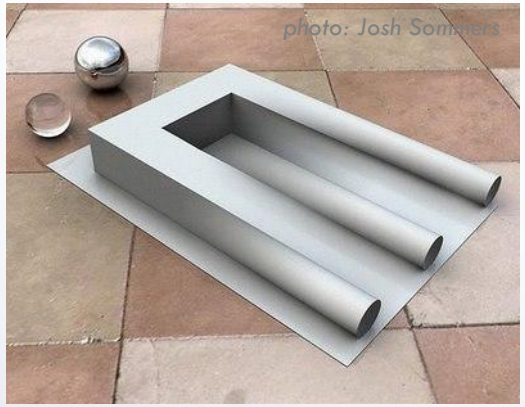


Your CPU Is Binary

…But Soon It Will Be Something Else

C++ now

charley bay

charleyb123@gmail.com

Today's Discussion

1. What Brought Us Here?
   - *(Brief!) Computing-History, definition-of-terms*
2. Contrast Binary *(2VL)* and Ternary *(3VL)*
   - *Ternary is simply the FIRST of the "Many-Valued-Logics" (MVLs)*
3. Review of Logic, "*Law Of Thought*"
   - *Basis of all reasoning (Plato/Aristotle), George Boole's "Book On Logic"*
4. Contrast: Binary APIs, Ternary APIs
   - *Where `bool` is wrenched from your vocabulary*
5. Toward a MVL Future
   - *Which is everything that exists beyond binary!*

Why Are We Here?

- The **CPU is binary**
  - *Therefore*, the **CPU ISA** presents a **binary interface**
  - *Therefore*, **programming languages are binary-based**
  - *Therefore*, **we think in binary**
- *However*,
  - **Binary is not the "real world"**
  - **Logical fallacies inject errors** into our systems
    - **Algorithms are incorrect** *(relying upon "**false choices**" and **incorrect assumptions**)*
    - **Bad reasoning** that does not handle **"normal-and-expected" corner cases**
    - **Bad APIs** that are **easy to use incorrectly**
  - *Ubiquitous* thinking in binary makes us **blind to alternatives**
- **Alternatives Exist!** *…with advantages…*

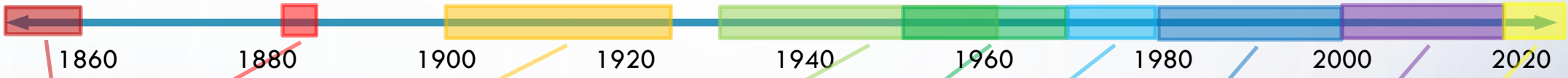
"Wizard Of Oz", 1939

What Brought Us Here?

…A *(very) brief* **Review**, and **History** *(so we are all on the "same page")*…

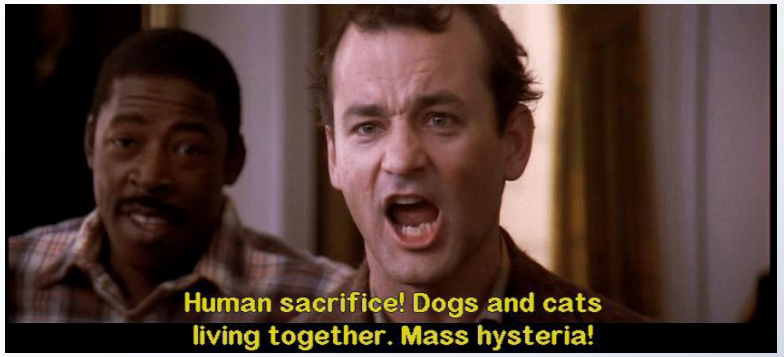
- *Definition of* ***terms***
- *Pay attention to the* ***"decades-in-time"***

Early Computing History: Overview



- 1850's: **Sophisticated vacuum tubes**
- 1883: **Discovery: Edison effect**, electrons flow through vacuum (Thomas Edison)
- 1900's-1920's: **Electricity, Vacuum Tubes, Radio**
  - 1903: Nikola Tesla patents electrical logic circuits called "gates" and "switches"
  - 1904: John Fleming invents diode
  - 1907: Lee De Forest patents the first vacuum tube triode (later used as electronic switch in the first electronic computer)
- 1930's-1950's: **Early Computers**
  - 1936: Z1 (electro-mechanical binary digital computer, used 22-bit word)
  - 1943: ENIAC (first general-purpose electronic computer)
  - 1943: Colossus (first programmable computer)
  - 1949: Manchester Mark 1 (could run a stored program [e.g., "von Neumann Architecture"], perform indirect addressing)
  - 1950: UNIVAC 1101 (could store and run program from memory)
- 1950's-1960's: **Transition to Integrated Circuits (ICs)**
  - 1957: IBM stops using vacuum tubes (releases computer using 2000 transistors)
  - 1958: First integrated circuit (Fairchild Semiconductor and Texas Instruments)
  - 1965: Transistor-transistor logic (TTL) developed by Texas Instruments
- 1970's: **Microprocessors, Consumer PC market**
  - 1971: First microprocessor (Intel), first edition of Unix (Thompson & Ritchie)
  - 1974: Intel's 8080 processor released, used on the Altair Computer
  - 1975: Microsoft founded, release Altair BASIC, later Microsoft BASIC (Bill Gates and Paul Allen)
  - 1976: Apple computer demonstrated (Steve Jobs and Steve Wozniak)
  - 1976: Intel 8086 released
  - 1979: Intel 8088 released, Motorola 68000 (16/32-bit processor) released (and later chosen for Apple and Amiga computers)
- 1980's-1990's: **Consumer PC Explosion**, **"Rise of the Wintel Empire"** – IBM PC, Intel, Microsoft, Apple
  - 1980: IBM hires Paul Allen and Bill Gates to create DOS, granting them ownership of the OS
  - 1981: IBM PC released
  - 1982: Compaq releases a clone of the IBM PC
- 2000's to Today: **Modern Era**
  - 2005: Pentium-D dual-core released (Intel), Athalon dual-core released (AMD)
  - 2010: Multi-core network processing devices are mainstream
  - GHz Processors, Multi-core, Multi-Level CPU cache
- 2020: **"Next Generation"?** **"Non-von-Neumann-Architectures"?**
  - 2014: "The Machine" Announced (HP), photons for communication, ions for storage, specialized processors sharing Universal memory
  - Storage, transport, and processing through quantum effects, optronics, 3D memory, quantum computing

Early Computer Hardware: *Chaos*

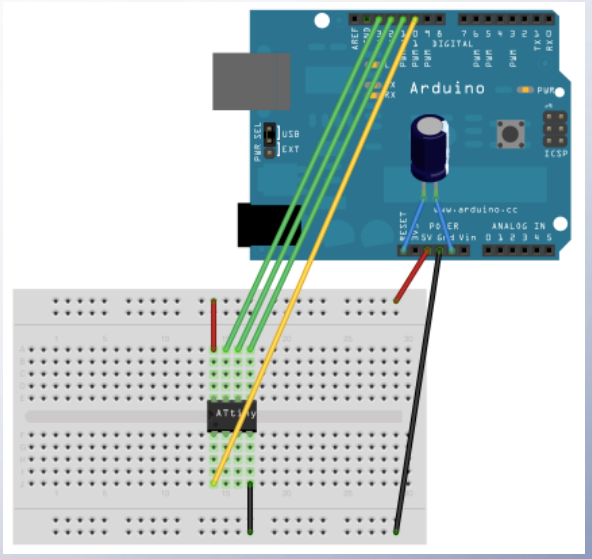


"Ghostbusters II", 1989

- **No agreement on wiring** (signal pulsing, hi/lo, Voltages, rates)
- **No agreement on "computer word-size"**
  - Bus-widths ranged from 1, 3, 5, 6, 7, 8, 9, 12, 17, 18, 20, 36, and 60 bits, depending on the era and the computer
- **No agreement on "byte"**
  - 20-bit byte used to be extremely common *(1950's)*
  - Byte of 6, 9, 12, 18, 30, 36-bits popular in the 1960's and 1970's, and even 1980's
  - 1956: Term "byte" came from phrase "by-eight" *(Werner Busholz, IBM 7030 Stretch computer had variable-length byte of 1 to 8-bits)*.
- **No agreement on radix**
  - **Decimal Computers** *(Base-10)* – wiring was done "by-hand", each wire had 10 states (decade counter or dekade vacuum tube), fewer wires required
  - **Octal Computers** *(Base-8)*, Based on Binary – Three bits to define `[0-7]`, preferred where bus was evenly divisible by 3, but not by 4. (Word sizes of 9, 18, or 36 were considered octal computers.)
  - **Hexadecimal Computers** *(Base-16)* – Four bits to define `[0-9A-F]`, useful when bus is evenly divisible by 4. (Word sizes of 8, 12, or 24 were considered hexadecimal computers.)
  - **Exceptions**: Octal became widely used even on 12-bit, 24-bit, and 36-bit systems that otherwise could have been classified as "hexadecimal computers".
- **No agreement on programming** *(Von Neumann architecture for "stored-program" concept came in 1945)*

Simple Binary Circuits

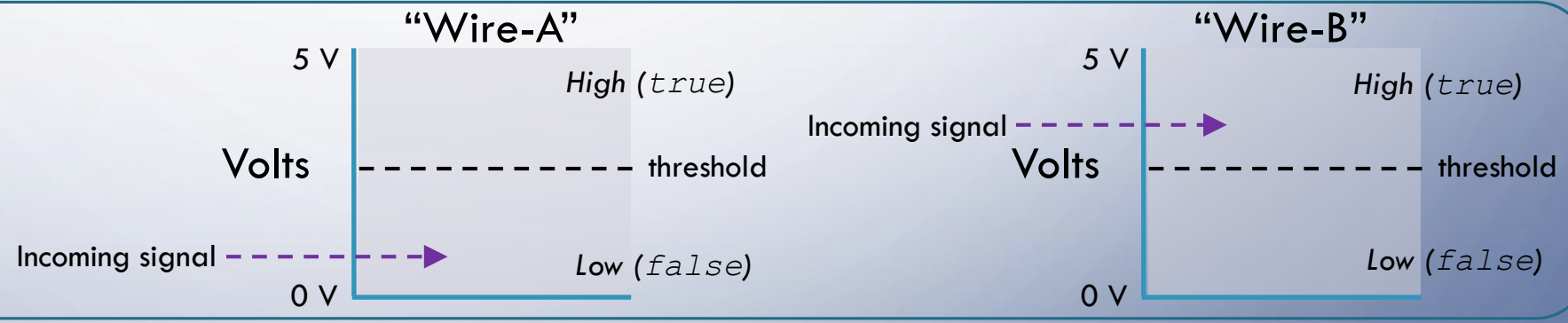
- **The "wire" or "pin" holds a Voltage potential** *(as measured from "ground")*
- We **"set" the Voltage**
  - Set the wire to "`high/hi`" … represents `true`
  - Set the wire to "`low/lo`" … represents `false`
- We later **"read" the Voltage**
  - Read "`hi`" … wire is `true`
  - Read "`lo`" … wire is `false`
- **Why** would you do this?
  - If we read the value we previously "set", wire is **acts as MEMORY**! *(Gives us back the value we wrote!)*
  - If we read the value someone else "set", wire is **acts as a BUS**! *(Gives us the value someone else wrote!)*
  - If we "set" some wires, wait, and "read" other wires: **We computed something**! *(We get computed results from our CPU!)*



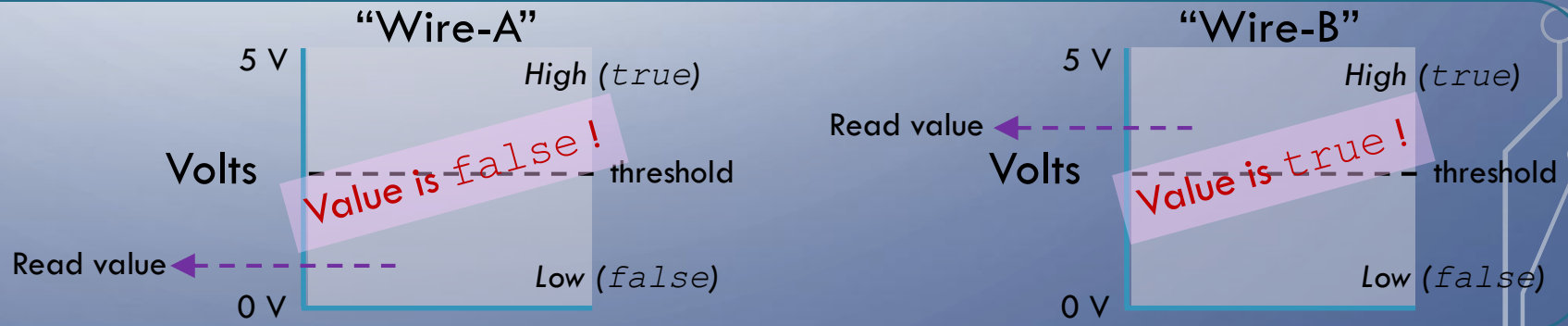
Simple Binary Circuits *(In Theory…)*

- **Physics Phenomenon**: Circuits are subject to fluctuations
  - *Stray voltage drops, radio frequency (RF) interference, heat fluctuations, cosmic rays, imperfect connections, aging components, etc.*
- **In theory**, the binary wire only needs a **"Voltage threshold"**:
  - Above threshold, value is `true`
  - Below threshold, value is `false`

1. **Record incoming signal** *(set Voltage to this level)*



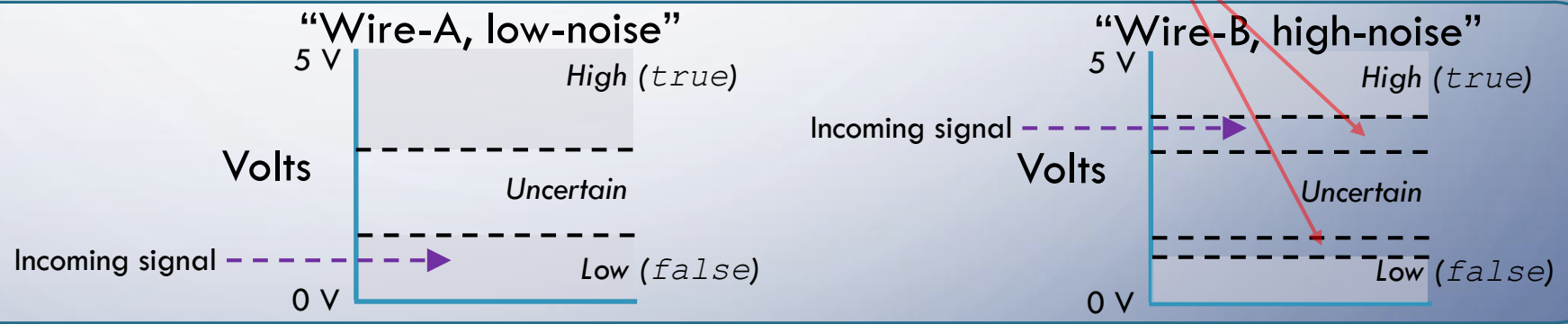
2. **Read value recorded**



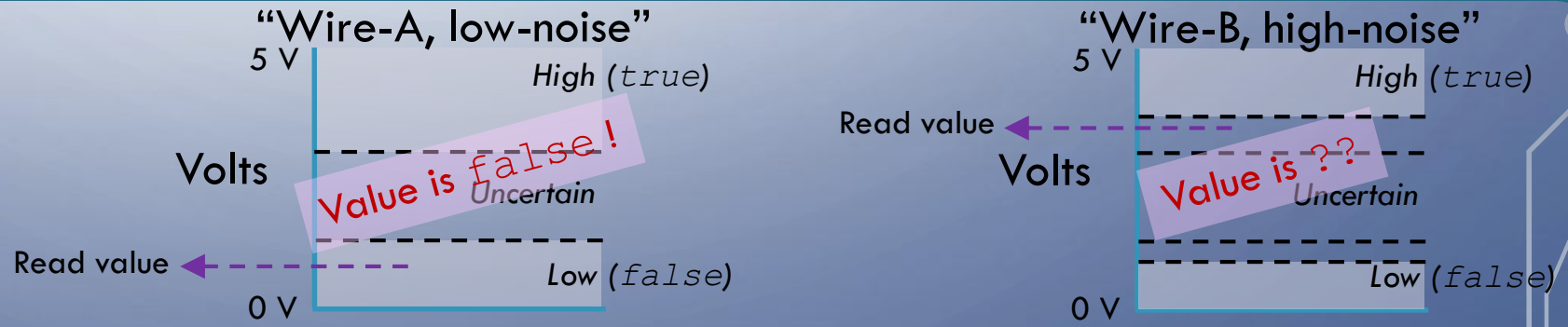
Simple Binary Circuits *(In Reality…)*

- **In reality**, the binary wire has **ranges that are "certain"**:
  - High-Voltage range, value is `true`
  - Low-Voltage range, value is `false`
  - Uncertain range, value might be read as either `true` or `false`

1. **Record incoming signal** *(set Voltage to this level)*



2. **Read Value Recorded**

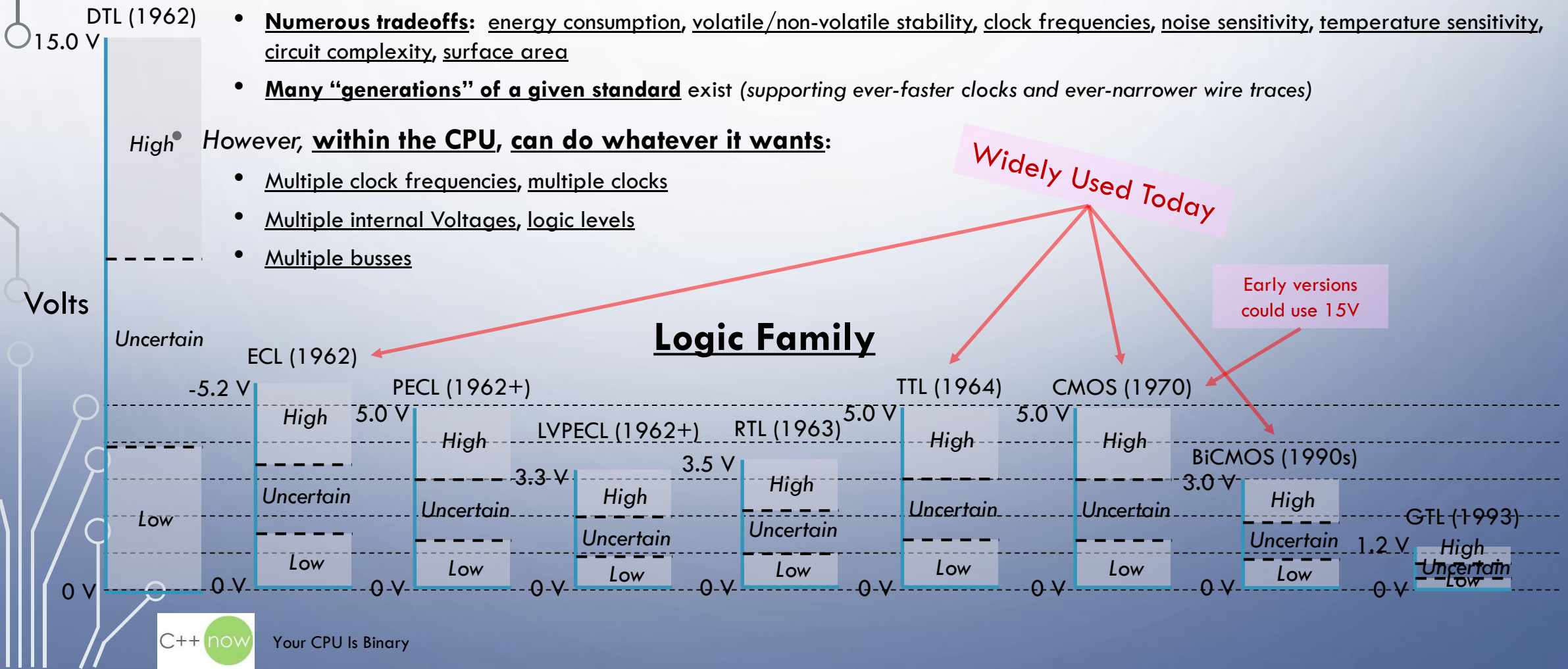


Logic Family: Establishes Logic Levels *(Voltage Ranges)*

- **Logic Levels**: The **number of finite states the wire can discriminate** *(binary has 2 logic levels)*
  - a wire only holds **one value at one time**
- **Logic Family**: **Defines technology**, **power supply**, **frequencies** to **establish logic levels** *(Voltage ranges, margins)*
  - **Numerous tradeoffs**: energy consumption, volatile/non-volatile stability, clock frequencies, noise sensitivity, temperature sensitivity, circuit complexity, surface area
  - **Many "generations" of a given standard** exist *(supporting ever-faster clocks and ever-narrower wire traces)*

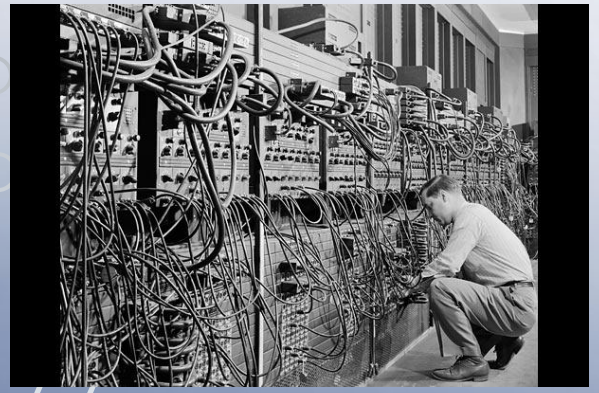
However, **within the CPU, can do whatever it wants**:

- Multiple clock frequencies, multiple clocks
- Multiple internal Voltages, logic levels
- Multiple busses



Early Advantages Of Binary

- **Simplicity**:
  - **Only 2 logic levels**! Binary "`lo`" and "`hi`" simpler than if we had more value ranges *(3 or more)*
- **Noise immunity**:
  - Allows for **wider voltage tolerances**; Random voltage fluctuations **less likely to generate erroneous signal**
- **However, is different world today!**
  - **THEN**: **Noisy copper wires**
  - **NOW**: **Mature electronics industry** with **amazing component tolerances**.



ENIAC, 1946
"First General Purpose Electronic Computer"

Hardware Then:

- Non-standard
- Unreliable
- Expensive
- Poor availability

Hardware Now:

- Standardized
- Inexpensive
- Reliable
- Commoditized
- Amazing advances in manufacturing science, and materials science

Tangible Computing, 2015
"Hand-held cubes to run Spotify"

Today: Binary Inertia

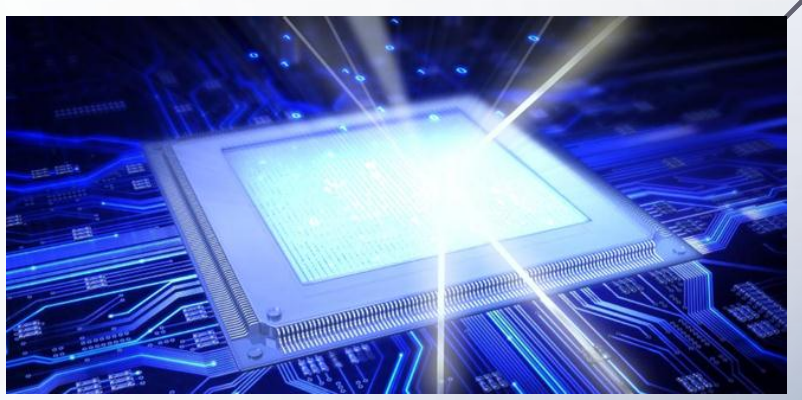
- **Binary electronics** fundamentally favor a **power-of-two**
  - **Hardware designers make sizes a power-of-two** *(adding another binary wire doubles the range of values)*
- **Binary hardware** creates **HUGE implications for the programmer** reasoning about performance and behavior!
  - **Hardware** provides a **binary interface**
  - *So,* **Programmer uses binary (bivalent) logic** to reason about the hardware
  - *So,* **Programming languages are optimized for binary** *(native types, branching logic)*
  - *For example,* Programmer reasons about **data alignment** and **cache strategies** on **powers-of-two boundaries**
- **Power-of-two encourages number systems** such as octal *(base-8)* or hexadecimal *(base-16)* *(for compact representation, and to enable "bit" or "fractional" manipulation of larger numbers)*
- **Binary breeds binary!** New CPUs are *(encouraged to be)* binary, in order to interface with existing binary CPUs, memory storage, bus interfaces, and network interfaces!)

Binary Works.

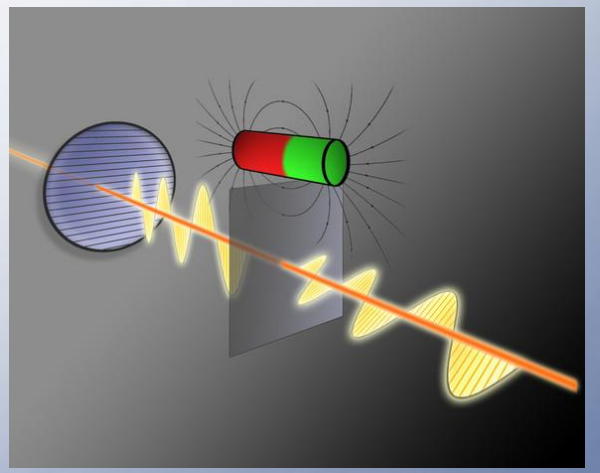

everything's not fine
theprofoundprogrammer.com


CHECK
ENGINE

Why Move From Binary?



1. **Hardware**: *"The End Of Silicon"*
   - New hardware **natively supports more states** *(binary is inefficient)*



Optical Transistor
(Adjusts light polarization via electrical current)

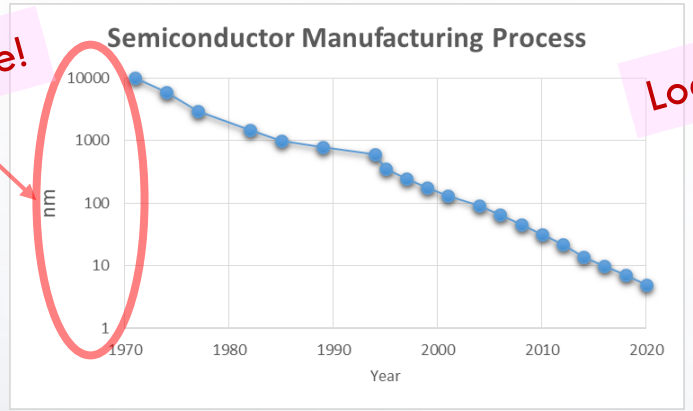
2. **Software**: Vagueness
   - Binary injects **logic errors** into our systems

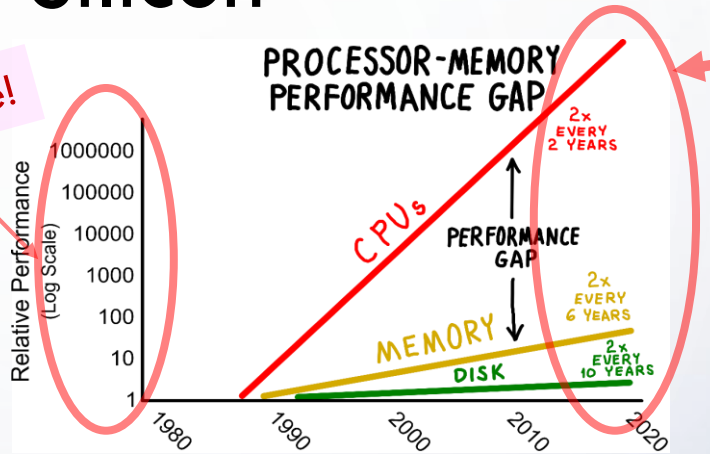
MATHEMATICS ALLOWS FOR NO HYPOCRISY AND NO VAGUENESS.

Stendhal

quotespin.com

Hardware: "The End Of Silicon"





OHZ NOZ!
EXPONENTIAL DIVERGENCE!

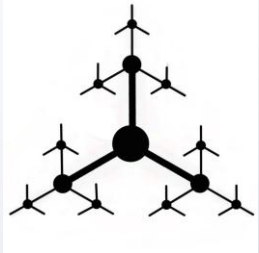
- **Clock rates have peaked** *(We can only generate more heat, not go faster)*
- **Shrinking circuits** reaching **"the end"** *(We are leaking electrons, smaller wires no longer work)*
- We are **bus limited** *(Need more data at the same clock rate!)*
  - 3VL, MVL – **Each wire transfers more data**, higher radix economy (3VL)
- **New Hardware**: **Natively supports MVL**
  - New hardware is **based on quantum effects**
    - *(photon polarization, ions, supercurrents, quantum spin, entanglement)*
  - **Binary is inefficient!**

Radix Economy

- **Radix** – **number of possible values per digit** (e.g., `base10` has radix of `10`)
- **Radix economy** – **how efficiently** numbers are represented

  *radix_economy = (num_digits_needed_to_express_number * radix);*
- **Proven**: Mathematical constant e is **most efficient** (*Euler's number, natural log,* `2.71828`)
- **Ternary is the closest** (*has the highest radix economy of any number system*)
  - Each **trit is worth `1.58496` bits** (*each bit is worth* `0.63` *trits*)
  - Is why ternary is used:
    - ternary search trees for database-search
    - optimizing telephone menu system to minimize the number of menu choices.
- `base2` and `base4` are "next-most-efficient" (*after* `base3`)

3VL: Radix Economy

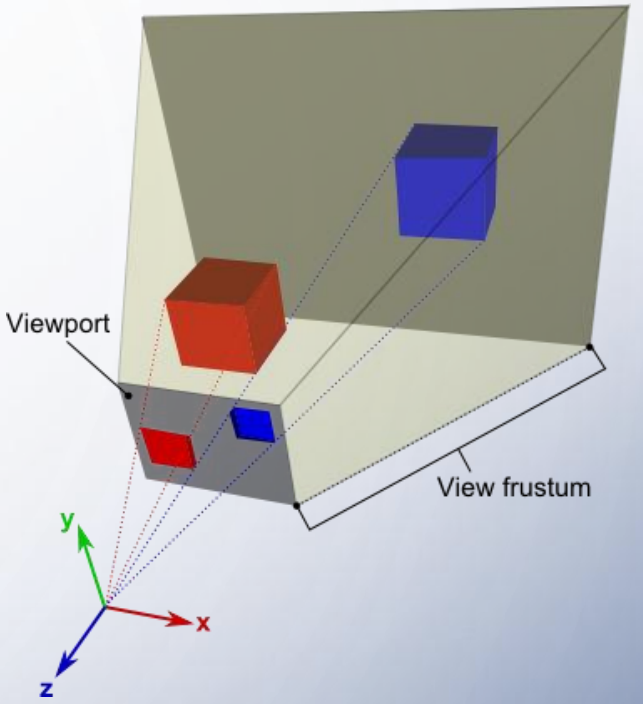


- All modern CPUs suffer the Von Neumann Bottleneck: **The Data Bus**
  - The **shared bus for data and instructions** means **transferring one interferes with the other**
- CPUs are "**von Neumann hardware**" onto which we write "**von Neumann Software**" *(Backus, 1977 Turing Award Lecture)*
- **`Base3` offers significant compactness-of-data representation** for transmission over `Base2`. *High Speed Computing Devices (1950):*

  "A saving of 58 per cent can be gained in going from a binary to a ternary system. A smaller percentage gain is realized in going from a radix 3 to a radix 4 system."
- **Ternary CPUs** would express **efficiencies through**:
  - **Faster data buses** *(at the SAME or REDUCED clock speed!)*
  - **Ternary `opcodes`** *(fewer digits per instruction)* **leaving more "room" for data**
  - **More efficient Ternary CPU ISA**:
    - More efficient operations: Could perform "less-than", "greater-than", "equal-to" in a single instruction
    - More tolerant of errors: Could overclock and more aggressively perform operations that robustly detect/report/re-try operations yielding errors *(because error-path is inherent within MVL values)*

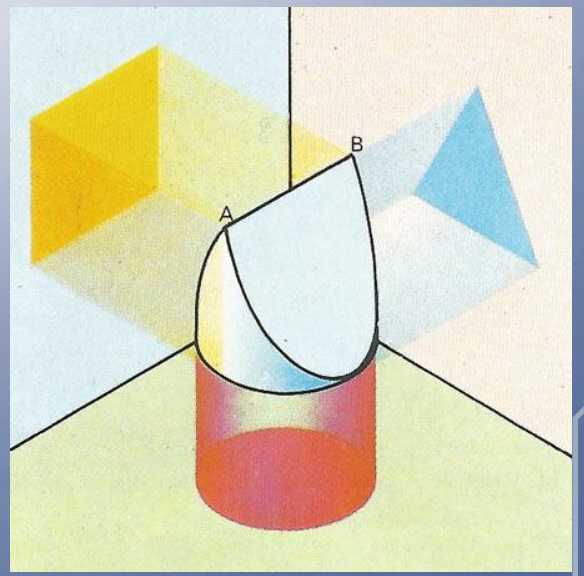
Software: Why Vagueness?

- **Binary logic** is a **mathematical projection** of **N-dimensional logic into 2D space**.
  1. We commonly have 3D *(or ND)* logic.
  2. We "project" that logic into 2D space for system definition, execution.
- We **can "project" logic of any dimension to any other dimension**
  - However, **is "lossy"** *(introduces ambiguities)* **as we project to fewer dimensions**
  - The "real-world" is not binary, so **binary systems have ambiguities**



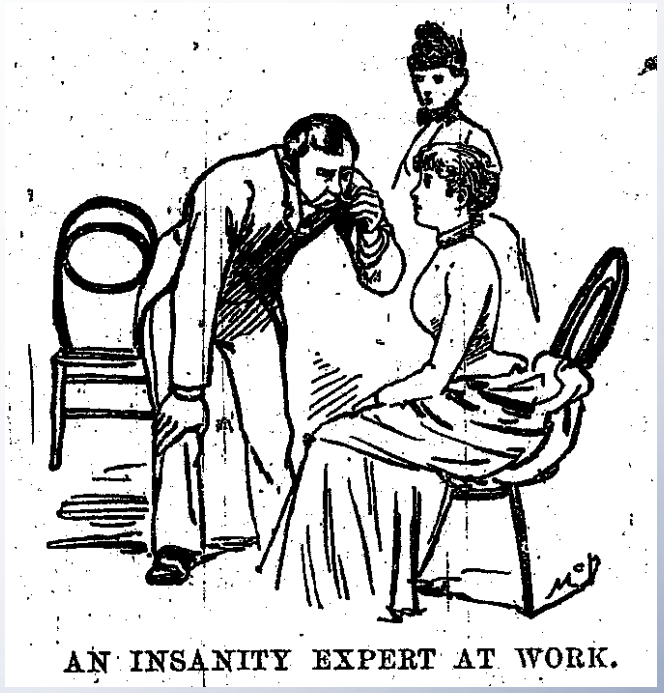
Similar: A **Single-threaded** application **is a projection of multi-threaded logic into sequential space**. *(If you are comfortable with multi-threaded logic, then projecting into single-threaded space is **tedious and error-prone**, even though it can be done.)*

- Why tedious and error-prone?
  - All concurrency *(and parallel)* **issues are handled by you**, **manually**. *(No higher-order reasoning exists such as for concurrency management across threads)*
  - **Complexity increases** to "interleave" concurrent and parallel processing, and resources contention. *(Your system hits the "complexity-wall" earlier)*
- Therefore,
  - **Hard to visualize** what's going on *(everything manually interleaved)*
  - **Hard to fix** phenomenon inherent in your design *(everything "fixed/locked" in the design/implementation)*
  - **Hard to evolve** system *(requires large re-working of the API, internal hard-coded tunings)*



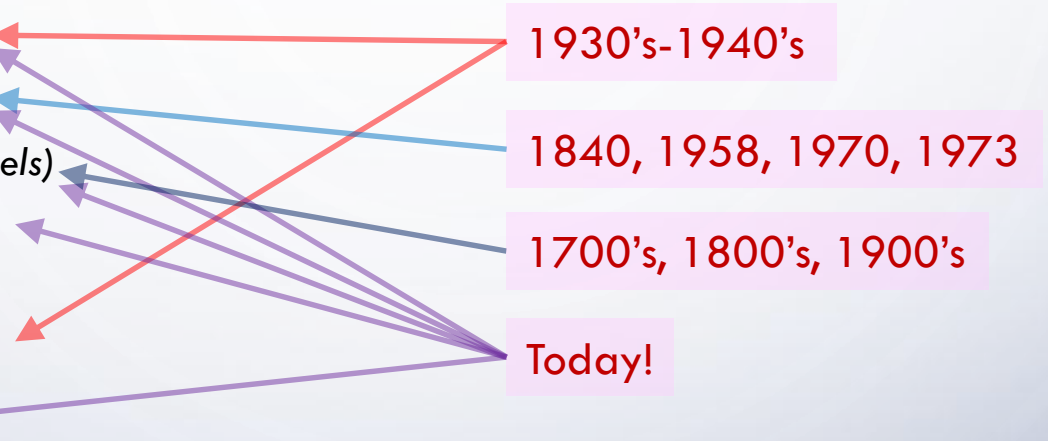
Software: Vagueness Result

- **Binary logic injects errors** into our systems *(even on native binary hardware!)*
  - **Algorithms are incorrect**
  - **Hard to reason** about real-world corner cases
  - **API problems** *(easy to use incorrectly)*
- **Interfacing with native MVL hardware:**
  - **Binary is expensive/inefficient** *(forcing 3-state wire to 2 states)*
  - **Binary is ambiguous** *(what if logic error or hardware issue gives you that third state?)*
- Need to **"re-visit" Software's "First Principles"**
  - **Logic & Reason**
  - **Algorithms & APIs**



What Would a Non-Binary World Look Like?

- **More than 2 logic levels!** *(Wire holds any one of 3+ discrete possible states)*
- What's **been done**?
  - Binary *(2 logic levels)*
  - Ternary *(3 logic levels)*
  - Quaternary *(4 logic levels)*
  - Quinary *(5 logic levels)*
  - …
  - Denary *(10 logic levels)*
  - …many more!



- What's **being done today**? All of them!
  - **Binary** – Commercial CPUs are almost entirely binary, many Logic Families *(for component interface)* are binary
  - **MVL** – Specialized MVL components exist, commoditized in some markets such as telecom *(communications)* and digital electronics *(memory storage, buses)*
    - **Ternary** – "third-state" increases data transfer rates *(telecom)*, allows devices to share buses *(digital electronics)*
    - **Quaternary** – "four-states" means each wire carries two bits, increasing data transfer rates to double throughput at the same clock speed *(telecom)*
    - **Other MVLs** for Exotic Computing…

Summary: Ternary (3VL) Computing History

- 1840: Thomas Fowler **builds balanced ternary calculating machine** *(entirely from wood)*
- 1854: **Boolean algebra**, "An Investigation of the Laws of Thought" *(George Boole's "Book on Logic")*
- 1950: Book, ***"High-speed Computing Devices"*** explores advantages of ternary systems *(by Engineering Research Associates on behalf of US Navy)*
- 1950: Herber Grosch **proposes ternary architecture** for Whirlwind computer project at MIT *(project later decided on binary, implemented control system for military radar network that deployed for 30 years)*
- 1956: **Term "Byte" originates** from phrase, *"by-eight" (Werner Bushholz, design of IBM 7030 Stretch computer, had variable-length byte of 1 to 8 bits)*
- 1958-1965: **Setun ternary computers built** *(designed by Brusentsov and team at Moscow State University)*
- 1960's: Projects explored **ternary logic gates**, **ternary memory cells**, **ternary adders**.
- 1970: **Setun-70**, enhanced version **built** by Brsentsov
- 1973: **TERNAC**, **Fortran-based ternary computing emulator** *(by Frieder and colleagues at State University of New York)*
- 1980's-2010's: **Rise of the Mighty Wintel Empire** which established worldwide the ubiquitous **binary x86 CPU**
- 2010-2015: **MVL commoditization** in **communications and memory technologies**; **specialized MVL CPUs**; **optical computing** and other **new computing strategies** using MVL

Ternary History (Greater Detail)

- 1840: Thomas Fowler builds **balanced ternary calculating machine** *(entirely from wood)*
- 1950: Herber Grosch **proposes ternary architecture** for Whirlwind computer project at MIT *(project later decided on binary, implemen control system for military radar network that deployed for 30 years)*
- 1958: **Setun** Ternary Computer *(Sobolev and Brusentsov, Moscow State University)*
  - **Named after Setun river** running through campus
  - **Goal**: **Inexpensive and reliable** *(Succeeded!)*
    - Lower electricity consumption
    - Lower production cost
    - More reliable (operated in different climatic zones, without requiring support or spare parts)
    - Comparable computers 2.5x cost
  - **18-trit words, 6-trit trytes**, **Fixed and floating-point** operations
  - **50 built** 1958-1965
- 1970: **Setun-70**: Updated auxiliary ternary logic and control instructions; variable operand length (1-3 trytes, results up to 6 trytes);
- 1973: **TERNAC**, Fortran-based ternary computer emulator (State University of New York)
  - 24-trit words, floating-point words had 42 trits for mantissa and 6 trits for exponent
  - Goal: Discover if implementing non-binary on binary computer was feasible. (Succeeded: Speed and price are comparable with that of binary computers)
- 1980: **DSSP**, stack-based ternary programming language, similar to Forth *(Brusentsov, Moscow State University)*; 32-bit version released in 1989.

  "DSSP was not invented. It was found. That is why DSSP has not versions, but only extensions. Forth is created by practice. DSSP is created by theory. But they are similar and it is a fact of great importance." – DSSP & Forth: Compare And Analysis
- 2008: **Ternary Computing Testbed 3-Trit Computer Architecture** *(Jeff Connelly, California Polytechnic State University) [https://github.com/shellreef/trinary]*
- 2009: **Ternary quantum computer proposed** using qutrits rather than qubits *(B.P.Lanyon, University of Queensland, Australia)*

Setun, 1958
Ternary Computer

Ternary computing might be essential for scaling quantum computing: A ternary quantum computer could use as few as 9 ternary gates, while a binary quantum computer would require 50 conventional quantum gates.

-- B.P. Lanyon, University of Queensland, Australia

Ternary *(Today)*

- *Today:* **Ternary telecom signals:**
  - **PAM-3 line codes:** hybrid ternary code, bipolar encoding, MLT-3 encoding *(used in 100BASE-TX Ethernet)*, B3ZS, 4B3T *(used in ISDN basic rate interface)*, 8B6T *(used in 100BASE-T4 Ethernet)*, return-to-zero, SOQPSK-TG *(ternary continuous phase modulation)*
  - **3-PSK:** Sine wave assumes phases of 0°, 120°, 240° relative to a clock pulse
  - **3-FM:** Carrier wave that assumes one-of-three frequencies dependent on three modulation signal conditions
- *Today:* **Digital electronics**
  - Three-state logic (3-state logic)
    - Circuit output port assumes 3 logic levels: `0`, `high-impedance`, `1`
    - Effectively "removes" a device's influence from the rest of the circuit
    - Allows multiple circuits to share the same output line *(such as a bus; essential to operation of shared electronic bus)*
    - Used in many registers, bus drivers, flip-flops, integrated circuits, internal and external buses in microprocessors, computer memory, and peripherals
    - Used to implement efficient multiplexers *(especially for large numbers of inputs)*
  - Open collector input/output
    - Default line is "high" *(line is available)*, device can drive "low" to take the line, then release it when done; Inactive devices are in "third-state" *(not impacting the line)*.
    - *Example:* I^2C bus protocol *(attaching peripherals to computer motherboards, embedded systems)*
    - A typical modern microcontroller has many three-state general purpose input/output pins for this use
  - 3-State Bus typically used between chips on a single PCB *(or sometimes between PCBs sharing a common backplane)*
  - Ternary CAM (content-addressable memory)
    - CAM is used in very high-speed searching applications (associative memory, associative storage, associative array)
    - Ternary CAM allows for third "don't-care" match, adding flexibility to the search.
- *Today:* **Research Venues**
  - IEEE International Symposium on Multiple-Valued Logic (ISMVL) – held annually since 1970
  - Journal of Multiple-Valued Logic and Soft Computing

- **Basic ternary logic circuits** use **a ternary gate** *(or T-Logic gate)*
- **Common application** in modern-day **switching elements**
  - CMOS
  - Resonant tunneling Diodes (RTD) uses quantum mechanical tunneling *(very compact, ultra-high-speed)*
  - Carbon Nano Tubes (CNT) circuits *(flexible electronics)*

Ternary *(Tomorrow)*

- **Optical computing**
  - Single-mode fiber: `0`==dark, two orthogonal light polarizations for `-1`, `1`
  - Multi-mode fiber: Different photon wavelengths for different logic levels
- **Josephson junction** *(phenomenon of supercurrent)* **– proposed as balanced ternary memory cell** *(currents clockwise, counterclockwise, off)*
  - *high-speed, low-power, very simple construction, fewer required elements for ternary operation*
- **Holographic associative memory** *(complex valued artificial neural network)* provides for a mathematical model for ternary "*don't-care*" using complex value representation on content-addressable searches
- **Digital electronics**: Multi-valued memory cells, Fuzzy memory cells, multi-valued circuits, fuzzy logic circuits, infinite-valued logic circuits *(these circuits exist today, but not in commoditized application)*
- **Quantum ternary logic gate** (*qudit* or *qurtit*, *not a qubit)* from photon polarization, atomic spin, or other quantum effects
- **Exotic Computing**: Multiple types of biological and biochemical computing, molecular and DNA computing, holography

How Does Ternary *(or Trinary)* Work?

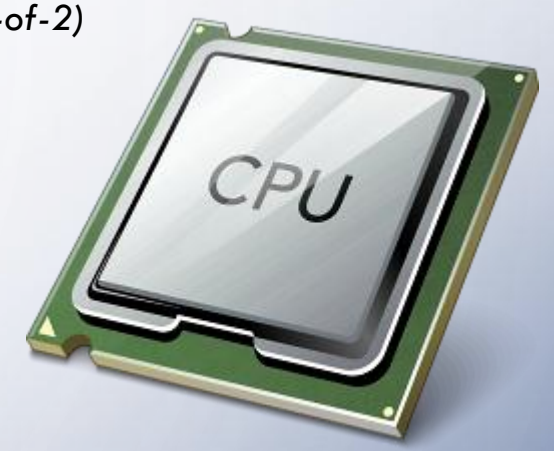
- **3 Logic Levels** *(three-value logic, or 3VL)*
  - Unbalanced Ternary: `(0,1,2)`
  - Balanced Ternary: `(-1,0,1)` or sometimes `(-,0,+)`
- Digit is **ternary trit**, *(not binary bit)*
  - Binary: bits, nibbles, bytes, words
  - Ternary: trits, tribbles, trytes, words
- **In source code**:
  - **trit values** often imply: `false, unknown, true`
  - "`unknown`" could otherwise represent "*maybe*", "*irrelevant/don't-care*", "*unknowable*", "*undecidable*"
- **Number System** *(Compact representation)*
  - **Binary** *(base-2, one-bit-per-digit)* becomes **Ternary** *(base-3, one-trit-per-digit)*
  - **Octal** *(base-8, thee bits-per-digit)* becomes **Nonary** *(base-9, two trits-per-digit)*
  - **Hexadecimal** *(base-16, four bits-per-digit)* becomes **Septemvigesimal** *(base-27, three trits-per-digit)*
  - **Heptavintimal encoding** (base-27): `0123456789ABCDEFGHKMNPRTVXZ`

The CPU Defines The ISA

- **The Instruction Set Architecture (ISA)** is the "programmer's view" of the CPU
- Nearly all CPU **ISAs target the C programming language** *(data types, stack-based, memory models)*
- **CPUs are binary**, so **the ISA is binary**, and **the C programming language is binary**
  - Native type for `bool`, not `trit`
  - *Ubiquitous*: `if/else` binary branching assumed for algorithms
- Migrating to **MVL CPUs**:
  - the C/C++ Programming Language is expected to migrate to new native CPU data types, machine instructions, memory models
  - **Or**, a new MVL language targeting the CPU ISA is required

A Theoretical Ternary (3VL) CPU *(for Today's World)*

- **Base-3 word sizes**, **data types** should be **power-of-3**. *(Base-2 word sizes are almost always power-of-2)*
- **CPU-Native representation**: **Balanced Ternary**
- **Primitive Types** *(native to CPU):*
  - **bit** *(2 values)* becomes **trit** *(3 values)*
  - **4-bit nibble** *(16 values)* becomes **3-trit tribble** *(27 values)*
  - **8-bit byte** *(256 values)* becomes **6-trit tryte** *(729 values, 2 tribbles)* or **9-trit tryte** *(19 683 values, 3 tribbles)*
  - **Balance ternary** has **NO "sign-trit"**, so **Universal number encoding**!
  - **Word/Int** *(Note on value ranges: 16-bits≈10-trits, 32-bits≈20-trits, 64-bits≈40-trits)*
    - **32-bit bus** *(≈4 billion)* becomes one of:
      - **18-trit** *(≈387 million)*
      - **24-trit** *(≈282 billion)*
      - **27-trit** *(≈7.6 trillion, ≈44-bits)*
    - **64-bit bus** *(≈big)* becomes one of:
      - **36-trit** *(≈big)*
      - **45-trit** *(≈reallybig)*
      - **54-trit** *(≈ridiculouslybig)*
  - **Floating-point**: **1-tryte exponent**, followed by **2-tryte mantissa** *(27 trits)*
    - Is more than 8 decimal digits precision *(32-bit float is 5 decimal digits precision)*
    - No "sign-trit" in balanced ternary, so "sign" is integrated with mantissa. *(Binary almost-universally separates "sign-bit" from mantissa)*
    - Heptavintimal ZZZ encodes values that are "not-a-number" or NAN *(as is done in binary IEEE floating point representation)*
- **Machine instructions** – variable length!
  - **Opcodes** – variable length!, 8-bit opcode becomes 5-trit opcode
  - **Operands** – variable length!



Negative Numbers In Ternary

- *Binary*: **Unbalanced around zero**, so we struggle with negative numbers
  - **C/C++ keywords** to "reserve-a-bit" for sign: `signed`, `unsigned`
  - **Code logic errors**, **compile warnings**: mixed expressions of `signed`/`unsigned`
  - **Alternative binary number systems** *(each has its own issues)*,
    - Two's complement, One's complement, Signed magnitude representation, Excess-K *(also called offset binary, or biased representation)*, Base-2, and other systems like zig-zag encoding and signed-digit representation
  - **Native representation ambiguities** exist for some bit-operations
    - C++11 Standard [4.7.2] specifies Two's complement only when converting between `signed`/`unsigned`
    - Some two's-complement CPUs don't have a sign-preserving right-shift that rounds-towards-zero, but merely a round-down right-shift which introduces zeros
    - DSPs commonly use saturating arithmetic where assigning an out-of-range value will clip it at the maximum, and not just drop the high-order bits.
- *(Balanced) Ternary*: **Balanced around zero**
  - **No "sign-trit"** *(same encoding for `signed`/`unsigned`, merely indicates a "base-shift")*
  - **Universal number system** within the CPU! *(A single integral encoding for `signed`/`unsigned`)*

```
trit      my_trit;    //                  -1  0 +1
tribble   my_tribble; //                -13 to +13
tryte     my_tryte;   //             -9,841 to +9,841
trit_word my_word;    // -3,812,798,742,493 to +3,812,798,742,493
```

Advantages of Universal Number System

- **Universal number system** within the CPU means **the same native-representation is used** for both `signed` and `unsigned` values.
  - **Consistency**: **CPU operations are NOT DIFFERENT for** `signed, unsigned` types *(same representation!)*
  - **CPU opcodes needed only for one Universal number system**! *(CPU transistor budget made available for other operations or optimizations)*
  - **Code logic errors**, **compile warnings**: Some no longer relevant *(are no longer error-corner-cases)!*
- **Simpler:**
  - Variable length operands
  - Singular shift operation
  - Tri-value of a function's "sign-of-a-number"
  - Optimal rounding by cutting off lower digits
  - Compensation of errors-rounding in the process of calculating
- **All ternary numbers have 3's complements** because the number range is entirely symmetrical around zero

Ternary: Some Things Different, Easier, Harder

- **Some things are "just different"** between binary and ternary
  - *Easier*: Can discretely represent `1/3` *(0.333-repeating)*
  - *Harder*: Cannot discretely represent `1/2` *(0.5)*
- **Some operations more efficient, (much) easier** *(than binary):*
  - Universal number system – no "sign-bit/trit"
  - Negation is trivial *(simple inversion)*
  - Addition table has only two symmetric carries *(in balanced ternary)*
  - Subtraction is simple: Negation, then add *(CPU uses same add for add/subtract!)*
  - Multiplication is simple *(one-digit multiplication has no carries in balanced ternary; the plus-minus consistency reduces the carry rate in multi-digit multiplication)*
  - `int=>float` is simple *(append mantissa)*
  - Truncation and rounding are the same operation in balanced ternary *(they produce exactly the same result, which is not possible in binary)*
  - The rounding-truncation equivalence in ternary reduces the carry rate in rounding on fractions
- **Some things harder** *(than binary):*
  - **Previously simple Bivalent logic can become ambiguous** *(when the wire natively holds 3 states)*

Knuth on Ternary

- Knuth suggests **balanced ternary is the "most elegant"** *(best?)* number system:

  D.E. Knuth, The Art of Computer Programming – Volume 2: Seminumerical Algorithms, pp.190-192. Addison-Wesley, 2nd ed., 1980. ISBN 0-201-03822-6.

- Knuth points out that ternary allows **numbers to be stored in about 36% less space**

- Knuth predicts ternary logic's **elegance and efficiency will bring ternary back** into development in the future.

"Perhaps the prettiest number system of all is the balanced ternary notation." *– Donald E. Knuth*

We Don't Need To Wait For Ternary CPUs

- We are C/C++, **the target language for all CPU ISAs**
  - Should probably **become familiar with MVL**, since we **will be charged with exploiting new MVL hardware**
- **Ternary logic can execute on binary CPU** *(just as binary logic can execute on ternary CPU)*
- Can **build our own Ternary CPU emulator** *(has been done, many successes)*
- Can **adapt our thinking** to ternary logic
  - Algorithms, Libraries, Code constructs
  - C++ Is "King" of static-type-value-semantics!

*(**Perfect place to play!** MVL algorithms, hardware)*

*The **Important Thing**™ is to*

know how to think about the problem…


How to think about this ternary thing?

What is Logic?

- **Logic:** Use and study of ***valid* reasoning**.
  - **Many types**, depending on "**rules**" and "**notation**" and "**basis-for-reasoning**":
    - *Formal* logic, *mathematical* logic, predicate logic, *modal* logic, *computational* logic …
  - If your reasoning is NOT VALID, then it is NOT LOGICAL.
- **Computational logic** based on work by Alan Turing *(1912-1954)* and then Kurt Gödel *(1906-1978)*
  - **Church-Turing Thesis** *(1937)*: values can be computed *(e.g., a "Turing-machine" can compute values that a human could otherwise compute)*
  - **Lambda Calculus (λ-calculus)** *(1937)*: universal model of computation where values can be *(recursively)* substituted; useful in functional thinking *(basis of functional programming)* and iterative reduction.
- Turing's work became the basis for computer machinery of the 1940's
- Mathematical notation formalized in the 1950's and 1960's

What is Binary?

- **Binary mandates "two states"** (e.g., `true` or `false`)
- Basis for **Boolean algebra** (all variables `true` or `false`, `&&`, `||`, `!`)
  - *The Mathematical Analysis of Logic (1847)*, George Boole *(1815-1864)*
- Supported by **all modern programming languages**
- Fundamental to ***(development and evolution of)* digital electronics**
- Used in **set-theory, statistics**

What is Valence (Valent)?

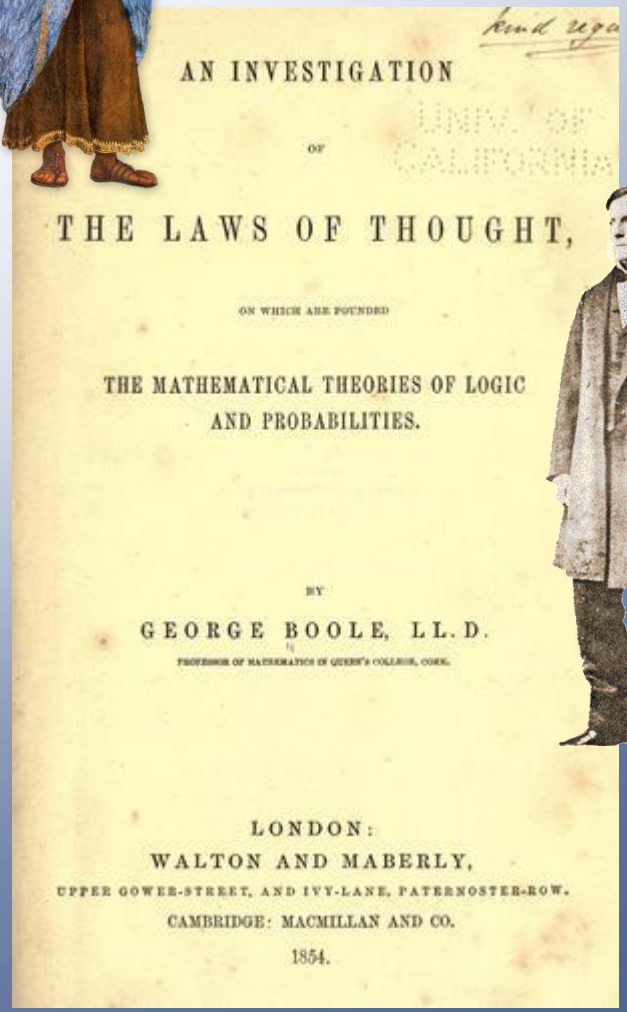
- **Valence**: "**capacity**" (from Latin "*velentia*" for "power" or "strength")
  - **Univalent**: "One form"
  - **Bivalent**: "In pairs"
  - **Multivalent** ***(Polyvalent)***: "Many forms/meanings/aspects"

So…

- **Bivalent Logic** is binary logic
  - **2VL**: Radix is 2
- **Multi-Valent Logic** is MVL *(3 or more states)*
  - **MVL**: Radix is 3 or more
  - *Same*: Multi-Valent Logic, Multiple-Valued Logic, Multi-Valued Logic, Many Valued Logic

Law Of Thought

- **Law of Thought: "The Three Traditional Laws"**
  (began with Plato, then Aristotle, 428-322 BC)
  - **(ID) Law of Identity**
  - **(NC) Law of *(Non-)*Contradiction**
  - **(EM) Law of Excluded Middle**
- Laws by which **valid thought proceeds, justifies all valid deduction**
- **Basis for Boole's work** *(Bivalent logic)*
- ***"The Laws of Thought"*** *(otherwise called,* ***"Boole's book on logic"****)*
  - *"An Investigation of the Laws of Thought On Which are Founded the Mathematical Theories of Logic and Probabilities", George Boole, 1854*



Law Of Thought: (ID) "Law of Identity"

- **(ID) "Law of Identity" –** everything is identical to itself
  - *Boole:* **"EQUIVOLENCE"** – Every class includes itself
- *Programmer's Take:* **Practical and non-ambiguous** *(for discrete values!)*
- Example:
  - "The value is the value."

```
bool b1 = true;
assert(b1 == true);
assert(b1 == b1);
```

- Issue: **Non-discrete values** *(e.g., floating-point approximations)*
  - **Different EPSILON for different floating-types** (e.g., `FLT_EPSILON`, `DBL_EPSILON`)
  - **Different EPSILON for values with different "ranges" or "units"** *(e.g., comparing values* `[0..10000]` or `[0..1]`*)*
  - **Do not assume** *(for floating-point):*
    - if `(a==b)` and `(b==c)` then `(a==c)`

```
float f1 = 42;
assert(f1 == 42); // unreliable
assert(f1 == f1); // unreliable

assert(fabs(f1 - 42) < FLT_EPSILON); // better
assert(fabs(f1 - f1) < FLT_EPSILON); // better
```

Sometimes works?

Tricky logic fix!

Law Of Thought: (NC) "Law of (Non-)Contradiction"

- **(NC) "Law of (Non-)Contradiction" –** no thing is the negative of that thing.
  (Otherwise stated: "A statement cannot be both `true` and `false` at the same time.")
  - *Boole:* **"AND" –** The product of a class and its complement is the NULL class

```
bool b1 = true;
assert(b1 != (!b1));
```

```
if(b1 && (!b1)) {
  // ...never here
}
```

- *Programmer's Take:* Becomes a **Logical fallacy** when **applying "negation" does NOT yield the "opposite"**

```
bool is_using_file = ...;
if(!is_using_file) {
  // ...we can delete the file,
  // we are not using it
}
```

What if someone else is using it?
What if we need it later?

```
Record r1(City("Chicago"));
Record r2(City(""));

bool is_same_city = (r1.city() == r2.city());
if(!is_same_city) {
  // ...must be different-cities
}
```

Might be in the same city,
We don't know!

- **Error in code** is one of:

A. **Semantic misunderstanding** *(you did not understand what the value means, or how to reason with it)*

B. **Value initialization is a logical failure** *(e.g., an "unknown" was forced to* `true` *or* `false`*)*

Law Of Thought: (EM) "Law of Excluded Middle"

- **(EM) "Law of Excluded Middle" –** everything has a value or the negative of that value
  - *Boole:* **"OR" –** The sum of a class and its complement is the universal class

```
bool b1 = true;
assert(b1 != (!b1));
```

```
if(b1 || (!b1)) {
  // ...always here
}
```

- *Programmer's Take:* Becomes a **Logical fallacy** when **neither `true` nor `false` is correct**
- **Big Killer** of programs! *(Leads to **unhandled real-world corner cases**!)*

```
bool is_red = ...;  // ??
```

```
if(is_red) {
  // ...entirely red?
  // ...mostly red?
  // ...a little-bit red?
} else {
  // ...not (mostly) red?
  // ...no red at all?
}
```

Ambiguous!

Ambiguous!

- *Example:* "The apple is red", "The apple is not red." *(...it might be 50% red)*

```
bool is_partially_red = ...; // Logically safer!
```

```
if(is_partially_red) {
  // ...ok, unambiguous
} else {
  // ...ok, no "red" at all!
}
```

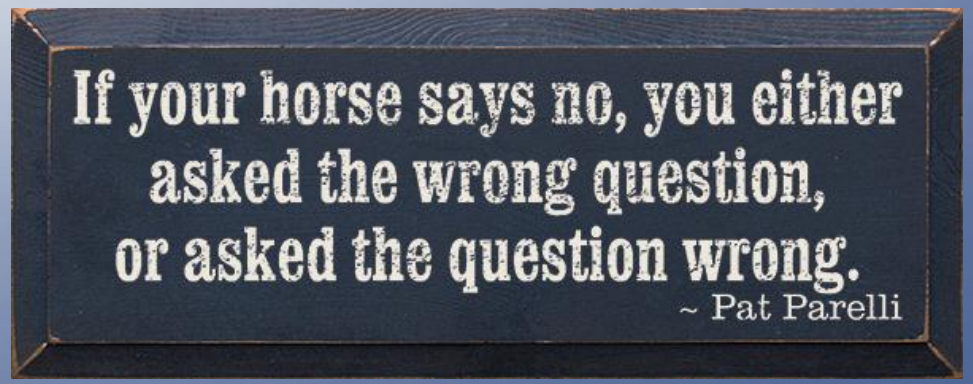
Better!

Better!

- *Better, but is it still useful within our intended algorithm?*

Law Of Thought: Summary of "Programmer's Take"

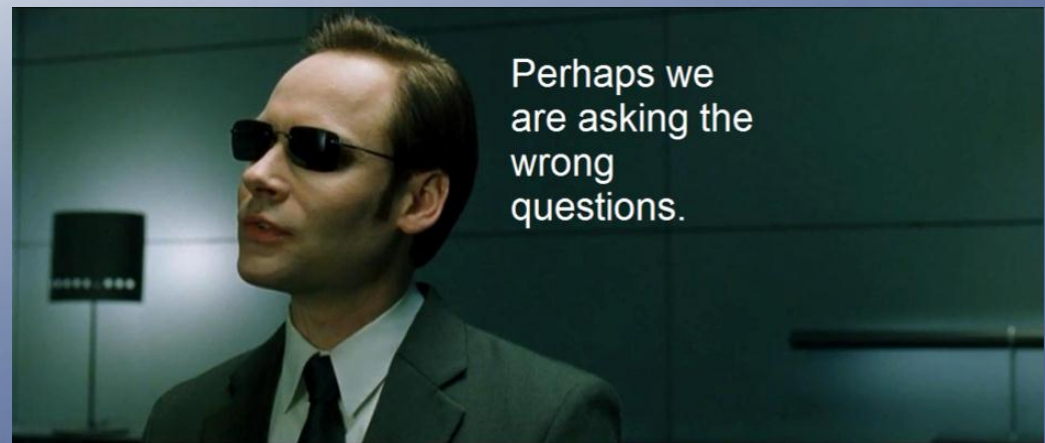
- Summary of, *"Programmer's Take"*:
  - (ID) "**Law of Identity**" – **Practical** and **non-ambiguous**, but **issues with non-discrete** *(e.g., floating-point)*
  - (NC) "**Law Of Non-Contradiction**" – Logical **fallacy when "negating" does NOT yield the opposite**
  - (EM) "**Law Of The Excluded Middle**" – Logical **fallacy when neither `true` nor `false` is correct**
- Failures are ***(ALWAYS!)*** **Semantic problem** because you asked the **wrong question!**
  - **Misunderstanding** for **what the value means**
  - **Faulty reasoning** for **how the value can be used** *(algorithm requires different value, or different question)*
- Examples:
  - "The apple is red." *(…it might be 50% red)*
  - "You are with us, or against us." *(…you might not take a position)*
  - "Male or Female." *(…sexual dimorphism, gender reassignment, transgender …Facebook offers 56 categories!)*



Wrong Question Asked All The Time!

- We **frequently ask the wrong question**, because our CPUs are binary.
- Programmers *Love* `bool`
  - Because it provides **the illusion of simplicity** *(for a complicated world)*
  - Makes **reasoning/algorithms simple** *(but not trivial)*
- **`bool` gets you into trouble** when you later find ambiguous corner-cases
  - **Expensive to move APIs from `bool`**
  - **Hard to argue in favor of an API change** for, "that one corner case" *(is never "just one" corner case, and even if it is, that one corner-case shows up in application-specific code in thousands (millions?) of application-specific scenarios)*

"The Matrix", 1999



The "Real World" Is Not Binary

- **Nature is not binary**. **Virtually no process in the "real-world" is binary**, including mathematics *(e.g., Principle of Excluded Middle)*.
- **Humans DO NOT** think in binary
  - Are comfortable with the idea of "`unknown`", because is consistent with what they observe to be `true`.
  - Understand "false dilemmas" *(fallacy of false choice)*, do not assume negation yields the opposite.
- **Programmers DO** think in binary
  - Programmers are conditioned for binary, thinking in languages that have difficulty expressing MVL.
  - After formal training and using binary programming languages, **programmers are NOT great** at handling `unknown`.
- We code our systems by **forcing real-world scenarios into binary choices**
  - **Often "too-coarse"** for elegant data structures, algorithms
  - **Often injects logical errors and *(unfortunate)* assumptions** into system
- **Root Cause, Source of Logical System Failure:**
  1. ***(Biggest):*** Inability to handle/respect "`unknown`"
  2. *(next):* Inability to handle multiple discrete "natural-states" beyond 2

Logic Options For Programmers

- **Univalent Logic:** *(Codes that use only `0`'s, or `1`'s, but not both)*
  - Not practical
  - Hard to make conditional *(branching)* statements
  - No known successes *(although "Old Timers" occasionally reference "The Good Ole' Days When…")*
- **Bivalent Logic:** *(Codes that use `0`'s and `1`'s)*
  - Ubiquitous *(all programming languages support 2-way branching)*
  - Much formal training for programming students/professionals
  - Many reference-algorithms, data structures, and published *(peer-reviewed)* approaches
- **Multi-Valued Logic** (or **Polyvalent Logic**): *(Codes that use `[0..n]` values)*
  - Strong basis in "Set-Theory" *(of which Bivalent Logic is a subset)*
  - Increasingly embraced on modern hardware *(performance, scalability, new features)*
  - Allows programmers to reason closer to problem domain, handle real-world corner cases, create better APIs

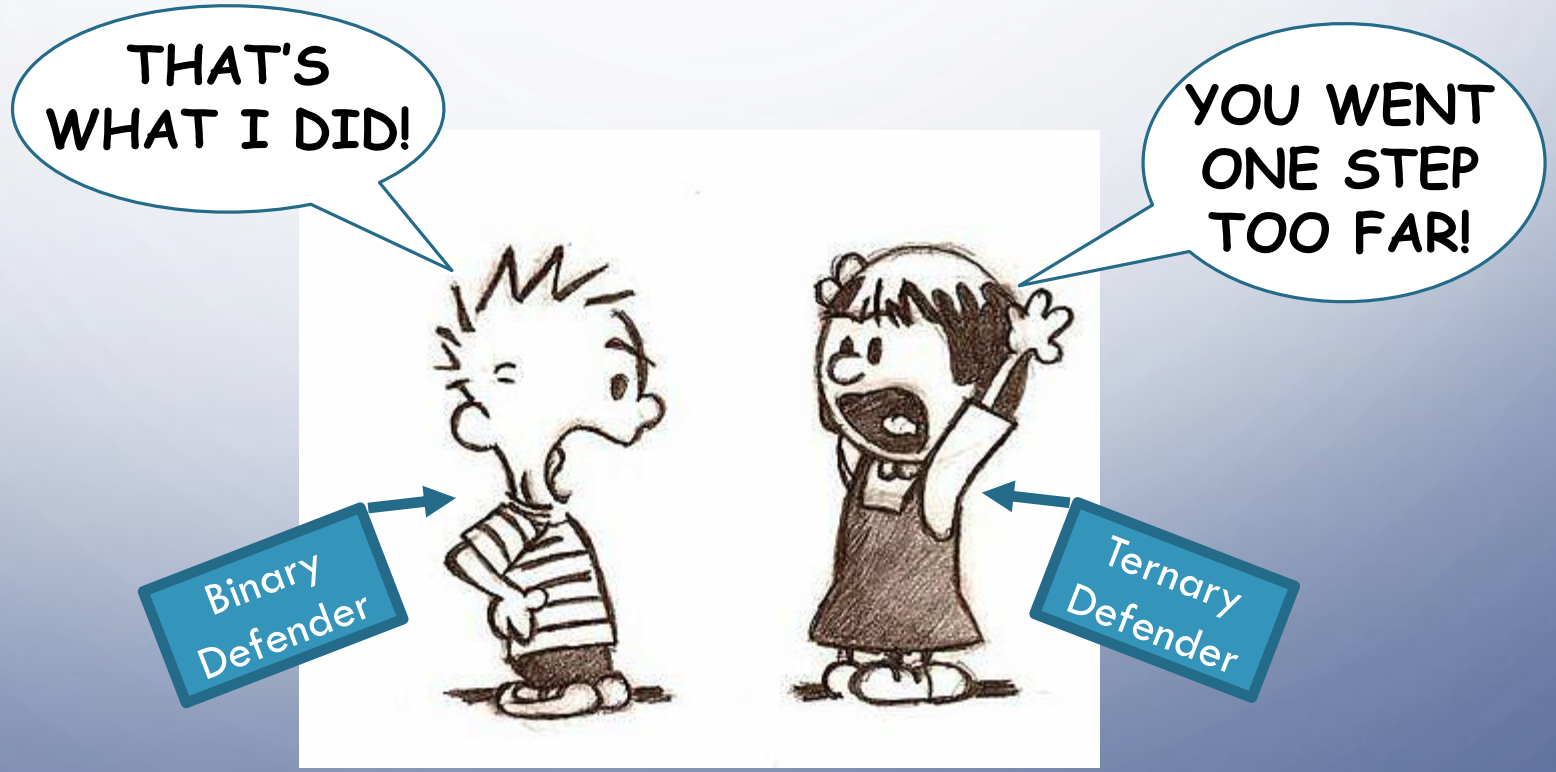
Summary Of Choices For Programmers

- **Only Two Options!** *(A Boolean choice!)*
- Programming languages, algorithms, logic, and reasoning **must be one of**:

	Bivalent Logic	Multi-Valued Logic
How	0's and 1's	[0…n]
Theory	Boolean Algebra	Set-Theory

- Note: **Boolean algebra** is a **special-case of set-theory**!
  - *Proved:* ***Every Boolean algebra is isomorphic to a field of sets****; Stone's representation theorem for Boolean algebras, M.H. Stone, 1936*
- **"Ternary logic"** is the **first multi-valued logic**
  - Then "quaternary logic", etc.

"Make everything as simple as possible, and not simpler." – *Albert Einstein*



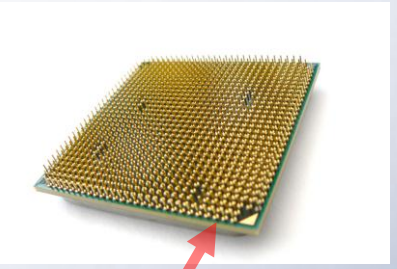
Our Universe

- **Our Universe has "Bad Spock"** – `bool` is our fundamental building block
- Many **parallel Universes have "Good Spock"** – `trit` is our fundamental building block *(`bool` does not even exist!)*
- **"Other Universe" Implications** *(in addition to having "Good Spock"):*
  - No such thing as bits! *(They use `trits`!)*
  - No such thing as `bool`, except some languages implement an uncommon *(and expensive!)* forced runtime conversion from `trit`
  - *(Almost)* All logic is ternary, or MVL *(not binary)!*
- **Binary is discouraged in ternary systems**, because introduces ambiguities *(errors!)*
  - Wire natively supports more than 2 states, so:
    - **Is inefficient** *(expensive)* to force to 2 states;
    - **Is ambiguous** when unexpected state occurs on wire *(e.g., when "third-state" shows up from logic error or hardware issue)*

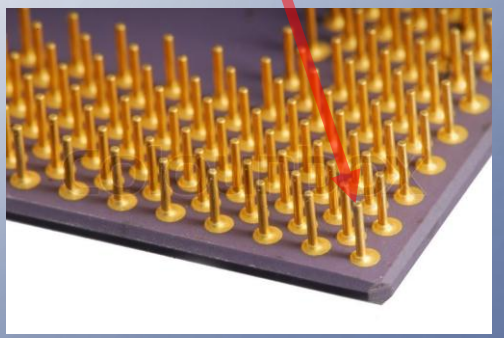
"Star Trek", Mirror Mirror, 1967

What's A Bit?

Binary wire



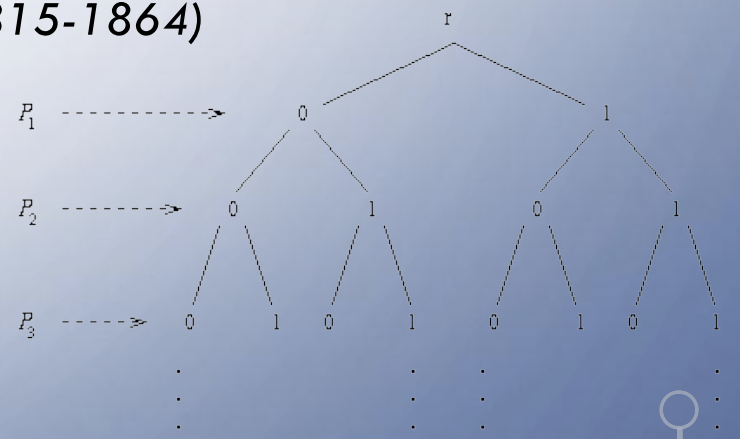
CPU pins ("wires")



- Is **One (binary) "Wire"**
- **Value is "Boolean"** – `bool` (is `true` or `false`)
- Bit is **NOT Trivial** *(in hardware)!*
  - **Logically**: "Above" or "Below" a *(Voltage)* level *(to be `true` or `false`)*
  - **Reality**: Within-recognized-band for `true`, other band for `false`
- **Is the simplest thing** we can manage for a "value" **in binary hardware**
  - **Not trivial** because but must account for phase, noise fluctuations/ambiguities
- **On ternary hardware**, **simplest thing would be `trit`** *(not bit)*
  - Would be **expensive** to convert **`trit` to bit** *(take extra circuitry, require resolution of ambiguities)*
- **On MVL hardware, simplest thing would be value within radix-range**

What's A `bool`?

- Binary Programmers: `bool` **is most "primitive" type**!
  - **Building block for algorithms** *(Bivalent logic)*
  - **Building block for ALL higher-order types** (`int`, pointer, `float`, `strings`, `structs`...)
- C/C++ keyword: **`bool`**
  - Should be keyword `Bool` *(capitalized proper noun, George Boole 1815-1864)*
  - Could be keyword `Boole` *(proper spelling)*
  - Could otherwise have been keyword `bit` *(fewer letters, no capitals)*



- Programmers **view as "trivial"** type
- **Value** is one of "**keyword literal**": `true`, `false`
- **Easy to reason about!** *(BRANCH on `true/false`!)*
- **All** programming languages **ASSUME and PROVIDE for bivalent branching**, because that's how our Binary CPUs work!

Principle of Bivalent Logic:

*All propositions have **exactly one truth value**, which is either* `true` **or** `false`.

Bivalent Ambiguities *(Vagueness!)*

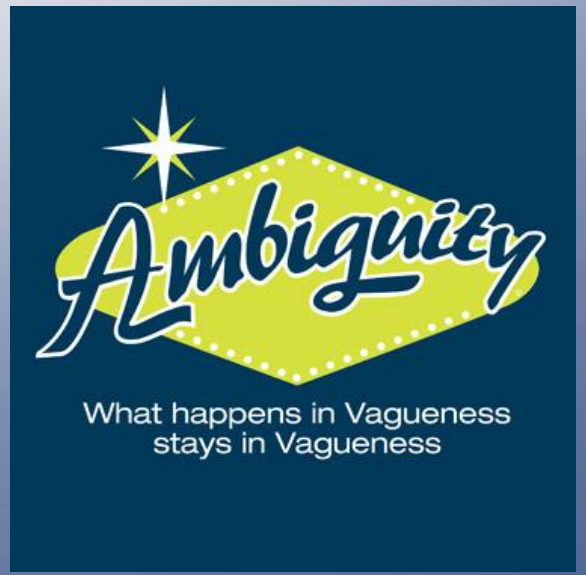
- **Principle of Bivalent Logic** otherwise stated:

  "All declarative sentences can be answered by `true` **or** `false`*."*

- **If that's not true** for your *(binary)* design**, you are in trouble**.
- Programmer's Job: **Specify detail**.
  - **Design**: Specifies **what is disallowed** *(what can be ASSUMED)*
  - **Implementation**: Specifies **how something works** *(also called, "Detailed Design")*
- Your design/implementation:
  - **Allows or disallows something**
  - **Is NEVER vague!** *(You always know behavior)*

If you do not know behavior, your design is incomplete

*(i.e., "**Vague**": unhandled case/scenario)*



Examples Of Vagueness

- **Ambiguity**: Anytime **"three-or-more-states" are FORCED into "two-states"** (`bool`).
- **HARDER** to write *(good)* **APIs using `bool`** in the following scenarios:
  - Device or network connection: State is (1) on, (2) off, (3) idle.
  - Hardware pin: Is (1) "lo", (2) "hi", or (3) "unconfigured" *(we are not using it)*.
  - Data value: Is (1) missing *(never updated)*, (2) valid, (3) stale
  - Diagnostic result, or subsystem configuration: Is (1) good, (2) bad, (3) unknown
  - Numerical computation: Results in (1) known: zero, (2) known: non-zero, or (3) unknown *(e.g., divide-by-zero)*.
- Our **"thinking in binary" forces us to inject errors** into our systems *(because our API was insufficiently expressive)*.
- **Ambiguities can be resolved**
  - Requires **policies *(intended behavior is decided)***, **extra documentation**, **extra code/logic**
  - **Frequently requires extra variables** *(such as a "companion-`bool`" variable or data member to explicitly track "is-known" or "is-valid" or "was-set")*.

How Do We Resolve Ambiguity *(Today)*? `throw`

- **Best sign of a bad API**: *No Way Out*, so `throw`
- **Why Exceptions**: To Handle "*Exceptional Scenarios*"
  - A "normal-error-case" is NOT exceptional *(is EXPECTED!)*
- A **function** *(operation)* **finishes in one of**:
  1. **Success** – operation completes productively *(with good-or-bad result)*
  2. **Failure** – operation completes with "normal-and-expected" error
  3. **Exception** – operation does not complete (`throw`, *stack-unwind*)
- **Bad API**: **No path exists for normal-and-expected errors**.
  - For example, error-path is needed when:
    - User called function improperly.
    - Module is currently "offline" or "unconfigured"
    - Needed resource is currently unavailable *(especially for asynchronous/non-blocking APIs)*
    - Operation cannot complete successfully *(such as when user asked to load a file of the wrong format)*
- **Using `throw` to handle "normal-error-cases"** merely **leverages "exception-handling" for your API**
  - **You "cheat"** by making your synchronous API *partially* synchronous: Sequential in the optimistic-path, unwinds program in the error-case path.
  - Why is this bad?
    - **Exception handling now unavailable for "real" exceptions** (CANNOT `throw` *when unwinding an exception; can* `catch`, *and re-*`throw`)
    - **Your error path is hard to understand and use** *(is non-obvious and POORLY documented!)*
- **Avoid `bool` in APIs** because it often requires the implementer to `throw` *(because no path exists to communicate failure scenarios)*

NOT the same thing!

"Yes, I successfully talked to the module, and things are NOT good."

"I can't reach the module. No response."

If your **API cannot communicate** with the caller, then you **must `throw`**!

A Parallel Universe: NO `bool`

Pretend we are in this Universe!

- **If we did not have `bool`**, what would we use? *First example:* **Use `enum`**
- **Never use `bool` in our APIs!** *(Never pass-as-parameter, never as a return-type)*
  - `enum` **is better documentation** *(and unambiguous!)*
  - `enum` **allows API to safely evolve** *(because new values can "show up later" or "change/evolve" to handle real-world scenarios)*
  - `enum` is **type-safe** *(a Really Big Deal™)*
  - **C++11 `enum`** can be **strongly typed** *(but C++98 `enum`-type-safety is still way better than `bool`!)*
  - **C++11 `enum`** can be **forward declared** *(for more flexible/powerful module definition and integration)*

```
Instrument inst;
... = inst.findParameter("Laser2", true/*case_sensitive*/);
... = inst.findParameter("Laser3", Case::CASE_SENSITIVE);
```

Which would you rather see?

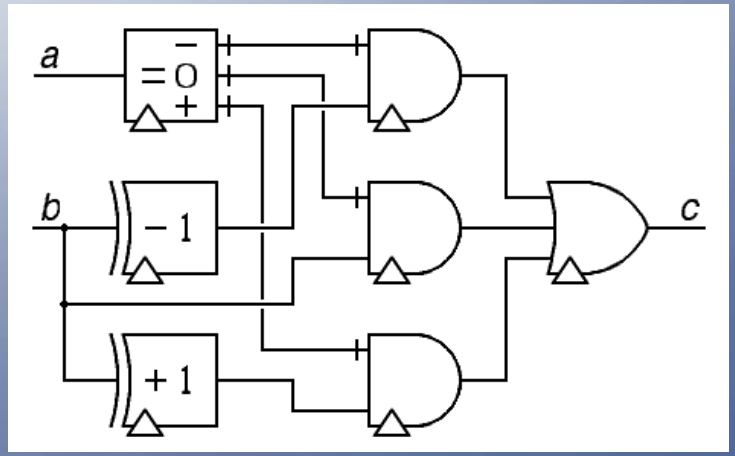
```
... = my_board.setBit(0, true/*true==hi*/);
... = my_board.setBit(1, Board::BIT_HI);
```

Which would you rather see?

- **Side effect**: If you do not use `bool`, **your algorithms start going MVL** *(not binary!)*

Many-Valued Logics

- Issues with Bivalent logic gave rise to **Many-Valued Logics** in the 1960's, Based in **set-theory**
- **Addresses ambiguities** by establishing a "third-state" (*unknown*), or additional discrete states for reasoning
- MVL **Increasingly used** in new hardware
  - **each wire holds/transfers more data at same clock rate**
  - **Additional state enables specialized processing** *(error-correction, bus protocols)*
- Practical **real-world applications deployed** using Three-Value-Logic *(3VL)*
  - communications, switching systems, memory
- **Today: Mature MVL Circuitry Understanding**
  - MVL transistors, gates
  - MVL registers
  - MVL data multiplexing, processing

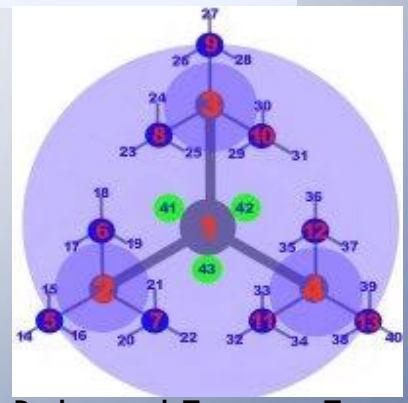


The Rise Of MVL

- **Clock Rates Have Stalled** – *(physics power density limits)*, Need to transfer more data on same wires, at same clock rate.
  - **Quaternary line codes** are used for transmission, since the telegraph *(1700's)* to today *(2B1Q code used in modern ISDN circuits, since 1995)*
- **New hardware:**
  - Photon-based *(multi-mode optics, many different-colored photons "on the same wire")*
  - Ion-based *(Oxygen ions to store data, HP's "The Machine" memristors)*
  - Other quantum-effects-based *(electron spin, other quantum effects)*
- **Examples:**
  - **Current-based electronics** – today's electronics are Voltage-based (value is assumed to be difference from some "ground"), alternative is based on "current-flow" *(value is current-direction, or no-current: e.g., current flows **"left/right/off")*.**
  - **Optics/Photonics** – different colored photons can "share" the same "wire" *(e.g., multi-mode fiber optics have several photon signals on the same wire, at the "same" or "different" frequency, they pass through each other)*
  - **Genetics** – DNA has four nucleotides (`A`, `C`, `G`, `T`) which represents quaternary (base-4); The phenomenon of "base pairs" *(nucleobases)* of A⇔T and C⇔G are represented as the "complement" of [0,3] and [1,2].
  - **Memory Density** – addressing more memory takes longer wires, which slows clock rate. More logic levels on fewer wires is more efficient, as is new developments in 3D circuits, holographic storage, other novel materials and memory approaches
  - **Interfacing With Novel Computing Technology** – Specialized processors, chemical computing, biological computing, quantum computing

Code Looks Different!

- In ternary system, **`bool` becomes uncommon** in code/algorithms
  - `bool` becomes expensive, and *(sometimes)* ambiguous *(because the wire has 3+ states)*
- **Programmers become comfortable** with **3-way branch**, or **multi-way branch**
- Binary **approaches/algorithms are replaced** with ternary approaches/algorithms. *Example:*
  - B-tree data structures becomes Ternary-tree data structures!
  - Binary navigation becomes ternary navigation!
- **Programmer** now **thinks in MVL!**
  - Data structure design
  - Data object navigation
  - Algorithms/processing *(handling "third-state" or "Nth-state")*



Balanced Ternary Tree *(at 43rd Step)*

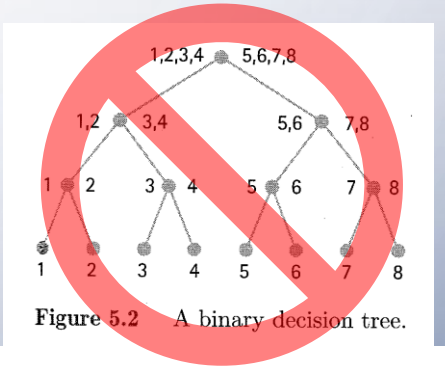


Figure 5.2 A binary decision tree.

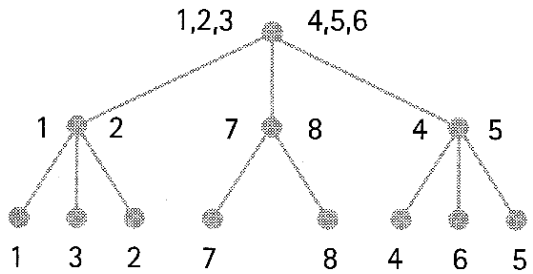


Figure 5.3 An optimal decision tree.

Example Different (MVL) Code: `boost::logic::tribool`

- `boost::logic::tribool` provides, "**3-state Boolean logic**" *(from Boost docs)*
  - Values: `true`, `false`, `indeterminate` *("third-state" can be renamed)*

```
tribool b = some_operation();
if (b) {
  // b is true
}
else if (!b) {
  // b is false
}
else {
  // b is indeterminate
}
```

```
tribool x = some_op();
tribool y = some_other_op();
if (x && y) {
  // both x and y are true
}
else if (!(x && y)) {
  // either x or y is false
}
else {
  // neither x nor y is false, but we don't know that both are true

  if (x || y) {
    // either x or y is true
  }
}
```

- **Notes:**
  - **In bool context**, "`indeterminate`" converts to "`false`"
  - **Explicit check**: `indeterminate()`

Explicit *(unambiguous)* check for value `indeterminate`

```
tribool x = try_to_do_something_tricky();
if (indeterminate(x)) {
  // value of x is indeterminate
}
else {
  // report success or failure of x
}
```

3VL on 2VL Programming Language: Tricky

- **Proper**: Explicit check

Explicit *(unambiguous)* check for value `indeterminate`

```
tribool x = try_to_do_something_tricky();
if (indeterminate(x)) {
  // value of x is indeterminate
}
else {
  // report success or failure of x
}
```

- **Improper**: Implicit conversion to `bool`

```
// IMPROPER!  "indeterminate" converts to "false"!
tribool x = try_to_do_something_tricky();
if (false == x) {
  // x is false, or indeterminate
}
else {
  // x is true
}
```

```
// IMPROPER!  "indeterminate" converts to "false"!
tribool x = try_to_do_something_tricky();
if (x) {
  // x is true
}
else {
  // x is false, or indeterminate
}
```

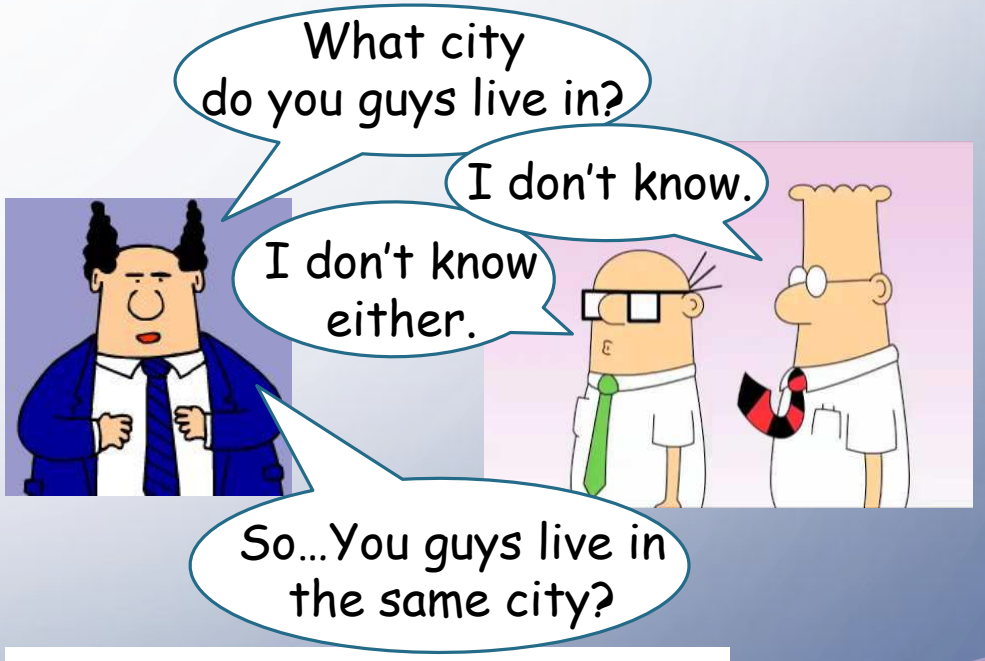
- **Issues**:
  - **Conversion to `bool` is silent!** *(Relational operators by default are `bool`!)*
  - Ambiguous:
    - **Conversion-to-`bool` is ALWAYS be ambiguous**! *(Is always lossy!)*
    - **Comparing values**, or **Identity-checking values**? *(Very Big Difference!)*

2VL: Comparing Values

- **Comparing Values**: **Rank relative** to each other *(less-than, equal, greater-than)*
- **"Empty"** – ranked **at same level** as **"Stateful"** *(no "unknown" option exists)*


```
// Result SHOULD be "unknown"
if(City("Chicago") == City("")) {
  // ...same city...
} else {
  // ...different city...
}
```

NO! NEITHER IS CORRECT! ANSWER SHOULD BE "unknown"!




```
// Result SHOULD be "unknown"
if(City("") == City("")) {
  // ...same city...
} else {
  // ...different city...
}
```

NO! NEITHER IS CORRECT! ANSWER SHOULD BE "unknown"!

How To Compare *(Rank)*?

- **Application-specific types** may define an interface supporting any-or-all of the following **attributes**:
  - **Unknown/Known**: The instance is "indeterminate" or "recognized"
  - **Empty/Stateful**: The instance has "no-state" or is at least "partially-populated"
  - **Valid/Invalid**: The instance has full integrity as expected by the type, or not
- **Mixing-and-matching** *(depending on application-specific type)*:
  - An "empty" instance may be valid-or-invalid
  - A "stateful" instance may be valid-or-invalid *(e.g., partially populated might be "invalid")*
  - An "`unknown`" instance is probably not valid, uncommon to be "stateful"
- In the **maximal scenario**, this implies the following ranking *(low-to-high)*:
  1. **`unknown`** – probably "empty", state is "indeterminate" or "unknown" or "cannot-be-known"
  2. **Empty, Not-Valid**
  3. **Empty, Valid**
  4. **Stateful, Not-Valid**
  5. **Stateful, Valid**

Probably one-or-neither *(not both)*

Optional *(is type-specific)*

Always

- **Ranking within categories is *(usually)* expected** *(such as the default expected behavior for ranking all "valid" instances relative to each other)*
- Application-specific type **must decide if "no-state" correlates to "empty" or "`unknown`"**

3VL Comparison (Ranking) Rules

- 2VL Comparison:
  - **Results** in one of:
    - `less_than, equal_to, greater_than`
- 3VL Comparison:
  - **Results** in one of:
    - `less_than, equal_to, greater_than, unknown`
  - **If either** *(or both)* operands is `unknown`, **result must be unknown**
  - Can be **any of**:
    - `(unknown < false < true)` ← Probably preferred
    - `(false < unknown < true)` ← OK
    - `(false < true < unknown)` ← Unlikely to be interesting

`Trit`: An Alternative to `tribool`

- Would prefer C++ native-language type, but enables all the basics:
  - Holds **single discrete value**, one of: `false`, `true`, `unknown` *(defaults to `unknown`)*
  - Well-defined **inter-operation with `bool`**, C++ keyword literals **`true`**, **`false`**
  - **Relational operators** *(results in `unknown` if any operand is `unknown`)*
  - **`Trit::operator!()`** *(toggles `false`/`true`, `unknown` remains `unknown`)*
  - **Operators:** **`&&`**, **`||`**, **`^`** *(with `bool` and `trit` operands)*
  - *(Logically)* **throws when converting `unknown` => `bool`** *(exception handling can be disabled, then `unknown` => `false`)*
- All *(Above)* are expected of a ***(future)* native C++ type: `trit`**
- **Bonus**: Serialization, string-parsing (`-1`, `0`, `1`), (`'-'`, `'0'`, `'+'`), (`'F'`, `'?'`, `'T'`), (`"False"`, `"Unknown"`, `"True"`), etc.

```
class Trit {
  public:
    enum TritVal {
      val_false,
      val_true,
      val_unknown,
    };
  private:
    TritVal trit_val_;
  public:
    Trit(void) :trit_val_(val_unknown)
    {}
    // ...
};
```

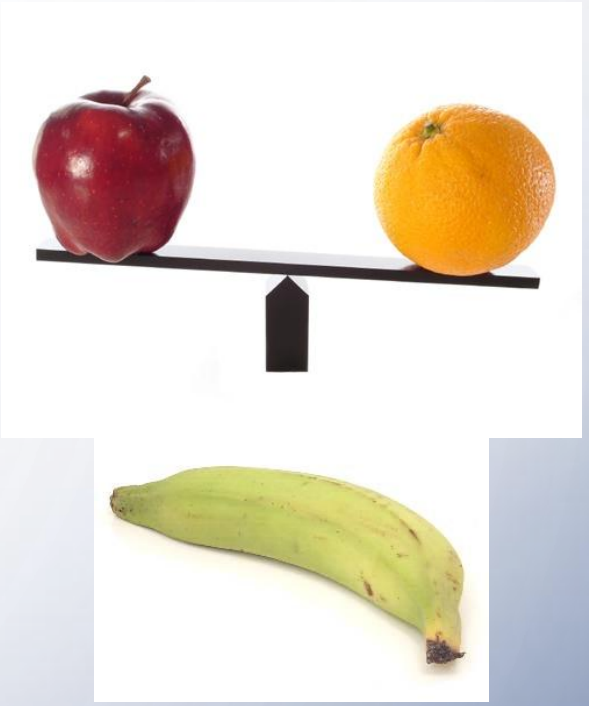
- Issues:
  - **Conversion: `trit` => `bool`** *(is well-defined to construct "from-`bool`", is lossy "to-`bool`")*
  - **Tricky**: **Comparing Values** versus **Identity-checking**

3VL Comparison Details

- **Ternary comparison-result requires four (4) states:**

```
namespace TritUtil {
  enum BinaryCmpResult {
    is_less_than    = -1,
    is_equal        =  0,
    is_greater_than =  1,
  };
}
```

```
namespace TritUtil {
  enum TernaryCmpResult {
    is_less_than    = -1,
    is_equal        =  0,
    is_greater_than =  1,
    is_unknown      =  2,
  };
}
```



- **A single "`Compare(Trit,Trit)`" does-the-work:**

```
class Trit {
    // ...
  public:
    // ...
    static TritUtil::TernaryCmpResult Compare(
      Trit operand0, Trit operand1);
};
```

- **Relational operators are "Predicate-statements" yielding ternary results.**

  *Example: "**is-equal**" yields one of `true`, `false`, `unknown`.*

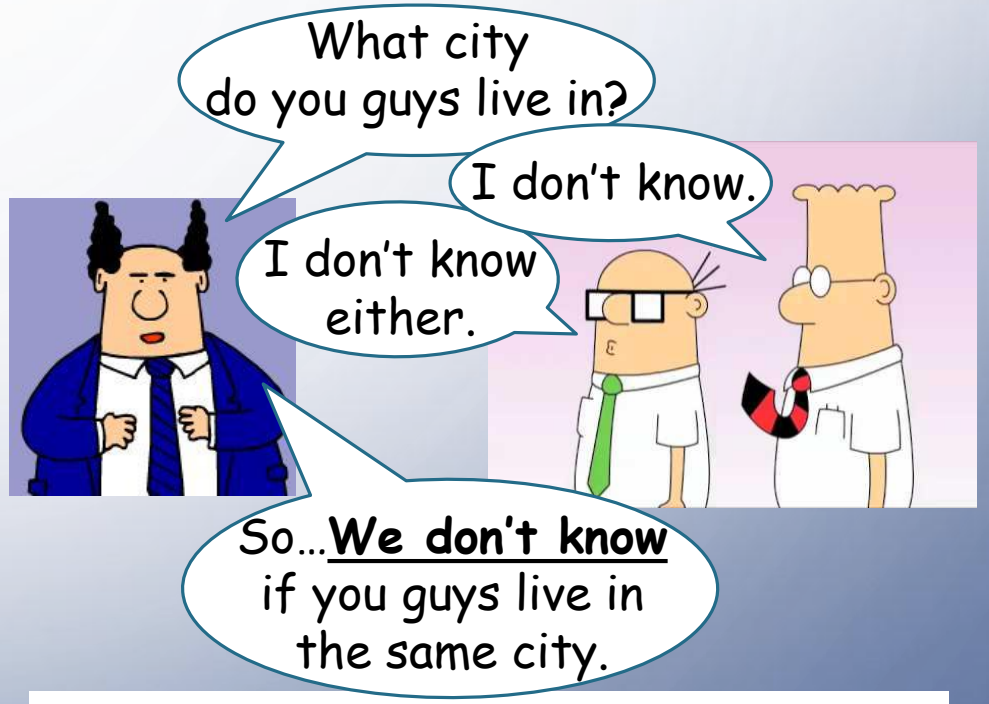
```
class Trit {
    // ...
  public:
    Trit operator==(const Trit& other) const;
    Trit operator>(const Trit& other) const;
    Trit operator>=(const Trit& other) const;
    Trit operator<(const Trit& other) const;
    Trit operator<=(const Trit& other) const;
    Trit operator!=(const Trit& other) const;

    // ...
};
```

3VL: Comparing Values

- **Comparing Values**: **Rank relative** to each other *(less-than, equal, greater-than)*
- **3VL**: **Adds unknown**, ranks **"empty" at different level** than "stateful"


```
// Result SHOULD be "unknown"
if(indeterminate(City("Chicago") == City(""))) {
  // ...we do not know...
} else {
  // ...we do know (is yes or no)...
}
```

Good! (Will enter here)




```
// Result SHOULD be "unknown"
if(indeterminate(City("") == City(""))) {
  // ...we do not know...
} else {
  // ...we do know (is yes or no)...
}
```

Good! (Will enter here)

Comparing Values Versus **Identity-Checking**

- **Comparing Values**: **Rank relative** to each other
  - *(One of)*: less-than, equal-to, greater-than, `unknown`
- **Identity-Checking**: The instance is a **member of what "category"?** *(what George Boole would call a "class")*
  - The **unknown category**
  - The **known-of-specific-state category** *(e.g., in 3VL would be the "`true`-category" or "`false`-category")*

```
Trit my_trit = unknown;
if(my_trit == unknown) {
  // ...
}
```

Comparing values: Would NEVER enter here! One-or-more operands is `unknown`, so result is `unknown`!

Identity-Checking: Values are the same *(both objects are in the same "category")*, so should ALWAYS enter here!

*Need specific syntax, or convention, to **resolve this ambiguity**!*

An OLD Problem!

- **Distinguish** between "**value-is-empty**" and "**value-is-not-present**"! *(Examples):*
  - **Date-of-death** *(not present – properly represented – is not dead yet!)*
  - **Date-of-birth** *(not present – missing data – should be treated as* `unknown`*, because logically exists, but we do not have it)*
- **Problem to be addressed: Missing Values!**
  - Am I comparing two values that are "empty", or is one-or-both "missing"?
- **One Approach**: Represent **the concept of "missing"** with **NULL**
  - **Structured Query Language (SQL)** – popular with databases – **is bivalent logic (2VL) with additional ad-hoc support for NULL** *(e.g., a "third-state")*.
    - Codd *(1970)* explicit stated was based on first-order predicate logic *(bivalent logic)*.
    - Codd added NULLs in 1979 *(Section 2.9 of Codd, 1979)*.
  - **Project MAC** Advanced Interactive Management System at MIT **used NULLs** in 1970-1971.
- **Issues Remain!**
  - **Missing Values treated inconsistently** by all vendors, **is inherent with the way SQL is defined**.
  - Pearson *(2006)* notes use of **NULLs in SQL databases**, "**introduces significant practical complications**"
    - Applying 3VL to a language based on 2VL: **"not-true" is not the same as "false".**

"Monty Python's Flying Circus", 1969-1974

- **Better Approach:**
  - Higher-order logics have three-or-more values.
- **Today: No "true" multivalued logics are used for real world applications** *(SQL is probably the closest.)*
  - ***With language additions, C++ could be the first!***

Rewriting our APIs

```
// Using 2VL
class Laser {
    // ...
  public:
    // Returns:
    //   true    -- we have "known/recognized-state" (might not be valid)
    //   false   -- we have "no-state"
    //
    bool hasStateAny(void) const;

    // Returns:
    //   true    -- we have "known/recognized-state" that is VALID
    //   false   -- we have "no-state", or state that is NOT-VALID
    //
    bool hasStateValid(void) const;

    // Returns:
    //   true    -- we have "no-state"
    //   false   -- we have at least some state (might not be valid)
    //
    bool isEmpty(void) const;

    bool operator==(const Laser& other) const;
    bool operator!=(const Laser& other) const;
    // ...other relational operators...
};
```

```
// Using 3VL
class Laser {
    // ...
  public:
    // Returns:
    //   true    -- we have "known/recognized-state" (might not be valid)
    //   false   -- we have "no-state"
    //   unknown -- we have "unknown/unrecognized-state"
    //
    Trit hasStateAny(void) const;

    // Returns:
    //   true    -- we have "known/recognized-state" that is VALID
    //   false   -- we have "no-state", or state that is NOT-VALID
    //   unknown -- we have "unknown/unrecognized-state"
    //
    Trit hasStateValid(void) const;

    // Returns:
    //   true    -- we have "no-state"
    //   false   -- we have at least some state (might not be valid)
    //
    bool isEmpty(void) const;

    Trit operator==(const Laser& other) const;
    Trit operator!=(const Laser& other) const;
    // ...other relational operators...
};
```

Could leave as `bool` if "empty" is the same as "`unknown`", or should make `Trit` if "empty" and "`unknown`" are different

Biggest Change *(Going Ternary)*: `if/else`

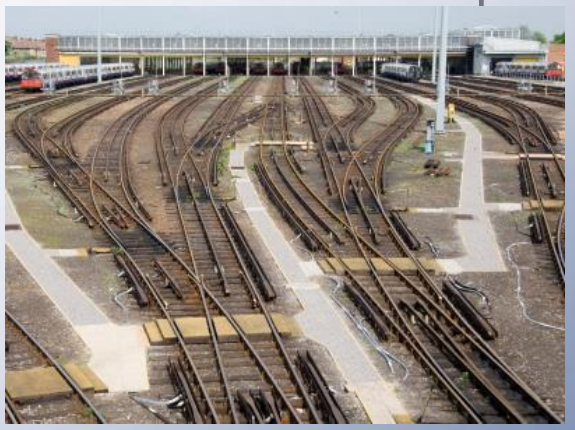
- **2VL to 3VL Biggest change**: `if/else` assumes binary
  - **Becomes less useful**; or
  - **C++ semantics could change** to enable MVL *(such as 3-way branch)*
- `if/else` is **fundamentally a binary switch**

Powerful: `if/else`

- Can compose **elaborate algorithms**, entirely **from "chaining"** binary switches

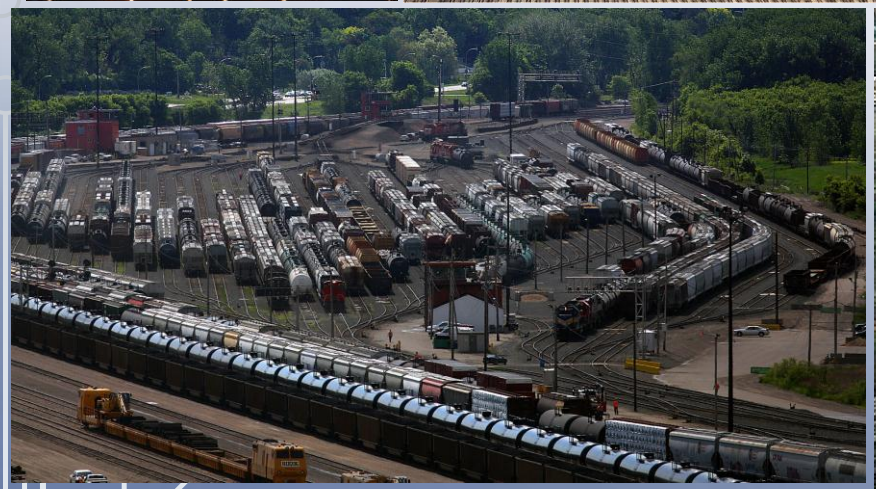











Problematic for MVL: `if/else`

- Is **problematic** composing **N-way switching** from `if/else`
  - **Inflexible** – changing algorithms requires a "rewrite"
  - **Hard to reason** *(e.g., "branch-testing" is surprisingly complex)*
  - **Does not scale well**
  - **Not composable for MVL** – hard to "connect" more than two output-results

- If you think **your (binary) system is hard to refactor**,
  - That's because **it is**!
  - `if/else` defines our algorithms, and it **is not designed to be composable**!

Changing Semantics: `if/else`

- **Possible**: Convert to **3-way branch**
  - `if/else` might become `if/else/unknown`

```
// TODAY:  "Binary" scenario
if(my_bool) {
  // ...true == my_bool...
} else {
  // ...false == my_bool...
}
```

```
// TOMORROW:  "Ternary" scenario
if(my_trit) {
  // ...true == my_trit...
} else {
  // ...false == my_trit...
} unknown {
   //...unknown == my_trit...
}
```

FUTURE: Possible "dangling-`unknown`"

- However, **scaling *(composability)* issues**…

  …*(next slide)*…

An MVL `if/else`: Issues-to-resolve

- **Scaling 2-way branch to 3-way, or N-way:**
  - Does the code compose well? *(Is it easy-and-obvious?)*
  - Should "dangling-`unknown`" behave like a `catch` clause, executed when any expression evaluates to `unknown`? *(That would be most consistent.)*

```
// TODAY:  "Binary" scenario
if(my_bool) {
  // ...true == my_bool...
} else if(my_bool2) {
  // ...(false == my_bool)
  //    &&
  //    (true == my_bool2)
} else if(my_bool3) {
  // ...(false == my_bool)
  //    &&
  //    (false == my_bool2)
  //    &&
  //    (true == my_bool3)
} else {
  // ...(!my_bool) && (!my_bool2) && (!my_bool3)
}
```

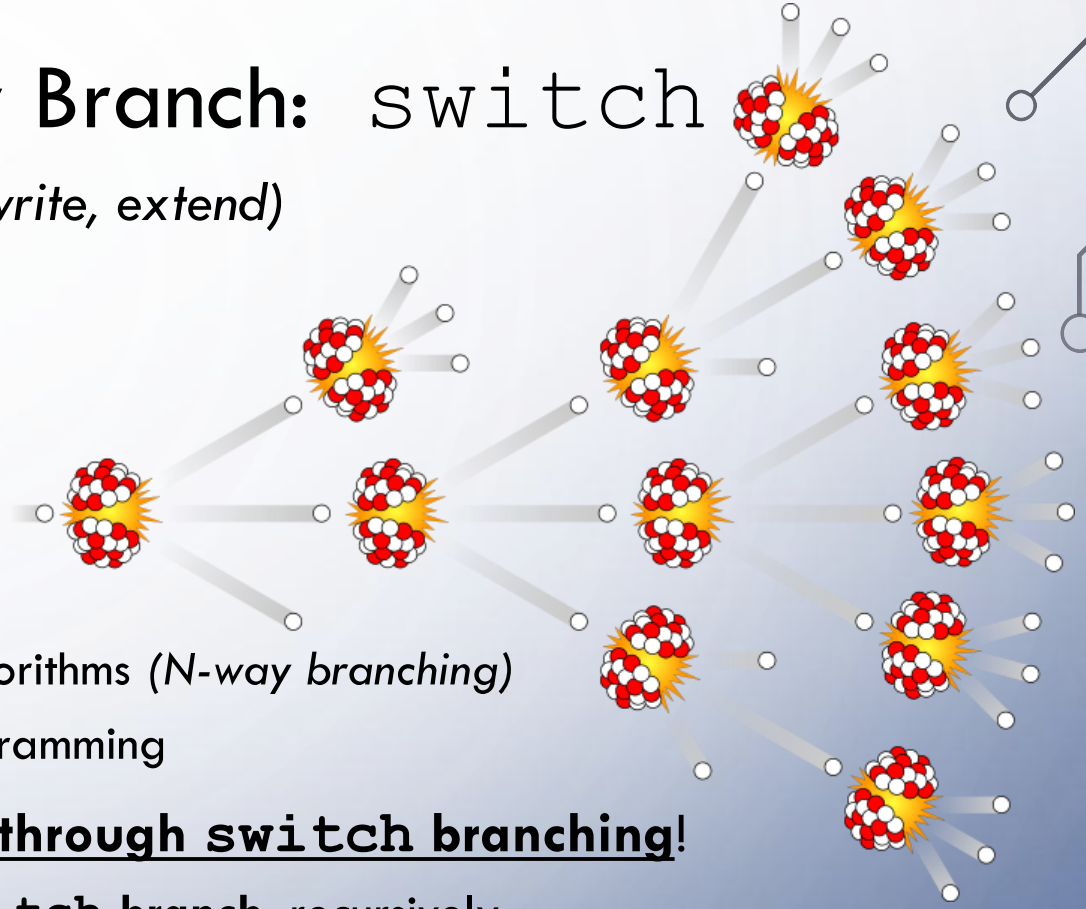
FUTURE: Should "dangling-`unknown`" logically "`catch`" any expression resulting in `unknown`?

```
// TOMORROW:  "Ternary" scenario
if(my_bool) {
  // ...true == my_bool...
} else if(my_bool2) {
  // ...(false == my_bool)
  //    &&
  //    (true == my_bool2)
} else if(my_bool3) {
  // ...(false == my_bool)
  //    &&
  //    (false == my_bool2)
  //    &&
  //    (true == my_bool3)
} else {
  // ...(!my_bool) && (!my_bool2) && (!my_bool3)
} unknown {
  // ...(unknown == my_bool)
  //    ||
  //    (unknown == mybool2)
  //    ||
  //    (unknown == mybool3)
}
```

We Already Have Multi-Way Branch: `switch`

- *(MUCH)* **More composable**! *(Easier to read, write, extend)*

```
switch(my_trit)
{
case Trit::val_true:  fooTrue();
case Trit::val_false: fooFalse();
case Trit::val_true:  fooUnknown();
}
```



- **`switch` is interesting** for MVL:
  - Useful **as implementation detail** within MVL algorithms *(N-way branching)*
  - Useful **as conceptual "metaphor"** for MVL programming
- MVL *(N-way branching)* **can be implemented through `switch` branching**!
  - Every **`switch`-branch can lead to another `switch`-branch**, recursively
  - **`if/else` is a "special-case" of `switch`-branch leading to `switch`-branch** *(where `if/else` assumes exactly two branch `cases`)*
- **Is done today**; Sufficiently useful that *(all?)* compilers issue *(compile-time)* **warnings when you omit `enum`-cases within a `switch` statement**. *(Helps ensure algorithm correctness/completeness for all branch options.)*

3VL Convenience: Quaternary Operator

- *(Today)* C/C++ Ternary Operator: `? :`

```
my_value = (my_bool) ? "Yes!" : "No!";
```

- *(Tomorrow)* For Ternary, need **Quaternary operator**: `? : :`

```
my_value = (my_trit) ? "Yes!" : "No!" : "Maybe?";
```

- **Is elegant**, because **discretely *(unambiguously)* resolves all 3VL states**
- Is **not composable above 3VL**, but 3VL is likely a "common-case" for MVL
- Can **make our own**!

```
template<typename VAL_RESULT>
VAL_RESULT Qo(Trit t,
              VAL_RESULT val_if_true,
              VAL_RESULT val_if_true,
              VAL_RESULT val_if_unknown) {
  // ...
}
// Poor man's quaternary-operator (Composable!)
my_value = Qo(my_trit, "Yes!", "No!", "Maybe?");
```

Composable MVL APIs

- Can **make MANY composable MVL functions**!

```
template<class CALL_IF_TRUE,class CALL_IF_FALSE,class CALL_IF_UNKNOWN>
inline
static auto Call(
  Trit trit_value,
  CALL_IF_TRUE callable_if_true,
  CALL_IF_FALSE callable_if_false,
  CALL_IF_UNKNOWN callable_if_unknown) -> decltype(callable_if_true())
{
  // ...
}
```

- Invoking:

```
Call(my_trit,
     myFunctorTrue,
     myFunctorFalse,
     myFunctorUnknown);
```

```
Call(my_trit,
     [](){ fooTrue(); },     // lambda-if-true
     [](){ fooFalse(); },    // lambda-if-false
     [](){ fooUnknown(); }); // lambda-if-unknown
```

```
my_value = Call(my_trit,
                [](){ return "True!"; },
                [](){ return "False!"; },
                [](){ return "Maybe?"; });
```

- Nesting/Composing:

```
// connect_now:
//   true    -- connect to device NOW
//   false   -- DO NOT connect now (return existing connect-status)
//   unknown -- Emulation-mode, DO NOT attempt external connection
//              (pretend we are connected)
const char* GetConnectionStatus(Trit connect_now)
{
  return Call(
              Call(my_trit,
                   [](){ return connectNow(); },   // lambda-if-true
                   [](){ return isConnected(); },  // lambda-if-false
                   [](){ return emulateNow(); });  // lambda-if-unknown
              [](){ return "Connected!"; },
              [](){ return "Not Connected!"; },
              [](){ return "Emulated!"; });
}
```

MVL APIs, Taking Parameters

- **Forwarding parameters** to *(composable)* MVL functions

```
template<class CALL_IF_TRUE,class CALL_IF_FALSE,class CALL_IF_UNKNOWN,class TYPE_P0,class TYPE_P1>
inline
static auto Call_p2(
  Trit trit_value,
  CALL_IF_TRUE callable_if_true,
  CALL_IF_FALSE callable_if_false,
  CALL_IF_UNKNOWN callable_if_unknown,
  TYPE_P0 p0,
  TYPE_P1 p1) -> decltype(callable_if_true(p0,p1))
{
  // ...
}
```

- Invoking:

```
Call_p2(my_trit,           // trit_value
        myFunctorTrue,     // callable-if-true
        myFunctorFalse,    // callable-if-false
        myFunctorUnknown,  // callable-if-unknown
        42,                // p0
        "param-name");     // p1
```

`template` specialization:

Could simply have one `Call()` function for:

- `return-void` or `return`-type
- Any number of parameters

Or, could have explicit "naming-convention" for parameter-count, return-type.

```
my_value = Call_p2(my_trit,
                   [](int n,const char* s){ return fooTrue(n,s); },    // lambda-if-true
                   [](int n,const char* s){ return fooFalse(n,s); },   // lambda-if-false
                   [](int n,const char* s){ return fooUnknown(n,s); }, // lambda-if-unknown
                   42,              // p0
                   "param-name");   // p1
```

An MVL Functor: `TritCall`

- A "**functor" that can be called later** *(when given `trit`, and optionally parameters)*

```
template<class CALL_IF_TRUE, class CALL_IF_FALSE, class CALL_IF_UNKNOWN>
class DECL_SDQ_DLL_COMMON_TEMPLATE TritCall
{
  private:
    const CALL_IF_TRUE&     callable_if_true_;
    const CALL_IF_FALSE&    callable_if_false_;
    const CALL_IF_UNKNOWN&  callable_if_unknown_;
  public:
    explicit TritCall(
      const CALL_IF_TRUE& callable_if_true,
      const CALL_IF_FALSE& callable_if_false,
      const CALL_IF_UNKNOWN& callable_if_unknown)
      :callable_if_true_(callable_if_true)
      ,callable_if_false_(callable_if_false)
      ,callable_if_unknown_(callable_if_unknown)
    { }
    auto call(
      TritUtil::TernaryValue ternary_value) const
      -> decltype(callable_if_true_())
    { /*...*/ }
    // ...other call() overrides with parameters, return-types...
};
```

- Invoking: Scenario (a)

```
auto ct = [](){return "True!";};
auto cf = [](){return "False!";};
auto cm = [](){return "Maybe?";};

TritCall<decltype(ct),decltype(cf),decltype(cm)>
  trit_call(ct, cf, cm);

const char* str = trit_call.call(my_trit);
```

- Invoking: Scenario (b)

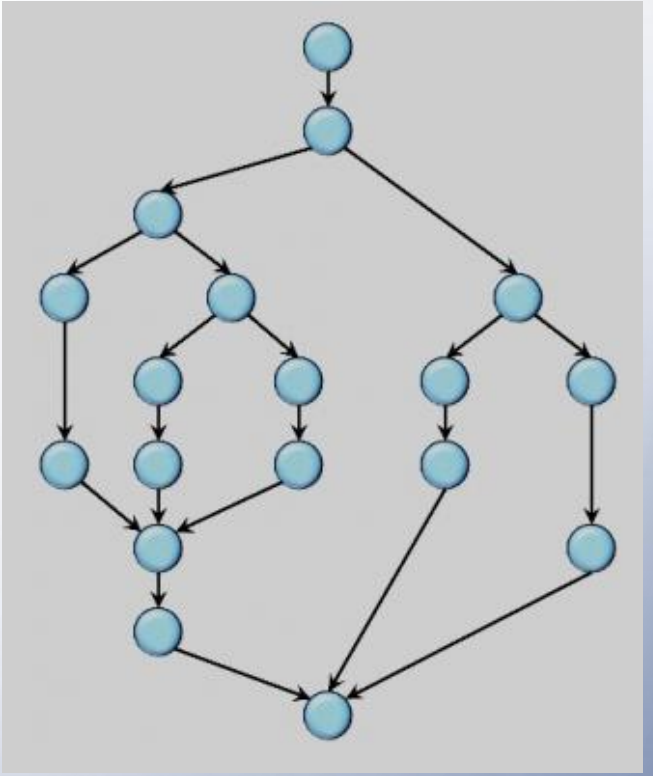
```
auto trit_call2 = Make_TritCall(
  [](){return "True!";},
  [](){return "False!";},
  [](){return "Maybe?";});

const char* str2 = trit_call2.call(my_trit);
```

- Invoking: Scenario (c)

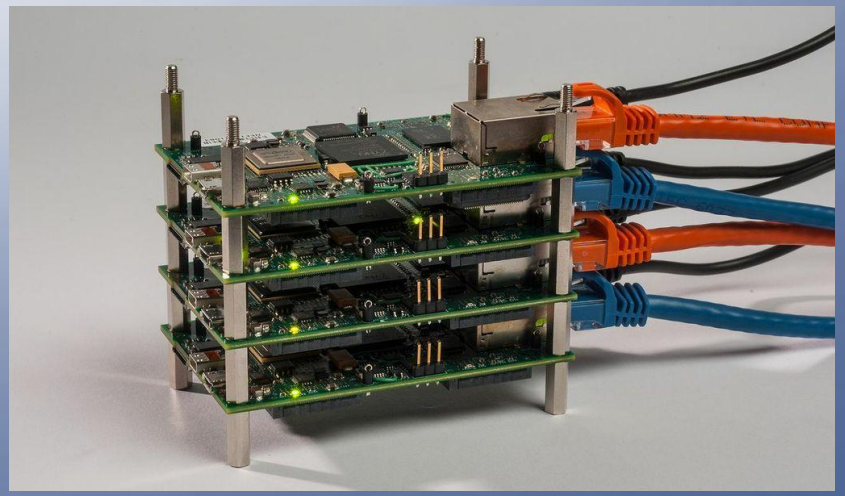
```
auto trit_call3 = Make_TritCall(
  myFunctorTrue,      // callable-if-true
  myFunctorFalse,     // callable-if-false
  myFunctorUnknown); // callable-if-unknown

const char* str3 =
  trit_call3.call(my_trit, 42, "param-name");
```

`TritCall::call()`
Overloads allow any number of parameters to be "forwarded"

`Trit` and Asynchronous APIs



- **Asynchronous APIs** are **increasingly the "trend"**
  - **Lots of cores available** *(more cores added because single-cores cannot go faster)*
  - **Lots more parallel processing** *(to resolve bigger problems)*
  - **Lots more concurrency/coordination** *(to resolve greater complexity)*
- `Trit` *(Ternary)* is **ESPECIALLY useful with asynchronous APIs**
  - `true` – yes, operation completed fine
  - `false` – no, operation did NOT complete fine
  - `unknown` – I'm still working, ask me later



`Trit` is like `const`

- ***Like chips* – *you can't eat just one***
- **APIs using `Trit`** is like APIs using `const`
  - You use `const` or not, there is no "try"
  - **Early days**: **`const` was debated** *(legacy code without `const`, lots of `const_cast<>()`, you either used or didn't, was tedious!)*
  - **Now**, is Answered Question: **Use `const`** *(everywhere appropriate)*
  - `Trit`: **Ditto** *(use everywhere appropriate)*
- Once you start **using `Trit`**, you **don't want to go back**. *(Programmers using `const` in APIs don't ever want to go back to APIs without `const`)*
- In **MVL**, our **APIs use `Trit`** *(instead of `bool`)*, **tends to propagate across whole system**

"South Park", Season 13, Episode 3, 2009

Interfacing 2VL and 3VL (or MVL)

- **Can be done**. **Can be tricky**.
  - Our programs consist of **data** and **algorithms**:
    - **Data**: Number of any-base *(radix)* can be projected/converted to number in any other base *(discrete, not hard)*
    - **Algorithms**: Logic of any dimension can be projected to logic of any other dimension
      - Extra rules/assumptions are required across dimensions
      - May be lossy "scaling-down"; may inject mathematical anomalies/ambiguities "scaling-up"
- **Practical** Interfacing **Issues**:
  - **Physical: Interfacing 2VL/3VL Hardware** *(CPU instructions, buses)*
    - Requires *(hardware)* standards, buses, protocols, etc.
  - **Logical: Interfacing 2VL/3VL Logic** *(algorithms, assumptions)*
    - Requires "policies", "conventions", and "rules"

Example Tricky Interfacing (*We already do…*)

- Interfacing can be ***tricky***: Logic and rules are different.
  - *Example*: **Multiplication using Roman Numerals is difficult** because system does not properly handle zero (`0`). In contrast, **multiplication in Arabic-decimal is relatively easy** and well-supported.
  - **Implication**: Some "natural-and-obvious" **MVL algorithms are not well-represented in 2VL**!
- Tricky Interfacing we already do:
  - **Interface "asynchronous-APIs" and "synchronous-APIs"** – *can be done*
    - Is easier for "Asynchronous APIs" to call "synchronous", because implementation completes synchronously
    - Is harder for "synchronous" to call "asynchronous", because must "block" until asynchronous implementation completes
  - **Interface "code-with-exceptions" and "code-without-exceptions"** – *can be done*
    - Is easier for "exception-code" to call "`noexcept`" *(does not worry about exceptions)*
    - Is harder for "`noexcept`" code to call "exception-code" *(must understand details for what to `catch`, handle, re-attempt, re-throw)*

Interfacing: ***Ternary*** CPU and ***Binary*** CPU

- **Existing** CPUs, Logic families, protocols **are binary**
- **How** would a Ternary CPU interface?
  - **Could be "hybrid" 2VL/3VL CPU**:
    - Native registers for 2VL and 3VL values *(e.g., bits and trits)*
    - Bus switches between 2VL and 3VL *(or use separate data channels for 2VL/3VL)*
    - 3VL CPU has "hybrid-logic" programming language features and algorithms (opcodes) that use both bits and trits
  - **3VL CPU could natively interface with binary**
    - Binary is treated as a "special-case" *(the "third" logic level would merely be "invisible" to the binary device)*
  - **Similar issues are resolved** when interfacing devices **across binary logic families**!
    - Output signal from one logic family is input signal to another logic family
- Recall:
  - Can execute **ternary logic on a binary CPU**
  - Can execute **binary logic on a ternary CPU**
  - Unless is a "hybrid-CPU", **only one is "native" execution** *(e.g., a performance penalty exists when executing the other)*

Step-Function: New Hardware, New Software

- **Transitioning to 3VL/MVL** *(from Binary)* **for both hardware and software** is an **Opportunity To Solve Problems**!
- **An appeal** to the *Good People Of The World*:
  - **Big/little-endian**: **ONE STANDARD**!
  - **Number representation**: **ONE STANDARD**! *(Balanced Ternary)*
    - After decades of bloodshed, "Two's Complement" is still NOT standard!
  - **Newline-character: ONE STANDARD**! *(Probably <CR>)*
    - *You gotta be kidding me, we need to support:* CR, LF, CRLF, FF, VT, NEL, EBCDIC-CR, EBCDIC-LF, NL, Unicode-NL, RS
  - **Unicode**: **No more multi-byte-characters**!
    - Text handled by
      - **6-trit tryte** *(729 values, 2 tribbles) (Compact!)*
      - ~~**9-trit tryte** *(19 683 values, 3 tribbles)*~~
      - ~~**12-trit tryte** *(531 441 values, 4 tribbles)*~~
      - **27-trit tryte** *(7.6255975e+12 values, 9 tribbles) (Holds Anything!)*
    - NOTE: Unicode defines:
      - Unicode 7.0 *(June 2014)* has 113,021 characters
      - Defines 1,114,112 code points (`0x … 0x10FFFF`)
    - **Unicode is biased to binary, might need ternary-character set standard**
  - **RISC or CISC**: **Pick one**!
    - Modern CPUs are mostly hybrids of both, but could have common standards and conventions

Summary: Ternary Code Differences

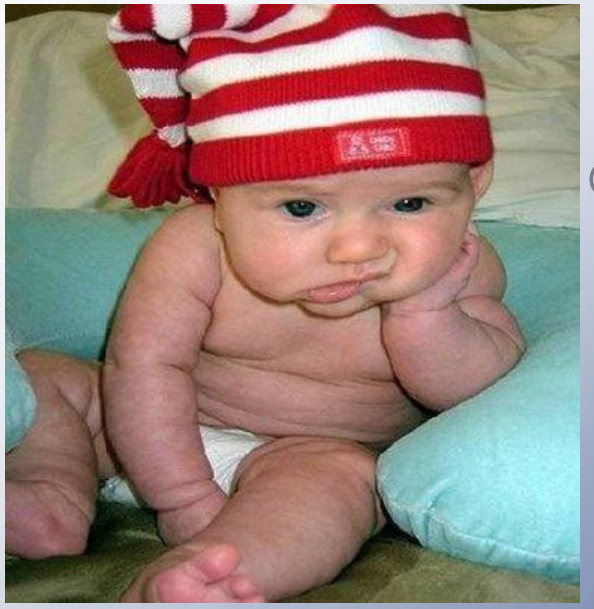
1. **Different assumptions/rules for 2VL⇔3VL**, which is why interfacing can be tricky
   - Must resolve corner cases, and use common assumptions
   - "Policy" and "Rules" required:
     - `bool` ➔`trit` is simple *(implicit promotion)*
     - `trit` ➔ `bool` is tricky …*what does `unknown` mean? Code expression can become ambiguous: `hasState()`, `isEmpty()`*
2. **`bool` is not used in APIs**, is **replaced with `enum`**
3. Explicit **conversion to `bool` becomes expensive**, often **ambiguous**
4. **Algorithms and data structures are 3-way or N-way**, uncommonly binary *(2-way)*
   - *Example: Balanced B-Tree is replaced with Balanced-Ternary-Tree*
5. Different questions are asked: **Code usually reflects a range of options, not binary choices**
6. **Predicate statements are more robust**; **Code frequently considers "`unknown`" scenarios**
7. **Error paths** are **propagated through APIs**
8. **Reduced reliance upon exception handling** *(throw, catch)*
9. **`if/else` becomes less useful**, is replaced with:
   - **`switch`** *(multi-branch logic)*
   - **Multi-branch "callables" or "functors"** such as `Trit`, `Call()`, `TritCall`, *(C++11 "Lambdas" are great for this)*
10. **Functional programming approaches become more appealing** *(code becomes more composable, less reliant upon `if/else`)*

Summary: MVL Application

MVL algorithms/data-structures are **interesting for**:

1. **Distinguishing between "empty" and "`unknown`" state**
   (Example: "moved-from" object … is it "empty", or an "zombie-object-with-`unknown`-state"?)
2. Implementing **Fuzzy Logic**
3. Implementing **Neural Networks**
4. **Managing reliability across modules, APIs, hardware** *(due to presence of `unknown`); Example: Distributed computing*
5. Supporting **Asynchronous APIs**
6. Supporting **error-paths or enhanced communication with the caller** across APIs
7. MVL algorithms are composable, enabling **dependency injection**

Summary: C++ Enhancements For Ternary *(Modest)*



- **Modest Ambitions:** These are relatively **"simple-and-safe" additions to C++.**
  (Aggressive requests are on the next slide)
  - **Keep `bool`** *(in an alternate universe, or in a new programming language, `bool` might not remain)*
  - **New keyword literal: `unknown`** *(similar to `true`, `false`)*
  - **New datatype: `trit`** *(with exclusive discrete possible values `unknown`, `false`, `true`)*
    - **Type Conversions:**
      - **Policy: Well-defined rule converting `unknown` to `bool`** *(probably `false`, perhaps `throw`)*
      - ***Implicit* promotion `bool` to `trit`** *(compile-time)*
      - ***Explicit* conversion `trit` to `bool`** *(compile-time)*
      - ***Explicit* promotion `trit` to `int`** *(compile-time)*
      - ***Explicit* conversion `int` to `trit`** *(compile-time)*
    - **Runtime check:** Converting `unknown` to `bool` *(?`throw` runtime exception converting from `unknown`?)*
    - **Runtime check:** Converting `unknown` to `int` *(?`throw` runtime exception converting from `unknown`?)*
  - **New Quaternary operator:** `trit_value ? true_case : false_case : unknown_case`
  - **Standardized, expanded-by-compiler: New 3VL, MVL libraries for composable N-way branch calls**
    (probably first added to Boost, http://www.boost.org/)
- Requires **further investigation:**
  - **Allow "dangling-unknown" on an `if/else`** chain

Summary: C++ Enhancements For Ternary (*Aggressive*)

- **Aggressive Ambitions:** These ambitiously support 3VL and MVL programming at the cost of **breaking C++ backwards compatibility**, and thus **may prove impractical**.
  - Should an alternative language be required, possible names might be: **C+++**, **C3+**, **C3VL**, **CMVL**
    - *(Note: C3 appears to be "taken" by Alexandre Cossette, interested in further details on that effort)*
- **Perhaps Unrealistic:**
  - **Deprecate `bool`** *(eliminates all ambiguity converting `trit` to `bool`, requires users to use `enum` or `trit` to create non-crappy APIs)*
- **Totally Serious** about these:
  - **New keyword literal**: **`unknown`** *(similar to `true`, `false`)*
  - **New datatype**: **`trit`** *(with exclusive discrete possible values `unknown`, `false`, `true`)*
    - **Type Conversions**:
      - ***Explicit* promotion `trit` to `int`** *(compile-time)*
      - ***Explicit* conversion `int` to `trit`** *(compile-time)*
    - **Runtime check**: Converting `unknown` to `int` *(?`throw` runtime exception converting from `unknown`?)*
  - **All logical (||, &&) and relational (==,<,<=,>,>=,!=) operators assume `trit`** *(unless explicitly overridden by the user)*
  - **New Quaternary operator**: `trit_value ? true_case : false_case : unknown_case`
  - **Standardized, expanded-by-compiler**: **New 3VL, MVL libraries for composable N-way branch calls** *(probably first added to Boost, http://www.boost.org/)*

- Requires **further investigation**:
  - **Allow "dangling-unknown" on an `if/else`** chain
  - **Consider deprecating `if/else`** *(possibly replacing with an MVL alternative)*

Conclusion

1. **MVL *(Ternary)* Is The Future**

2. **CPUs need it:**
   - **Harder to make a "next-gen" processor** that is better than the "previous-gen"
   - **Need efficiencies of MVL, 3VL** *(such as more efficient instruction set, higher radix economy)*
   - **New hardware natively supports more states** *(binary is inefficient and ambiguous)*

3. **Programmers need it** *(more!)*:
   - **Better APIs** *(handles corner-cases better)*
   - **Better thinking about the problem** *(more realistic reasoning for practical real-world scenarios)*
   - **Better algorithms** *(more expressive, performant, powerful, composable)*
   - **Better error handling** *(resolved corner-cases)*

Special Thanks to

Bruce Baily, EE

David Van Maren, EE