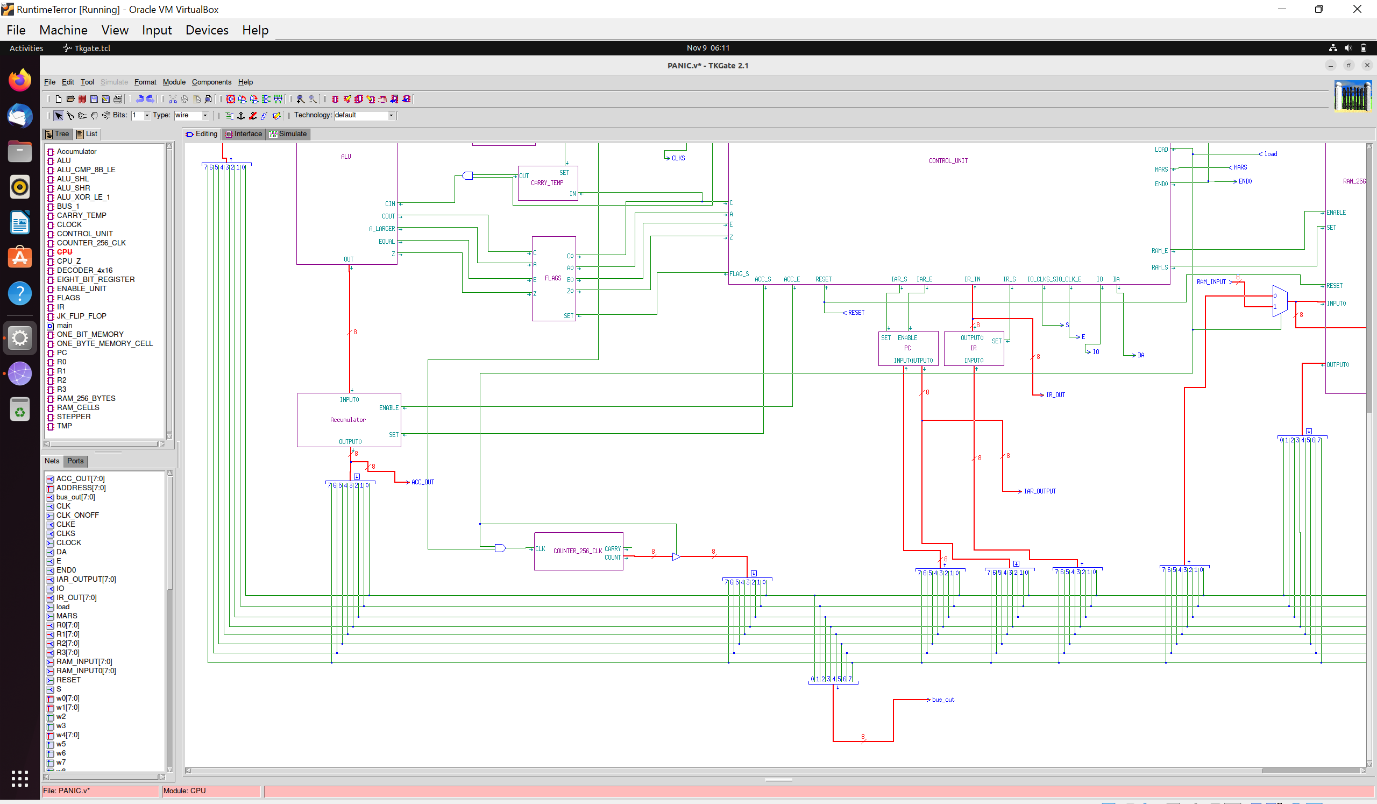
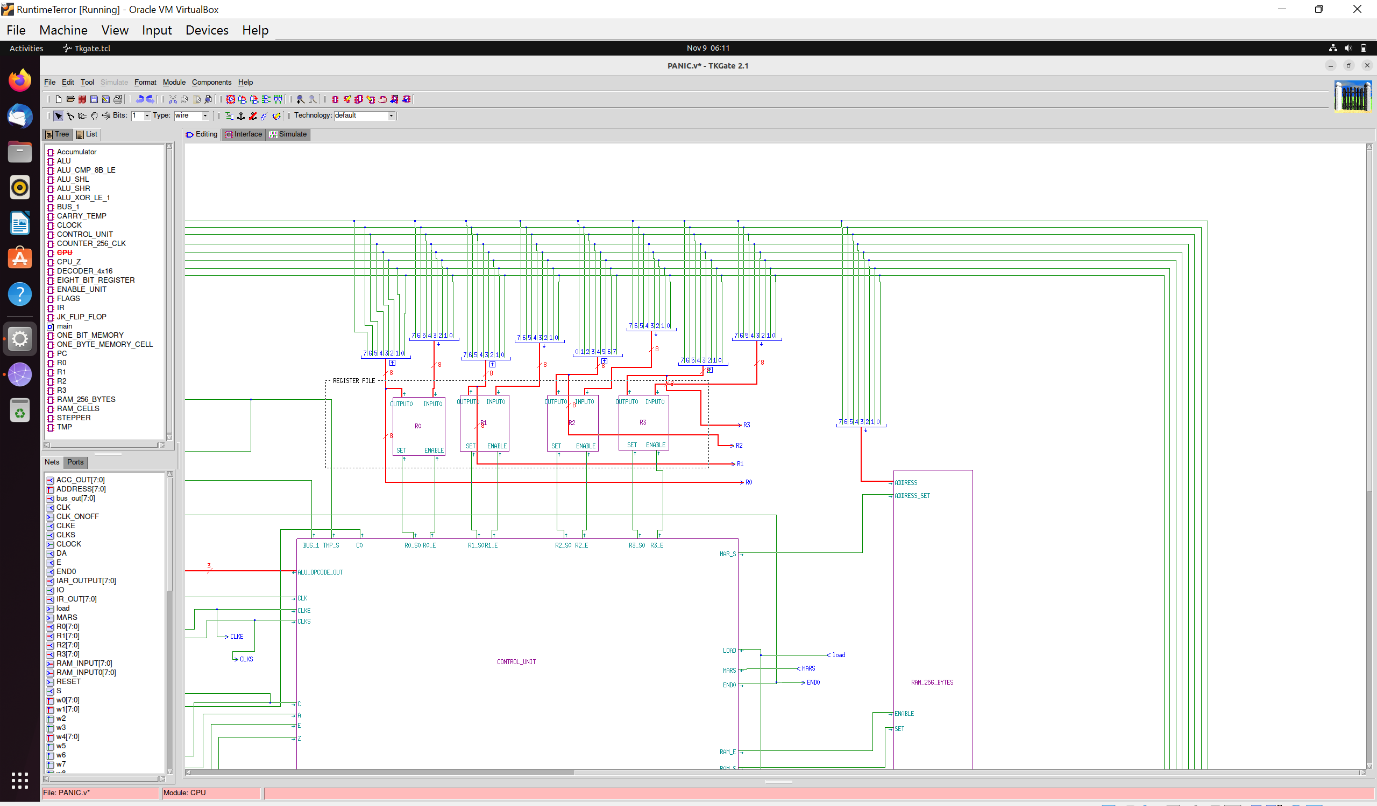
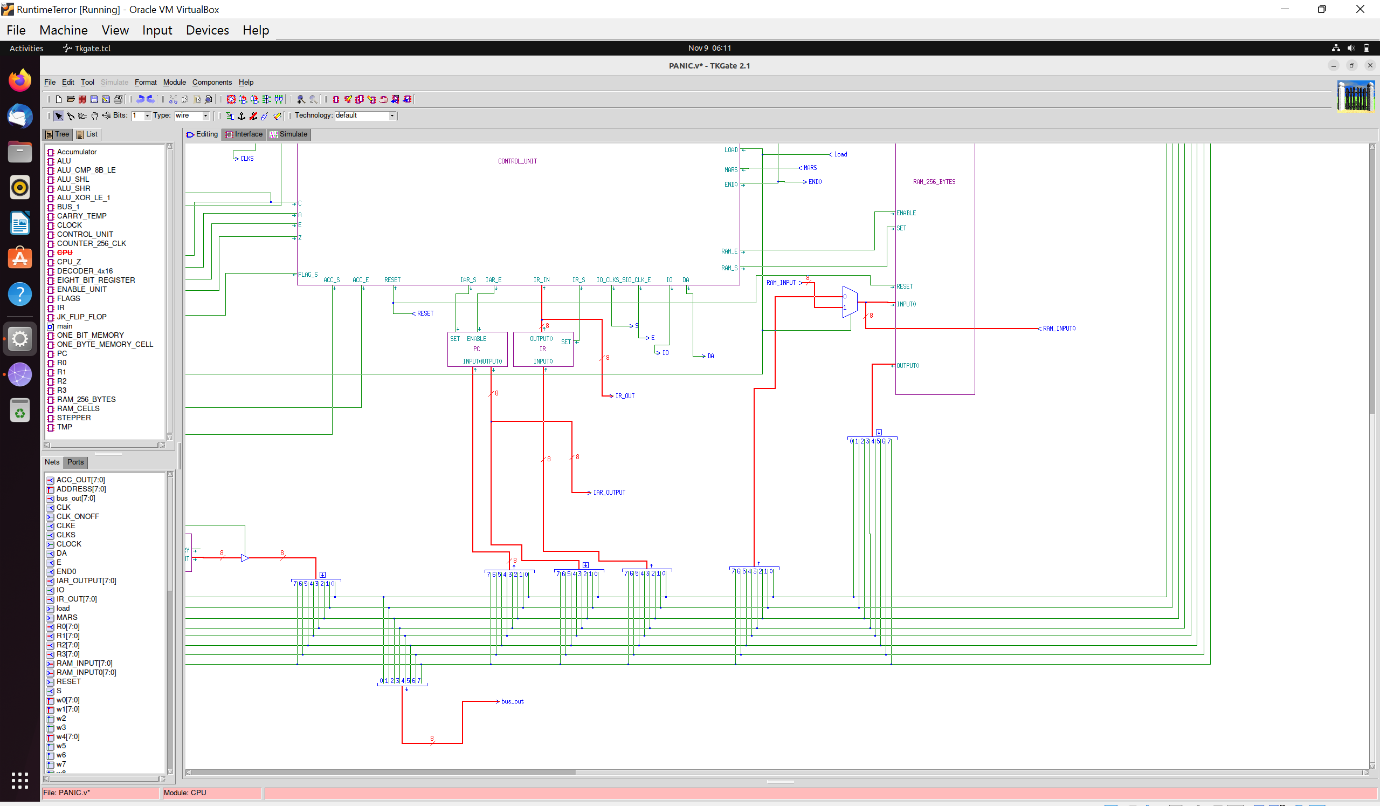
Step 5: R1’s enable is set high (received by ALU as operand A) and Accumulator’s set is set high. The result of R0 + R1 is now in Accumulator.

Step 6: Accumulator is enabled and R1 is on set so output goes to R1 as should be.



*(Complete circuit of Control Unit)*

**HOW IT ALL COMES TOGETHER?**



The bus runs around the circuit connecting all the components. Movement of data is tightly controlled by the Control Unit. Details of each component has been explained above **in depth.**

**Thank you for reading!**