**8-Bit Computer George Sleen**

**Overview**

**Design Goal:**

Create from scratch [[1]](#footnote-1) a Turing complete 8 bit computer.

**Timeline:**

|  |  |
| --- | --- |
| May:  Design decisions | Logic family  Proof of concept logic gates  What are the macro components I need? |
| June:  Logisim macro components, Breadboard macro components |  |
| July:  Full Logisim computer,  Breadboard macro components |  |
| August:  Complete design, Manufacturing |  |

**Logic Families**

|  |  |  |
| --- | --- | --- |
|  | **Pros** | **Cons** |
| **Resistor Transistor Logic (RTL)** | Incredibly simple  BJT transistors  Inexpensive components  Static electricity resistant | Power inefficient  Lots of components needed  Slow switching speed  Susceptible to noise  Bad fan out |
| **Diode Transistor Logic (DTL)** | BJT transistors  Inexpensive components  Static electricity resistant  Handles noise well | Power inefficient  More complicated than RTL  Lots of components needed  Slow switching speed |
| **Transistor Transistor Logic (TTL)** | Power efficient  BJT transistors  Inexpensive components  Static electricity resistant | Ideally uses multiple emitter transistors  More complicated than RTL  Lots of components needed |
| **Complementary Metal Oxide Semiconductor Logic (CMOS)** | Incredibly power efficient  Very fast switching speed | Static electricity susceptible  Expensive components  Complicated logic gates |

There are of course more logic families, these are simply the most common and have differences that will be relevant to the project. Due to the low clock speed of the final computer, I don’t have to worry too much about transistor saturation[[2]](#footnote-2).

After screening[[3]](#footnote-3) I cut DTL for the more complicated gate design and CMOS for the price and complexity again. Comparing TTL and RTL, RTL’s simplicity won out for me. I do not have access to an oscilloscope currently and I want to minimize the amount of high-speed troubleshooting I need to do.

**Digital Logic**

**Logic Gates**

The fundamental building blocks of digital logic, these simply take in one or more input and give back one output.

Logic gates are commonly shown as block diagrams, which are summarized below:

|  |  |  |
| --- | --- | --- |
|  | Block Diagram | Truth Table |
| **BUFFER** |  | |  |  | | --- | --- | | IN | OUT | | 0 | 0 | | 1 | 1 | |
| **NOT** |  | |  |  | | --- | --- | | IN | OUT | | 0 | 1 | | 1 | 0 | |
| **AND** |  | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 0 | | 1 | 0 | 0 | | 0 | 1 | 0 | | 1 | 1 | 1 | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **NAND** | A black and red line with blue lines  Description automatically generated | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 1 | | 1 | 0 | 1 | | 0 | 1 | 1 | | 1 | 1 | 0 | |
| **OR** | A black and red line with blue lines  Description automatically generated | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 0 | | 1 | 0 | 1 | | 0 | 1 | 1 | | 1 | 1 | 1 | |
| **NOR** | A black and blue line with a pointy triangle  Description automatically generated with medium confidence | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 1 | | 1 | 0 | 0 | | 0 | 1 | 0 | | 1 | 1 | 0 | |
| **XOR** | A black and red line with blue lines  Description automatically generated | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 0 | | 1 | 0 | 1 | | 0 | 1 | 1 | | 1 | 1 | 0 | |
| **XNOR** | A black and blue line drawing on a white surface  Description automatically generated | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 1 | | 1 | 0 | 0 | | 0 | 1 | 0 | | 1 | 1 | 1 | |

**Boolean Algebra**

Boolean algebra allows us to compute what will happen in a more complex circuit. All the logic gates above can be represented through 4 mathematical operations:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **NOT** |  | Inverts the value of A | |  |  | | --- | --- | | IN | OUT | | 0 | 1 | | 1 | 0 | |
| **OR** |  | Returns 1 if either A or B is 1 | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 0 | | 1 | 0 | 1 | | 0 | 1 | 1 | | 1 | 1 | 1 | |
| **AND** |  | Returns 1 if both A and B are 1 | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 0 | | 1 | 0 | 0 | | 0 | 1 | 0 | | 1 | 1 | 1 | |
| **XOR** | ⊕ B | Returns 1 if only A or B is 1 | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 0 | | 1 | 0 | 1 | | 0 | 1 | 1 | | 1 | 1 | 0 | |

We can combine these operations to get any of the logic gates above.

**RTL Logic Gates**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **NOT** | **A diagram of a circuit  Description automatically generated** | |  |  | | --- | --- | | IN | OUT | | 0 | 1 | | 1 | 0 | |
| **NAND** | A diagram of a circuit  Description automatically generated | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 1 | | 1 | 0 | 1 | | 0 | 1 | 1 | | 1 | 1 | 0 | |
| **NOR** | **A diagram of a circuit  Description automatically generated** | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 1 | | 1 | 0 | 0 | | 0 | 1 | 0 | | 1 | 1 | 0 | |
| **XOR** | **A diagram of a circuit  Description automatically generated**  (Notice that this is just four NAND gates) | |  |  |  | | --- | --- | --- | | A | B | OUT | | 0 | 0 | 0 | | 1 | 0 | 1 | | 0 | 1 | 1 | | 1 | 1 | 0 | |

<https://www.youtube.com/watch?v=nB6724G3b3E>

May 25th, 2024

**Physical Logic Gates**

After creating the schematics for physical RTL logic gates last week, I built all of them up on breadboards:

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| **NOT** | A white circuit board with a red light  Description automatically generated | A close-up of a circuit board  Description automatically generated |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **00** | **01** | **10** | **11** |
| **NAND** | **A white circuit board with a red light  Description automatically generated** | **A hand holding a screwdriver and a circuit board  Description automatically generated** | **A person holding a screwdriver on a white circuit board  Description automatically generated** | **A hand holding a tool to a white circuit board  Description automatically generated** |
| **NOR** | **A white circuit board with wires and wires  Description automatically generated** | **A person using a tool to connect the circuit board  Description automatically generated** | **A person using a tool to connect the circuit board  Description automatically generated** | **A hand holding a screwdriver on a white circuit board  Description automatically generated** |
| **XOR** | **A circuit board with wires and wires  Description automatically generated** | **A hand holding a tool to a circuit board  Description automatically generated** | **A circuit board with wires and wires  Description automatically generated** | **A circuit board with wires and wires  Description automatically generated** |

**74LS Series Chips**

Making a computer out of entirely transistors is… completely pointless. As I’ve built up logic gates from transistors, I will now be using the 74LS series for all my logic. It’s TTL with Schottky diodes for ultra fast switching speed.

**Half Complete Clock Module**

Ben eater has an amazing video on the 555 timer (among other things) which I’m using for my clock module. It has both an automatic and manual step mode with variable timings and a debounced button.

Here is what I could make this week with the components I had on hand:

A hand holding a green circuit board

Description automatically generatedA hand holding a green circuit board

Description automatically generatedA hand holding a green circuit board

Description automatically generated

Video to come next week.

**Half Adder**

One of the big parts I’ll need for the final computer is an arithmetic and logic unit (ALU). The final design will be able to add, subtract, AND, OR, NAND, NOR, and XOR the numbers in registers A and B, then output them to register C.

The “core” of the ALU is the adder, and all that it does is add two 8-bit numbers together.

Next week will include what a full adder entails, and the algorithm for why a half adder doesn’t work.

A circuit board with wires and wires

Description automatically generatedA hand holding a screwdriver with wires

Description automatically generatedA hand holding a screwdriver with wires on a circuit board

Description automatically generatedA hand holding a circuit board with wires

Description automatically generated

**Almost Complete Simulated Computer**

Before committing to buying chips and manufactured PCB’s I went through exercises and built a Turing complete computer[[4]](#footnote-4) in a digital logic simulator

A diagram of a machine

Description automatically generated

There are still improvements to be made, for example there’s no circuitry to XOR and no stack implementation yet, however this is a great starting point for building my computer and lets me know exactly what I’ll need for the final design.

**Macro Components**

The final design of this project will be relatively tame, and as such doesn’t require and exorbitant number of parts.

**Core part list.**

|  |  |  |
| --- | --- | --- |
| **Part** | **Purpose** | **Number Required** |
| Arithmetic and Logic Unit  (ALU) | Do the following to 8-bit numbers in registers 1 and 2, then output to register 3:   * Add * Subtract * AND * OR * NAND * NOR * XOR * XNOR | 1 |
| Registers | Receive, store, and send 8-bit numbers on the bus. | 5 |
| Condition Unit | Checks the 8-bit number in register 3 if it meets one of the following conditions, if it does change the program counter’s value to what is in register 0:   * Never * Always * Reg3 = 0 * Reg3 ≠ 0 * Reg3 ≥ 0 * Reg 3 > 0 * Reg 3 < 0 * Reg 3 ≤ 0 | 1 |
| Program Counter | Counts up by 1 each clock cycle, then outputs that number to memory. If it receives a signal from the condition unit then it will change its value to the value in register 0. | 1 |
| Memory | Stores the program of the computer in 8-bit numbers called opcodes. The opcode output will correspond to the program counter’s output. | 1 |
| Instruction Decoder | Does one of the following with the current opcode:   * Put the opcode value into reg0 (immediate) * Copy from one register to another (copy) * Do a mathematical operation with the ALU (compute) * Check if a condition is met with reg3, if it is change the program counter value to the value in reg0 (condition) | 1 |
| Clock | Outputs a square wave to all synchronous components. Can run automatically or manually. | 1 |

These are the main bits, but I’ll need “glue logic” as well.

**Ben Eater’s 8-Bit Breadboard Computer**

The inspiration for this project, I went through and rewatched Ben Eater’s 8-Bit breadboard computer series in order. There are a lot of architecture differences between Ben Eater’s computer and the computer in Turing Complete.

The breadboard computer is relatively limited in memory, having only 16 bits of RAM for the program, but covers the more important computer elements that drive this project – namely microinstructions.

Close-up of a circuit board

Description automatically generated

This computer is largely based on the Simple as Possible 1 (SAP-1) computer architecture laid out in “Digital Computer Electronics”.

For my final design, I would like to closely mimic the SAP-3 architecture in Digital Computer Electronics

A white cover with black text

Description automatically generated

This week was largely a research week, and as such doesn’t include a lot of direct project progress. However, this is a vital step in finishing the computer on time.

**Latches**

1. I will not be manufacturing resistors and transistors. [↑](#footnote-ref-1)
2. If your project is going to run at a very high clock speed, or you care about transistor saturation you should look at Schottky TTL. [↑](#footnote-ref-2)
3. Yes, we’re using APSC 100-101 terminology. [↑](#footnote-ref-3)
4. A Turing complete computer, given enough time and memory can compute any computable problem. [↑](#footnote-ref-4)