Fundamental Analysis of Computational Structures

# Table of Contents

[**Table of Contents**](#_5ah7cg7ajooy) **2**

[**Introduction**](#_5trzerb9pfns) **5**

[**Applications**](#_3x7dmxu2m2mz) **6**

[**Design History**](#_xi6n9aevi6z0) **7**

[**Specifications**](#_hvlbhj3l56nl) **8**

[Transistor Count](#_kgsmc1a1tjmf) 8

[Voltage Level](#_117scqqxh6d1) 8

[Acceleration Sensitivity](#_a9i55eh0xvuh) 8

[Speed](#_9g604cngr8jg) 8

[Bus Sizes](#_m4d4ib8murvf) 8

[General Operation](#_rqvc81p5c15h) **9**

[Power](#_kmawqa6xsqc) 9

[Program Loading](#_ouuah3p16qi9) 9

[**Operation Paradigm**](#_navbw6jnvwd) **9**

[Noise Tolerance](#_7tczkl33lxrr) 9

[Instruction Rom](#_d2lddiwhooe) 10

[DPN](#_p7z1huakfxvh) 10

[Phases](#_bljlp9jvbsi) 10

[Reset](#_bljlp9jvbsi) 10

[Fetch/Execute](#_bljlp9jvbsi) 10

[Bus Scheduling](#_bljlp9jvbsi) 10

[Masked Register Writes](#_nebo93jnmz62) 10

[Conditional Jumps](#_nebo93jnmz62) 10

[RAM](#_nebo93jnmz62) 10

[Output Stages](#_nebo93jnmz62) 10

[Input Stages](#_nebo93jnmz62) 10

[Overall System Diagrams](#_cpsay0f0i4nc) **10**

[System Functional Diagram](#_o1sp9vds0qvu) 10

[System Circuit Diagram](#_d15kgba38n2t) 12

[System Waveforms](#_uk5hspa3kdj3) 12

[**Card Layout**](#_unjrsqpd5wdh) **12**

[DPN](#_qu1lmd5skfvp) 12

[Z-Pass Single Input](#_hfmxec5xh675) 12

[Z-Pass Double Input](#_uu5f7bsrjkpi) 12

[Value-Reset Register](#_r8gir8brxxf6) 13

[Mask Register](#_ftucfmirqccy) 13

[Jump-or](#_s3f3229fxmad) 13

[Base](#_jlunz59nes3q) 13

[Logic Circuitry](#_ezgdfrrf73nr) **13**

[Sram Direct Register Bit](#_47hgcyw8tu6r) 13

[Sram Mask Register Bit](#_rff7hzfjjztu) 13

[Sram Direct Register Bit with Reset](#_5156uve4l245) 13

[Logic Gates](#_q2wdej9hdl3a) 13

[Z-pass](#_l6167u3kx7ng) 13

[LED Display Bit](#_p0uw064kpn6r) 14

[Clock](#_sx7rb7w6vdun) 14

[Schmitt Trigger](#_bnvdvvy71058) 14

[Coupling Capacitors](#_sy1bckk82bwr) 14

[**Instruction Set**](#_ide82o3ih2ue) **14**

[Instruction Form](#_55ujnybx8w) 14

[Op Codes](#_o7hdzo1uzan6) 15

[**Micro-Instruction Scheduling**](#_kbtr7tswho8m) **16**

[**Entropy Analysis**](#_4ayfqvf2c8jz) **17**

[**Methods of Complexity Analysis**](#_gtlgurvavj62) **17**

[**Conjectures**](#_v42w5bgazext) **17**

[**Boundary Theory of Computation**](#_qrjpsf4q4uss) **20**

[Programming Paradigm](#_uurm1kx5b3hz) **21**

[Machine Code](#_z03znyapk1mv) **22**

[Micro-Instructions](#_z03znyapk1mv) **22**

[RAM Packing](#_z03znyapk1mv) **22**

[ROM Packing](#_z03znyapk1mv) **22**

[Fundamental Algorithms and Procedures](#_z03znyapk1mv) **22**

[Equality Testing](#_sjeg5w5toq5l) 22

[Program Address Entropy](#_en670znalcbv) 22

[If-Else Statements](#_sjeg5w5toq5l) 23

[Edge Detection Successor](#_sjeg5w5toq5l) 23

[Repeated Successor Addition](#_sjeg5w5toq5l) 23

[Virtual Logic Gate Addition](#_sjeg5w5toq5l) 23

[If-Else Addition](#_sjeg5w5toq5l) 23

[Repeated Addition Multiplication](#_sjeg5w5toq5l) 23

[Stack and Heap](#_sjeg5w5toq5l) 23

[Function Calls](#_sjeg5w5toq5l) 23

[I/O Bus Procedures](#_sjeg5w5toq5l) 23

[Parallel Instruction Procedures](#_sjeg5w5toq5l) 23

[Multi-threading](#_sjeg5w5toq5l) 23

[Path Indexing (deprecated)](#_sjeg5w5toq5l) 23

[Matrix Return (deprecated)](#_sjeg5w5toq5l) 23

[**Flakiness Constant**](#_5770vgdoh3wg) **23**

[**Function Degrees**](#_57of85a5szn5) **24**

[**Satisfaction of Four Basic Functions**](#_2910y7tf9xdx) **24**

[Arbitrary Data Writes](#_f8v2gxeonvcg) 24

[Arbitrary Array Access](#_6r3mgtoh9eru) 24

[Arbitrary Program Branching](#_i143hftwsjhx) 25

[Virtual Machine of Itself](#_5enbhjhju7ml) 25

[Design Challenges](#_9c9txmyl6axz) **25**

[Voltage Level](#_vk9gm464yig) 25

[Speed being 50x Slower than Expected](#_f3hnx3ag6f2m) 25

[Poor Connections](#_pebwoqmqhjej) 25

[Timing test for register was screwed up](#_nu4ojbv65sm5) 25

[Difficulty soldering tsop shits](#_o21mt454j14r) 25

[**Methods to Write Programs**](#_uce1kd8m2kog) **25**

[Upload Procedure](#_6pr8iwkst0ax) **25**

# 

# Introduction

This document will focus on mathematical conjectures, observations about real world mosfet behaviour on macroscopic scale, computer systems with “distributed” ALUs, and applications to modern technology. Of course also focuses on the computer itself.

Wanted to make a discrete computer for shits, ended up narrowing down on a discrete CPU because RAM and ROM require hundreds of thousands of bits of information to perform basic tasks, and this translates to millions of transistors. The cpu can get away with just having a few dozen register bits, which is much more manageable. Also decided to go with a program stored in ROM so that the program can just be loaded. Settled on cmos logic. The voltage range is a bit limiting but it provides the fastest results (nmos has a slew rate ~80% of cmos and pmos has a slew rate of ~30% of cmos from tests I conducted with the 2n7002et1g and bss84,215).

Brag a lot about the computer here. This is likely going to be read first, so it needs to be interesting and also a huge flex. Don’t assume that the computer speaks for itself, because it doesn’t. Without anchoring to form an opinion, this project can easily seem underwhelming even though it took a massive amount of creative and technical insight.

This bad boy should cover most of my journey through computer hardware and software design from October 2016 through November 2020. I guess a good place to start would be a description of the project. DESCRIPTION blah blah blah wanted to make discrete computer blah blah realized that you can’t even define something as a computer because there is no bright line etc. Objective is to create a computer that is functional but extremely low transistor count. Useful in applications where power consumption is critical, or just for low cost stuff. Like, you could put 2^24 of the map25.2 on a typical dye these days (with that sexy 5 nm technology). The one I created has 254,000 nm feature size and 5,000,000 nm transistor size, which is balls horrible. Also increase clock from 456 kHz to 5 GHz. Ok this needs to be moved to the other section. I will include the “useful” discoveries and inventions first, then the other crap. Note that notes from first few years were very lacking because everything was just burned into my memory (inventing everything from scratch will do that). I have included as much detail as possible in this thesis for all years. I tried to keep this purely based on my own inventive process, but of course I was polluted by outside sources. This is why I actually know what CISC, RISC, ALU, etc mean.

There are enough experiments here to write a dozen research papers at least. Should find a way to break it up and start publishing as useful stuff.

Many people underestimate the value of analyzing computation from a very low level. Research into the pure mathematics of computers and the cutting edge of software exists, but comparatively little attention is given to the nexus between the two: small scale computers and efficient design in both software and hardware. Closer analysis into this technology provides insights into the very construction of computers. The construction of the computer is literally the foundation for all software, so any application of the computer relies on intelligent hardware design. Furthermore, insights into hardware design provide insights into various programming paradigms. Example: procedural vs functional vs object oriented.

# Applications

I’m just going to get this out of the way: this architecture is not meant to be very useful in the real world; it is mostly just an exploration into the nature of computing. However, I believe that offloading the work of the ALU into the rest of the system can prove useful in scenarios where all you need is for the cpu to be tiny.

This cpu architecture is actually extremely useful in low power applications. Also, if you just need to get something done without being too complicated. Also, extreme parallelism. Also, if just need something to direct the activity of peripherals (eg you offload the heavy lifting to the gpu, etc). This cpu is basically an exploration into the simplest possible architecture you can have that still does things. This architecture can just be dropped onto just about any chip, even a monolithic chip, wherever you need something to do something. You can use a diode array instead of eeprom for these sorts of applications.

This project also provided mathematical insights. For example, I hope that I have at least been able to further humanity’s understanding of the P vs NP problem a little bit through my own exploration into the nature of computing. Spoiler alert: the closest I came to defining a computer was to realize that the task is literally impossible. Actually, proving this would be cool. Hmmmmmm.

This computer design process serves as a window into “program entropy.” Entropy describes how data is stored. Specifically, it can be a value in a register, ram, program read address. Describes machine state. Also provides a look ALU distribution. Basically, everything provides the ability to combine the function of the ALU and complex entropy handling into a VERY elegant system that combines these processes down to a very low level.

A grey area exists between mathematical turing machines and general purpose personal computers. This project establishes an in-between ground where complexities are measured as polynomials instead of exponential functions and computers manipulate entropy and perform mathematical functions in a way that can be applied readily to real life programs.

# Design History

Note: you should make sure to identify the 4 machines you actually built.

0. transistor messes

1. load/write/and/ifand (page 4)
2. first attempt at connections and gates (page 7+)
3. 78 pins machine
4. Miscellaneous machines
5. 63 tick machine
6. copy/write1/write0/if machines
7. Page 49
8. Page 51
9. Page 53
10. Page 55
11. Page 63
12. Page 85
13. Page 95 -> simplest possible machine but exponential efficiency
14. Page 1, page 3
15. Page 9
16. Page 19 -> built this guy
17. Page 43
18. Page 57 (built this guy)
19. BCP 0
20. created 5/16/19
21. created 5/16/19
22. created 5/20/19
23. TACP
24. VBP 5 (642)
25. QVBP 5
26. Quartic VBP 6
27. QVBP 7
28. VSBP 7.0
29. VS-P 8 (built this guy)
30. Splix 0.0, Splix 0.1, Splix 23.2
31. Map 24.0, Map 24.1, Map 24.2, Map 24.3
32. MAP 25.0, map 25.1, map 25.2 (built this guy)

# Specifications

## Transistor Count

2,861

## Voltage Level

2.65~3.05 V

Voltage fluctuation is ± 120 mV

## Acceleration Sensitivity

100G while not operating, less while operating.

Basically be gentle and dont do stupid crap because it is rather fragile and heavy, which is a horrible combination.

## Speed

| Clock Speed  Instruction Speed  Output Rise/Fall Time | 456 kHz ± 5,000 ppm  1 clock per instruction  180 ns |
| --- | --- |

## Bus Sizes

16 bits for like everything to make it ~clean~ and ~elegant~.

# General Operation

## Power

Note that there is no voltage regulator on the batteries (2x AA), so the burden falls on the user to ensure that the voltage across both batteries is within the range (2.65, 3.05). Just switch on. Reset circuitry is designed to automatically reset to address 0000h.

## Program Loading

Use zifs. See the last section for more details.

# Operation Paradigm

Overall stuff. Use LEDs like an MRI of the machine during operation. ZIFs too.

## Noise Tolerance

There will always be spikes. The deal here is that spikes are designed to 1) only occur when something is actually changing and not when the value is persistent and 2) everything can tolerate edge spikes. So when phases go 00 ->10->11->01 there are no spikes in the square waves except for on the edges, but nothing is edge-triggered, so it is fine. Designed so that bus is charged AND stabilized AND persists for writing to registers, and writing doesn’t care about edge necessarily.

All of this is to say that this design is just fucking beautiful and crispy.

\*\*Inset diagrams\*\*

## Instruction Rom

## DPN

## Phases

## Reset

## Fetch/Execute

## Bus Scheduling

Also has gen rotation btw.

## Masked Register Writes

## Conditional Jumps

## RAM

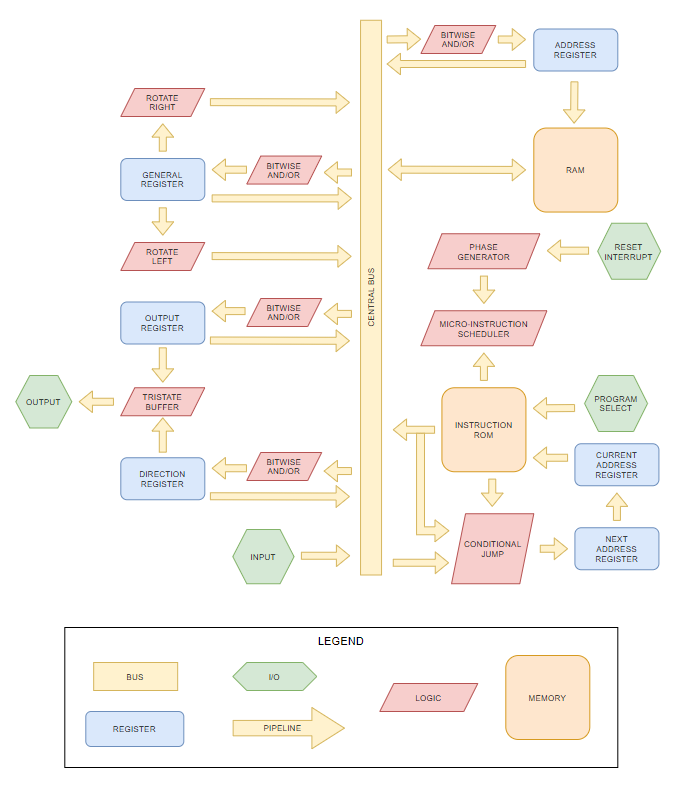
## Output Stages

## Input Stages

# Overall System Diagrams

## System Functional Diagram

This diagram is effectively what the system looks like from a programming point of view. Note that this diagram is not spatially accurate to the physical machine, and each block does not correspond to a card in the physical machine.



## System Circuit Diagram

This diagram corresponds directly to the physical layout of cards in the real system. The position of data buses, power buses, indicator lights, batteries, I/O busses, RAM, ROM, and selector switches is also accurate.

## System Waveforms

# Card Layout

This will break things down to the level of logic gates. The design corresponds spatially with the physical machine. Data connections as well as power connections will be drawn for the sake of completeness.

## DPN

Pretty freaking complicated, a lot of stuff jam packed over a card and a switch on the main board. \*\*waveforms\*\*

## Z-Pass Single Input

Eg tristate buffer between registers, etc and the bus

## Z-Pass Double Input

Eg the tristate buffer on the output stage

## Value-Reset Register

## Mask Register

## Jump-or

## Base

# Logic Circuitry

This is the transistor-by-transistor layout of the blocks referred to above. Again, it will be spatially accurate.

## Sram Direct Register Bit

This just holds a single bit. Has “data” and “latch enable” as inputs and Q and Qbar as outputs. Decided to go with an SR-style design so that there was no jakey crow-bar circuitry to switch the bits as there is with most cmos sram bits these days.

## Sram Mask Register Bit

Same as the previous one, but “latch enable” is the input bus and “data” is separated into the write-high and write-low channels, with one being selected based on what data you want to put on the card. Note that, if both are on, it will be fine and nothing will burn up, it will just be undefined behavior.

## Sram Direct Register Bit with Reset

Just put an extra input on one of the gates in the SR latch to yank it to zero upon reset.

## Logic Gates

These are just your standard, run-of-the-mill logic gates. I was lucky in that I didn’t have to put any gate resistors, bleed resistors at junctions, or stabilizing circuitry. The drivers are just direct connection to other gates without any sort of current booster, meaning the gates are relatively slow.

## Z-pass

This is just a tri-state buffer. It can be accomplished in cmos or nmos. An interesting effect of using cmos is that you can JUST enable a 1 or high-z or JUST a zero or high-z. Thus, enabling comes in pairs of enabling 1 and enabling 0.

## LED Display Bit

Just had to write a driver to send info to it.

## Clock

Not gate oscillator circuit, works pretty good, disappointing its not a crystal tho.

## Schmitt Trigger

This is a bit jank because it uses a resistor on the input and across the trigger, so it expects the input to be able to provide a few dozen microamps (which shouldn’t be too bad).

## Coupling Capacitors

Make everything a bit cleaner.

# Instruction Set

## Instruction Form

<16 bits to specify current address> <4 bits for upper op code> <4 bits for lower op code> <16 bits to specify immediate value> <16 bits to specify next address>

Distribution over EEPROMS:

EEPROMS are enumerated in little endian, looking at the board from the direction where you can read the text (MAP 25.2). Also, the pinouts for the data, address, control, and power on the eeproms is matched exactly to the gls29ee010 datasheet as these are the EEPROMS I have elected to use.

EEPROM 4: op code

EEPROM 3: upper 8 bits of immediate value

EEPROM 2: lower 8 bits of immediate value

EEPROM 1: upper 8 bits of next address

EEPROM 0: lower 8 bits of next address

Example:

Program looks like this (all numbers are in binary; spaces are only for legibility).

## Op Codes

upper 4 bits:

0x0 => "imm", "0" (immediate value is put on bus)

0x1 => "adr", "addr", "1" (address register is put on bus)

0x2 => "gen", "2" (general register is put on bus)

0x3 => "rol", "3" (general register rotated left is put on bus)

0x4 => "ror", "4" (general register rotated right is put on bus)

0x5 => "out", "5" (output register is put on bus)

0x6 => "dir", "6" (direction register is put on bus)

0x7 => "inp", "in", "7" (input is put on bus)

0x8 => "ram", "8" (value of ram at the address register address is put on bus)

0x9 => "dnc", "nc", "9" (bus is not connected, so it is pulled high)

0xa => "a" (same as 0x9)

0xb => "b" (same as 0x9)

0xc => "c" (same as 0x9)

0xd => "d" (same as 0x9)

0xe => "e" (same as 0x9)

0xf => "f" (same as 0x9)

lower 4 bits:

0x0 => "jzor", "jnz", "jumpor", "0" (the bus and immediate are bitwise "and"-ed,

then the result is "or"-ed with itself and this is used as the last bit

of the next program address value)

0x1 => "asnx", "asnnext", "1" (next program address is set equal to bus value)

0x2 => "out0", "2" (output register is set to zero with bus as mask)

0x3 => "out1", "3" (output register is set to one with bus as mask)

0x4 => "adr0", "addr0", "4" (address register is set to zero with bus as mask)

0x5 => "adr1", "addr1", "5" (address register is set to one with bus as mask)

0x6 => "dir0", "6" (direction register is set to zero with bus as mask)

0x7 => "dir1", "7" (direction register is set to one with bus as mask)

0x8 => "gen0", "8" (general register is set to zero with bus as mask)

0x9 => "gen1", "9" (general register is set to one with bus as mask)

0xa => "noop", "a" (no operation is performed)

0xb => "b" (same as 0xa)

0xc => "c" (same as 0xa)

0xd => "rlow", "ramlow", "d" (ram lower byte at address register

address is set to the value of the lower byte of the bus)

0xe => "rupp", "ramup", "rup", "e" (ram upper byte at address register

address is set to the value of the upper byte of the bus)

0xf => "rall", "ramall", "f" (ram upper and lower byte at address register

address is set to the value of the bus)

## Micro-Instruction Scheduling

Insert all of that crap, including scheduling for reset.

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

# Entropy Analysis

\*\*\*This is hella crappy analysis with like 50 logical fallacies, and it misses basically everything\*\*\*

Analyzing the system in terms of entropy can be helpful for classifying the speed of various algorithms. As the system is set up, there are four 16 bit registers (excluding “curr” and “next”), 65,536 words of ram (each word is 16 bits), and 65,536 words of rom (each word is 40 bits, the instruction length). Hence, there are 64 bits of entropy that can be used with only 1 clock (the registers) and 1,048,576 bits of entropy that can be used within 3 clocks (eg load the pointer and then use the value in ram). The 2,621,440 bits of ROM entropy are used in a slightly different way; the immediate values are used in much the same way as ram, except that they are indexed to based on the state of the “curr” register, so the entire thing is more complex. The position of the system (eg the value in “curr”) can be used as single instruction access entropy (16 bits of it), but it has the added advantage that it will change the STATE OF THE MACHINE ITSELF. That way, data can be stored in what the system is doing. This is great for arrays for functionality changes.

# Methods of Complexity Analysis

Analyze in terms of RAM, ROM and delta-time. Clocks directly proportional to time so may be used interchangeably. Also, calling this “efficiency” because that’s what I came up with before knowing what complexity is.

# Conjectures

\*\*Your basic functions require measurement of complexity of any arithmetic operation, wich doesn’t work.

\*\*Go ahead and clock computing??

\*\*Multi thread minimum?

Conjecture A: Every computable program can be described as some combination of the four fundamental functions.

Conjecture B: Computational speed can be described as the speed at which the four basic functions are performed.

Conjecture C: The “efficiency” of a program can be described as the big O notation for increase in a) RAM b) Program length and c) time as a function of word size.

Conjecture D: The size of a program in ROM is a function of the computer’s physical architecture and the architecture of the machine code.

Conjecture E: “Or” logic on its own is enough to create all logic gates.

Conjecture F: Mask writes to registers is just as effective as program complexity increases as an entire ALU would be.

Conjecture G: Mask writes with rotations up and down is ONLY PROPORTIONALLY slower than a computer with an ALU. Maybe also talk about how you still have to evaluate undecidable problems or whatever for the max amount of time on both machines; there is no algorithm to make it faster. -> oh shit but you might be able to make faster algorithms -> ok but the “better” computer can be simulated by programming on the “worse” computer, just replacing the “add” instructions and things like that with simpler stuff

Conjecture H: Increasing the complexity of the ALU or computer architecture in general only increases speed of defined functions; there will always be functions that must be explicitly defined down to a bitwise level. P vs NP?

Conjecture I: It is fundamentally impossible to define a “computer.”

Conjecture J: 100% ROM packing is impossible. (This might just apply to map 25.2).

Conjecture K: No computer will be capable of performing every function with maximum efficiency (maybe this should be an fpga???????) because the complexity blows up and the complexity of the computer is constant.

Conjecture L: Two pcb layers is always enough.

Conjecture M: Recursion is not powerful enough to describe positional entropy data storage

The classical forms of positional entropy operate either as FIFO or using the function stack to cheat. The only way to create a non-FIFO entropy system is to use goto.

Conjecture N: Knowledge and power of a program is an exponential function of time

Conjecture O: The smallest number of registers required for a non-exponential turing machine is 4 (if no complex logic is used): next register, current register, address register, output register; for this you can use path indexing

Conjecture P: The fastest edge detector boils down to the speed of log2(n) because it has to hone in on the edge, even in physical systems

Conjecture Q: VM operations can only be as efficient as base machine operations.

Conjecture R: 2 registers is required for array data transfer

* Can’t go back and replace fx call with address immediately because fx calls might be ahead
* Array by using next register?
* Alphabetize the labels to find edge quickly
* Don’t initialize currlabel; randomness can help with bug checking since it should work with any label unless there is a weird error in code writing

\*\*\*the 2 register transfer thing also proves that subsequent virtual machines of the map25.2 will suffer the same fault; virtual machines cannot create new logic out of nothing

\*\*\*cover the fact that, if you make functions of deference based on previous results of deference, your computer will require an increasingly large number of registers, then those registers will need to be deferred to; basically, self-editing code is required to avoid this maybe??

\*\*talk about different function family extensions of the four basic functions, and how their complexity varies as a function of this new complexity that has a range across clocks, ram, and rom; and how ram and rom and time changes in the algorithm affect each other

Families:

* Arithmetic
* Deference

Branching power of a computer (from least powerful to most powerful); note that this is regarding the code AFTER it is compiled, so things like macros are obviously excluded:

1. No branching, eg linear counter
2. if/else if/else
3. For loops without breaks
4. For loops with first order breaks (eg what happens in c)
5. Recursion of functions via the stack
6. Goto’s, where entropy is stored in position instead of the stack in a non-FIFO fashion
7. Self-editing code, which is the king of branching, because everything can be re-defined
8. Depth of programming languages and a rigorous definition; eg windows operating system is the integral of python which is the integral of c which is the integral of assembly which is the integral of machine code which is the integral of cpu circuitry design which is the integral of electrical engineering which is the integral of applied mathematics which is the integral of number theory which is the integral of logical axioms that we assume to be true.

Relationship between integration and deference and increasing complexity

Every program can be written as a complex expression where the three operators are data write, data point, and program branch

Write the entire processor as a function of logical operators; this is the mathematical manifestation of the logic gate diagram which is the logical manifestation of the block diagram. Allows for mathematical manipulation with calculus, limits, etc.

Make sure to talk about the simplest cpu, one where all manipulation is a result of the position of the system in rom.

\*\*\*NOTE THAT, for polynomial time and entropy efficiency, zero registers is possible, it just involves tate buffering and path indexing layered together, but to get word efficiency, you need two registers (all of this is regarding placing a value at an immediate in an array whose pointer is at an immediate)

\*\* see if a turing machine with a portion that indexes to RAM can do stuff in less than exponential time complexity

\*\* talk about how provides a model for power in terms of exponential input increase (eg array processing) without exponential time increase, and the entropy increase is only proportional to the data because, well, the data has to live somewhere

# Boundary Theory of Computation

Turing completeness provides a way to describe a computer that can compute any “computable” problem given infinite time and memory. This definition is useful for defining what a computer is an is not; however, Turing was never able to define “computable.” The infinity of time and memory also means that most algorithms with have a complexity/efficiency that is exponential.

I take a more practical approach thinking in terms of the P vs NP paradigm. NP problems require brute force to evaluate, leading to the exponential complexities above. However, universal turing machines also have exponential complexities when it comes to problems that should have polynomial complexities or transcendental complexities that grow slower than any exponential function. <insert good examples>

My definition of a “powerful computer” is as follows:

All problems that should be evaluated in a P way can be evaluated in a P way. The computer is still able to solve NP problems, but they will require atrocious amounts of energy. If a computer is able to compute the following four basic functions, it is guaranteed to behave in the specific way I just mentioned: arbitrary data manipulation, arbitrary array access, arbitrary branching, and VM of itself.

Because NP is exponential and equivalent to look up table, for the functions in P to be “complete:”

\*actually, NP can be a lot worse than a look up table? Maybe? Because have to evaluate to the end -> but np is where you don’t know the answer but can find by brute force; look-up table requires you to know beforehand -> “look up table” is the wrong terminology here i think, not the NP thing

Limit as x goes to infinity of C0(x)/ax = 0 for all a > 1 and lim as goes to infinity of C1(x)/ax = 0 for all a > 1. C1 is the complexity function on any computer, and C0 is the complexity on the computer you are trying to measure. Basically, the set of all functions that can be evaluated in a non exponential worst case efficiency on any classical computer can be evaluated in the same manner on the computer you are trying to test.

Furthermore, if the four basic functions can be completed with N efficiency on a system, then all classically computable problems can be computed with N efficiency on that system.

Pay attention to how your algorithms for extremely simple computers can also be applied to things like programming languages, virtual machines, and algorithms in order to determine the most efficient algorithms for them.

\*\*\*In analyzing the system, simply propose this method of program efficiency analysis, then use it to analyze the architectures and algorithms you invented.

# Programming Paradigm

Pretty fluid with the design and operation. The immediate value becomes next if use asnx. Also, eeprom can be used as data rom for a program with packing efficiency of like 12%.

Use random halts to check program integrity

Use indicators as MRI

Use dir register to check.

Also talk about boot up and reset procedures please.

# Machine Code

# Micro-Instructions

# RAM Packing

# ROM Packing

# Fundamental Algorithms and Procedures

## Equality Testing

Set address to where the value is, then load inverse into gen, then test for all zeros where they need to be in corresponding spots. Is very fast.

## Program Address Entropy

## If-Else Statements

## Edge Detection Successor

## Repeated Successor Addition

## Virtual Logic Gate Addition

## If-Else Addition

## Repeated Addition Multiplication

## Stack and Heap

## Function Calls

## I/O Bus Procedures

## Parallel Instruction Procedures

## Multi-threading

## Path Indexing (deprecated)

## Matrix Return (deprecated)

# Flakiness Constant

Just the product of <# of board connections>\*<# instructions required to complete program>\*<probability of failure at a physical joint during the execution of one instruction>

Can be used to describe physical systems, or stuff that you are programming and trying to bug check. Basically, just establishes direct relation involving points of failure, clocks, and probability of failure at a single point

# Function Degrees

Zeroth degree is a macro, first degree involves placing things on the stack and calling (can handle entropy complexity up to recursion); second degree allows for full manipulation of entropy beyond just recursion.

# Satisfaction of Four Basic Functions

These four basic functions will determine the speed of all other functions. In the worst case, eg a 2,3 state machine, each function will require an exponential amount of memory and clocks relative to data size. For this computer, each function requires a constant number of instructions (and this number is very small, like 2 or 3). These functions might seem almost too basic to build all of the functionality of a computer, but rather complex operations such as multiplication can be created from them in a non-exponential big O notation increase. Note that no computer can perform all possible functions in a constant number of clocks; at some point, the functions will become complex enough that they must be built from simpler functions that the computer has. For example, in a computer with an ALU that can only perform addition, multiplication big O notation for memory and clocks in a linear function of bit size.

## Arbitrary Data Writes

Just charge the bus with either imm or a memory/register/input value, then mask write to the register.

## Arbitrary Array Access

Just use gen as a buffer and load the pointer to the array from the address of the pointer, then transfer the value from gen to addr.

Actually, the computer is forced to use brute force entropy transfer when transferring data from one array to another.

## Arbitrary Program Branching

Just charge the bus with the address you want and then use asnx. Note that, if only using jzor, you will have to create a matrix of all possible addresses you might want to jump to because jzor can only assign the last bit of the next address.

## Virtual Machine of Itself

Just define in the program rom memory locations to represent each register and allocate a large portion to represent program rom, then use addr as curr and jump thru ram, performing the basic operations in the virtual machine based on definitions in program rom and jumping between stuff using asnx and/or jzor.

# Design Challenges

## Voltage Level

## Speed being 50x Slower than Expected

## Poor Connections

## Timing test for register was screwed up

This is because the voltage was too high, so the gates did that oscillation thing, so it looked like it was really fast.

## Difficulty soldering tsop shits

# Methods to Write Programs

Define a C program which defines how to write the assembly program. The assembly program is then directly translated into machine code simply by replacing mnemonics.

# Upload Procedure

Take the compressed, page version and write to eeprom using arduino. Use zifs for everything; otherwise, chip tabs will break off after a little while.