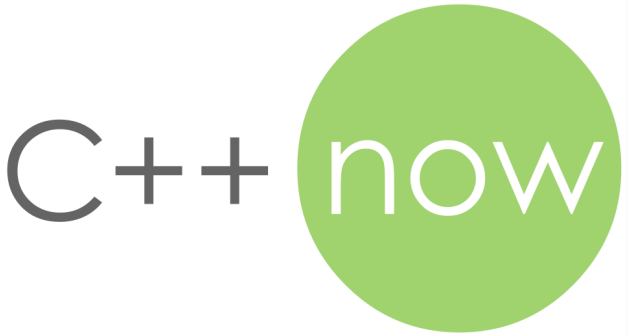
The CPU Cache:

Instruction Re-Ordering Made Obvious



charley bay
c.bay@f5.com
charleyb123@gmail.com

F5 Networks, Inc.

IN A PARALLEL UNIVERSE...

The C++ "As-If" Rule and the Role of Sequence:

Everything You Ever Wanted To Know

C++ now

charley bay

c.bay@f5.com

charleyb123@gmail.com

F5 Networks, Inc.

Abstract

- This is an introductory (i.e., "**First Principles**") dive into **instruction re-ordering** (at **compile-time**, and at **run-time**) **due to conspiring** by the **compiler and CPU** to make most efficient use of execution units and resources within the CPU processor core.

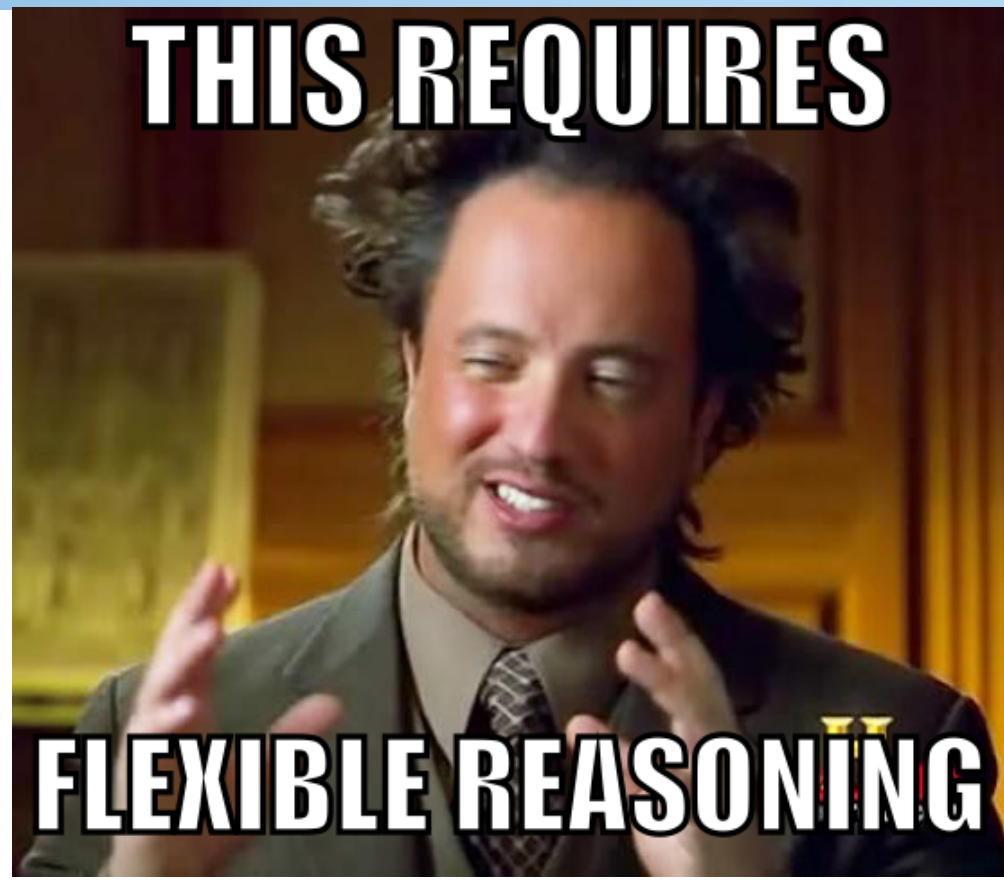
Corollary:

What is "**Sequence**", and *what is its role*?

- Take-Away:
  - **Dependencies** … Worry!
  - **Instruction Order** ... Don't Worry! *(Be Happy!)*

Goals

1. Explain the importance of the C++ "**As-If" rule**
2. **How "out-of-order" execution occurs**
   - (at *compile-time*, at *runtime*)
3. **Why that's a "Good Thing"™**



Some discussion:

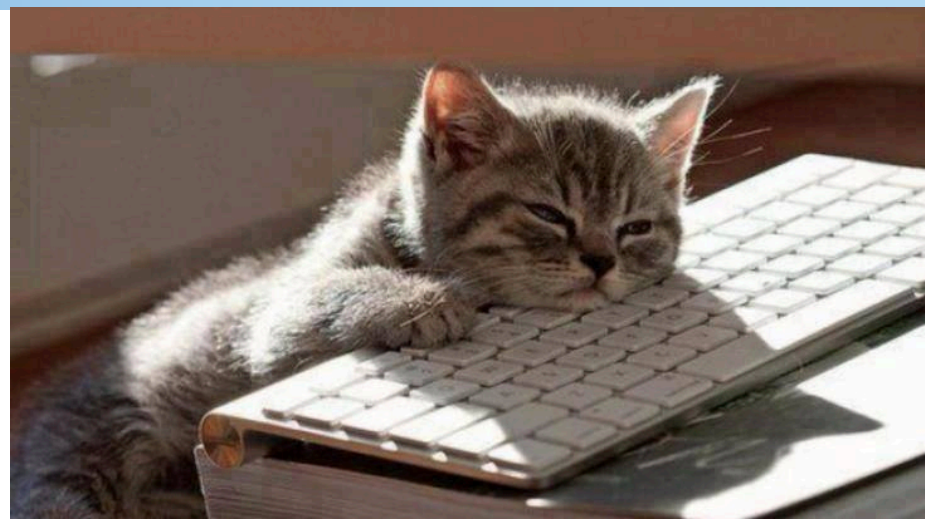
Software Engineers can **be more "flexible"** in their reliance on **imperative versus sequential** execution, **physical versus logical sequences**, and how to **leverage the cache line**.

After Today...

- After Today, **You Will Know**:
  - My statements **executed in the CORRECT order**
  - My statements **executed in the EFFICIENT order**
  - My statements probably did **NOT** execute **in the order I specified**
- After Today, **You Will Be**:
  - **CONFIDENT** in your compiler
  - **TRUSTING** of your CPU
  - **GRATEFUL** you don't need to worry about instruction order
- **Moral Of The Story**:
  - It is **YOUR JOB to Design and Implement** with concern **for dependencies**
    - *We are Engineers! This is what we do! Single-threaded or Multi-threaded!*
  - It is a **Fool's Errand** to be concerned with **Instruction Order**
    - *Pointless! Distraction! Red-Herring!*

The Good News Take-Away:

"Be More Lazy And Ignorant!"



- You **should NOT need to know** any of this!
- You should **WORK** to **be MORE LAZY** regarding what you care about!
- You can **TRUST** the **CPU is always correct**!
  - And usually the **Compiler too**!

The More You "Know", the more likely you are hurting yourself and your algorithm

- …through **misplaced and incorrect assumptions**

(definition) **Assumption**: Any perceived truth that is uncontrolled in the design or implementation

You CANNOT possibly know the architectural and runtime-specific details! (THEY CHANGE!)

ANY assumption is EVENTUALLY violated!

Today's Agenda

1. **Why Care** About **Sequence**? (Why *not*?)
2. (Order) **Assumptions In Design**
3. **Imperative vs. Sequential Devices**
4. **Physical vs. Logical Sequences**
5. **Compiler Internals**: Optimizing Logical Dependencies
6. **Program Execution**
   - Process Image (data and text segments)
   - Multi-Level CPU Cache
   - Locality: How The Cache Works
   - CPU Instruction Pipeline
   - CPU Architectural and Dynamic Registers (Register Renaming)
7. The **C++ "As If" Rule**
8. **Summary**

(not time for today):

- Influencing Instruction Order
- Hyper-Threading (HT) and Simultaneous Multithreading (SMT)

There is no guarantee that statements will execute in the order that the C++ programmer specified.

- This is called, "**Instruction Re-ordering**"
- *Why?*
  - Compiler Optimizations
  - CPU pipelining

Toy Story (1995)

- All is *not* just Chaos!

The Big Lebowski (1998)

There ARE Rules!

(What Are The Rules?)

Rules: That Upon Which We Can Rely

- We **Trust To Be True**, *Or All Is Lost:*

1. C++ "Well Defined" Behavior
2. The Compiler Works
3. The CPU functioned properly
4. The Laws Of Physics

C++ "Well-Defined" Behavior is very clear, and "Real-World" Practical!

It Does! (Usually!)

It Does! (Always!)

Latencies exist! (Always!)

System Failures:
The Software Engineer **ASSUMED** something other than (1), (2), (3), (4)

The Office (2005-2013)

- **No** Other Rules Exist
  - If designing "ecosystems", final rule is "*stuff breaks*"

Why Care About Sequence?

Because we reason about procedural algorithms

We DO Care About Sequence: Why?

(1 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

We DO Care About Sequence: Why? (2 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

- Sequence is how we know anything at all!
  - What **state** exists
  - How that state **changes/mutates** over ***time***

A *Some* **guarantees must exist** regarding **sequence**, or we cannot reason at all.

We DO Care About Sequence: Why? (3 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

- **Imperative** Programming
  - Sequence of, "**Do This!**"
- Anatomy and behavior of **algorithmic constructs**
  - **Sequence** (*order*)
  - **Selection** (`if`/`else`, `switch`)
  - **Iteration** (`do`, `while`, `for`, `foreach`)

B "Sequence" is a "**building-block**" (stepping-stone) in **logic training**!

We DO Care About Sequence: Why? (4 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

- "**Stepwise-reasoning**" introduces students to programming
- **Is simplest (most atomic)** level of reasoning
- **Is easier** to do "**ordered-walking**" **of myopic cases**, rather than understand the processing "big-picture"

C **Assumptions** related to "sequential reasoning" **become seductive**

We DO Care About Sequence: Why? (5 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

- Especially in **large systems** (where we do not know where to get started)
- Especially in **complex systems** (where it is hard to reason about "big-picture" behavior)

D Sequence is our "**Go-To" tool** when we are "**lost**" or "**stuck**"

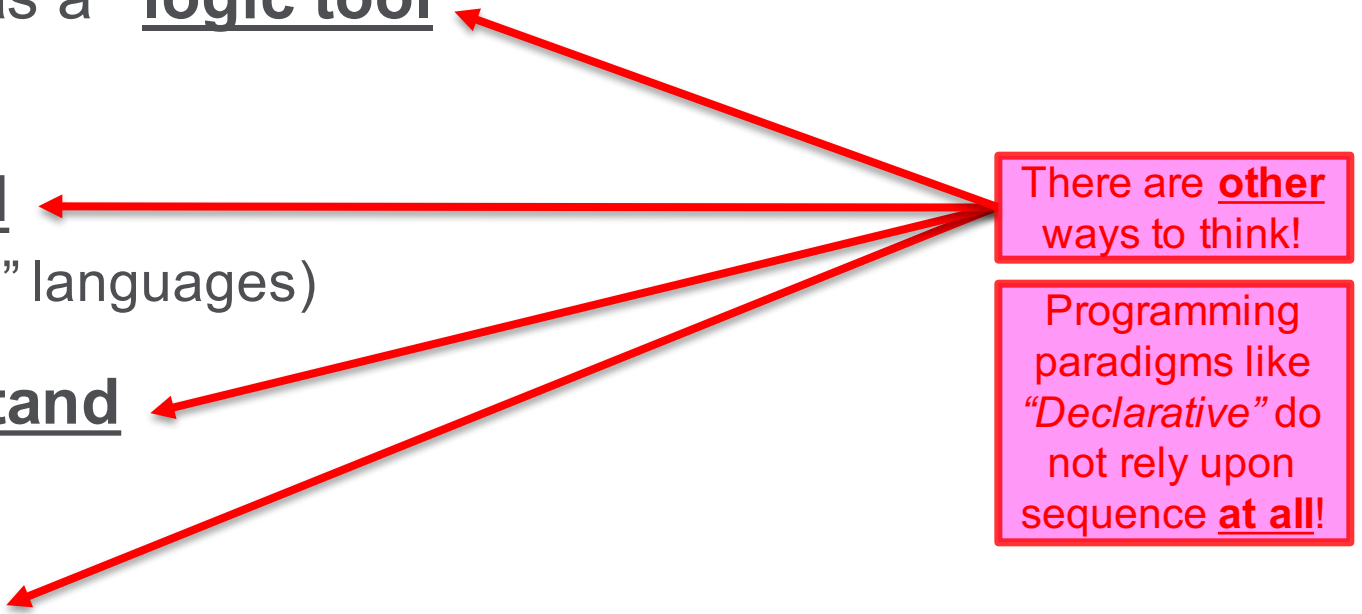
We DO Care About Sequence: Why? (6 of 6)

A. We **leverage sequence** as a **"logic tool"**
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it **"adds value"**



BUT:

- **Many** of these expectations **(assumptions) "do-not-hold"** for **how things actually work** in the compiler/interpreter, and within the CPU
- **Alternative programming paradigms**, and **the C++ Language** *(explicitly!)* do not grant "sequence" this level of reverence

A “Definition-Of-Terms” Problem!

- How does “**Sequence**” relate to “**Algorithm**”?

An **algorithm**
is NOT defined
by the **sequence of statements**
the programmer types.

An **algorithm**
is NOT defined
by the **sequence of statements**
the programmer types.

An **algorithm**
IS defined
by the **sequence of dependencies**
the programmer types.

Corollary: **Sequence** is a **tool** to **manage dependencies** across instructions.

What Defines An "Algorithm"?

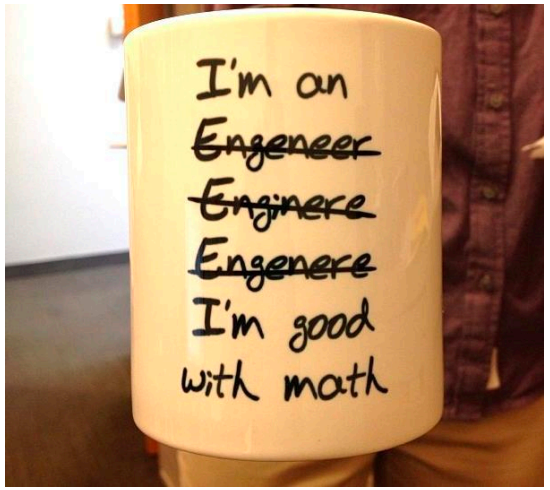
Algorithm
is defined by
logical dependencies
(*not* physical sequence)

- **Example Physical Sequences**:
  - The **compiled program** (instruction order is "fixed" on-disk)
    - *Is "same-program"* if `inline` functions or not
    - *Is "same-program"* after re-compile with different layout
    - *Is "same-program"* compiled with different optimization levels
  - The **loaded program** (instruction order is "fixed" in memory)
    - *Is "same-program"* after address-load randomization
    - *Is "same-program"* after dynamic linking/loading
  - **Self-mutating virus**
    - *Is "same-program"* after (randomizing, mutating) physical instruction sequences

Can change **physical sequence** (logical sequence remains intact)!

Is "Same Program" because algorithm exhibits the same **logical dependencies**!

Job: Software Engineer



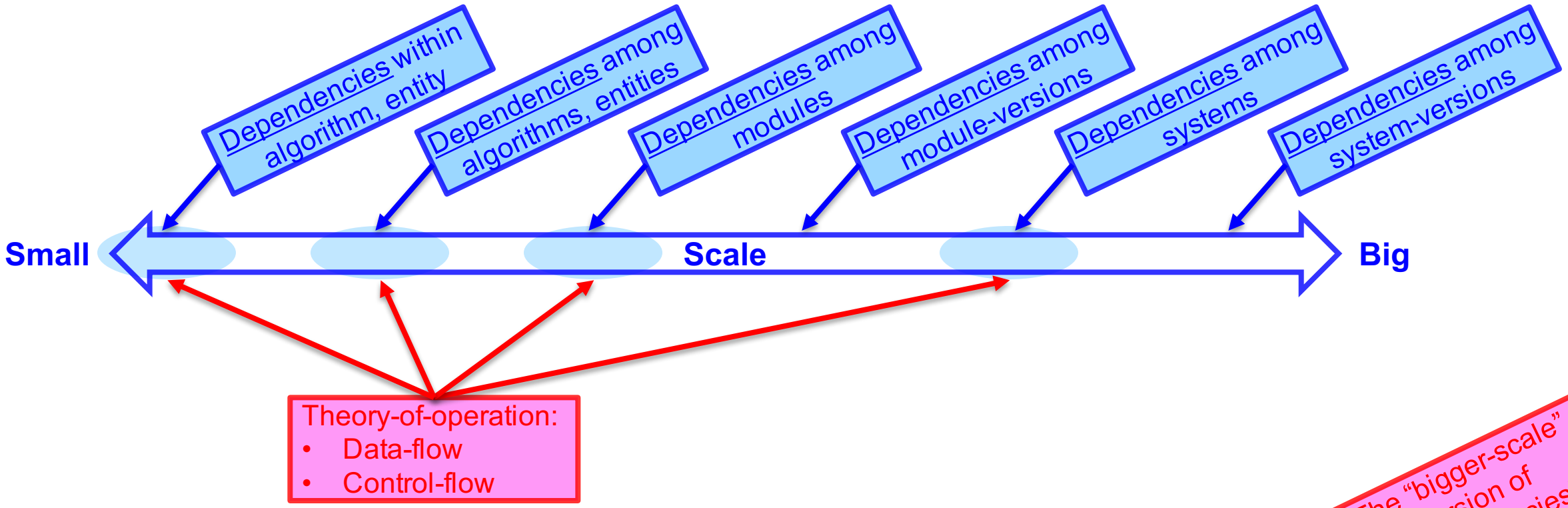
- *Software Engineer*: Our job is **NOT** to **manage lines-of-code**!
  - *Might be your full-time job if you live "Configuration Management" (CM)*
- These are **MECHANICAL operations** to "**keep our workshop clean**":
  - **Add** lines-of-code
  - **Remove** lines-of-code
  - **Move around** lines-of-code (e.g., "refactor")
  - **Reformat** lines-of-code (e.g., "beautify")
  - **Manage** lines-of-code (e.g., "version control")
  - **Build/ship** lines-of-code (e.g., "installer", "CM")

Is NOT software engineering! Is just "part-of-the-job"!

Are based on INTERACTIONS among DEPENDENCIES!

Software Engineer: Our job **IS** to **manage logical dependencies**!
- **Analyze and understand**: Behavior, Capabilities, Alternatives
- **Design and implement**: Algorithms, Features, Functional Changes

Logical Dependencies



- **Logical Sequence** is how we reason about behavior
  - At "**Small" scale**, reason about **dependencies**, **order-of-operations**
  - At "**Big" scale**, reason about **coupling**, **transactions**, **status-and-phases**

The "bigger-scale" version of "Dependencies"

Home | Sims | Stage | Tiles | Trains | Wang

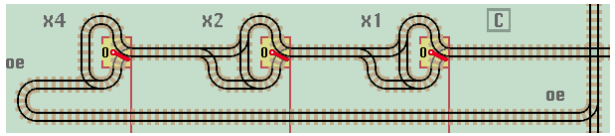
Turing Trains

computational train track layouts

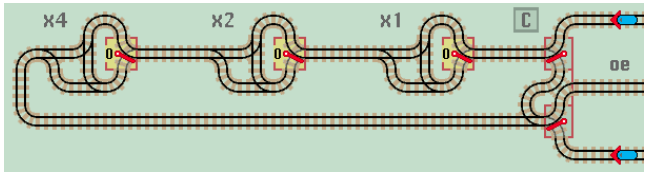
http://www.cr31.co.uk/stagecast/trains/tt2_func_basic.html

Reasoning About Sequence

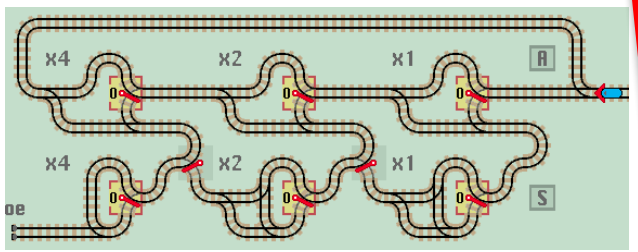
Clear and Error Line



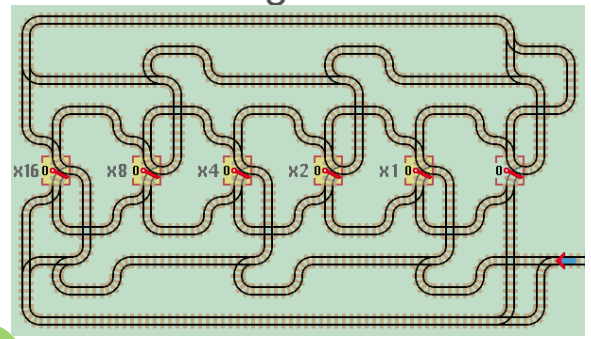
Count and Clear Function



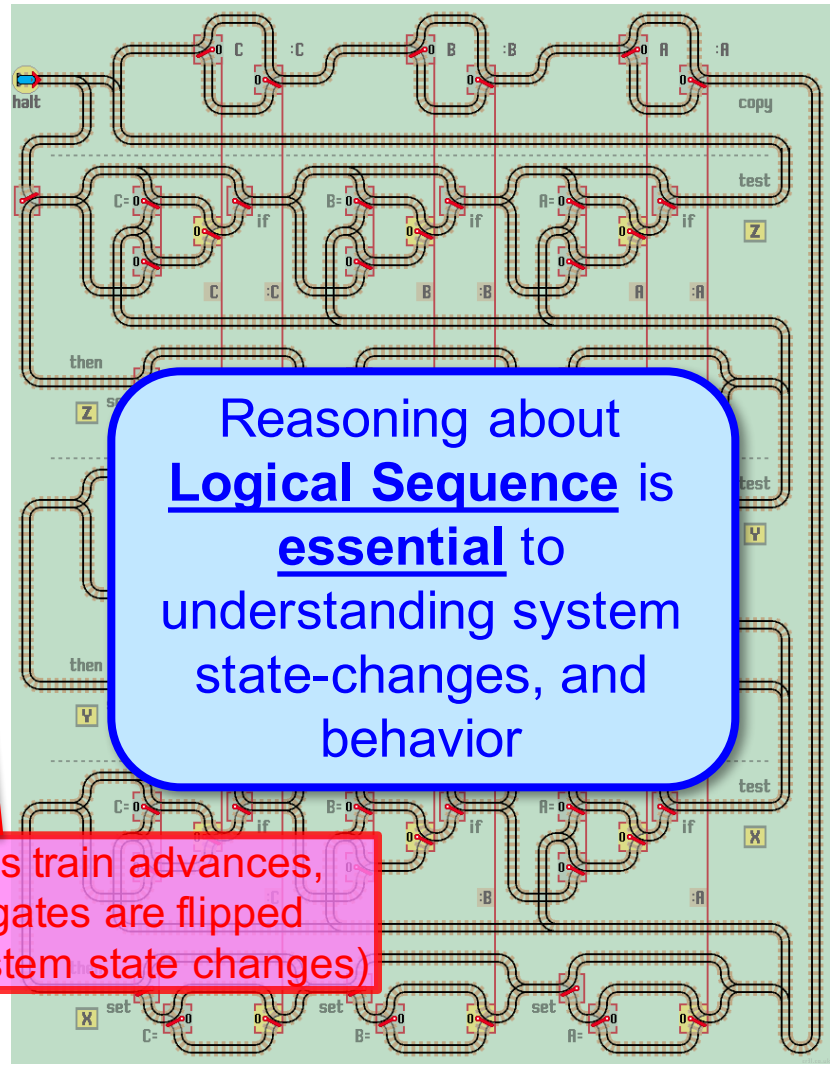
Accumulator Function



Rotate-Right Function



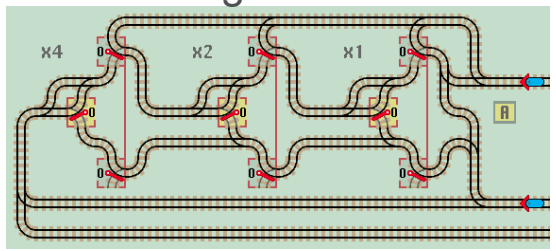
Triangular Number Series Calculator



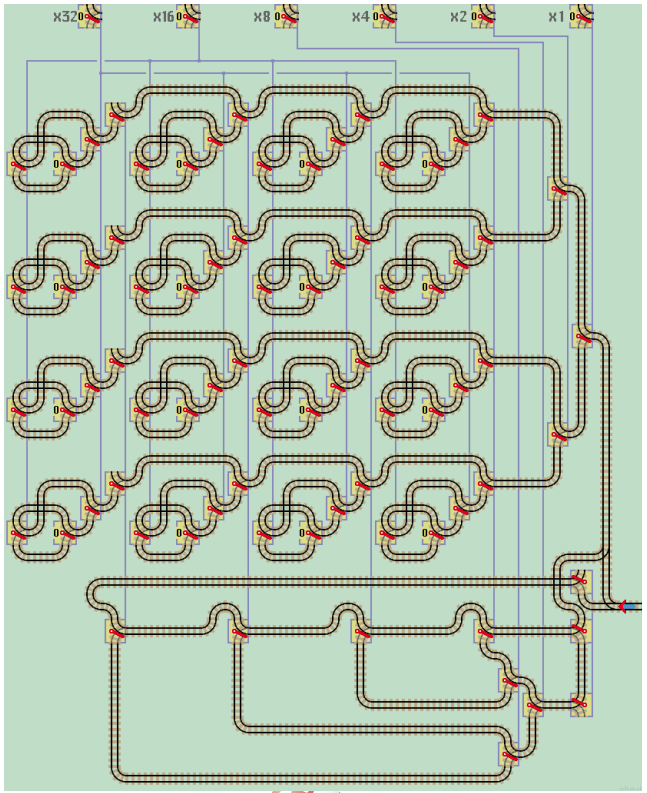
Reasoning about **Logical Sequence** is **essential** to understanding system state-changes, and behavior

As train advances, gates are flipped (system state changes)

Leading Zero Latch



16-Bit Random Access Memory



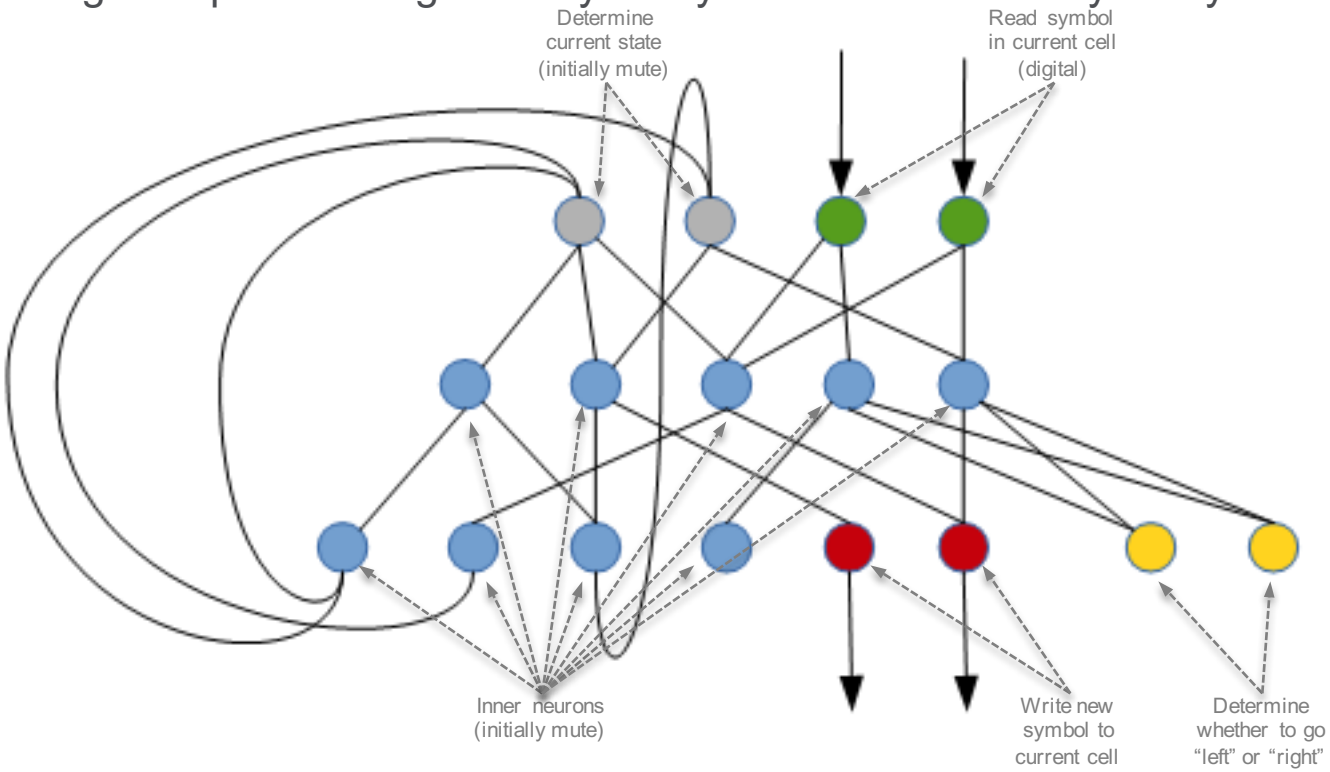
Recurrent Neural Network: *Reasoning About Sequence*

- **State-machine transitions** imply pre/post values based on change-in-state

Reasoning about
logical dependencies,
mutated through
sequential execution,
is required to understand the system

Recurrent Neural Network

Turing-Complete through finitely many neurons with finitely many states



Adapted from Hans Stricker
http://stackoverflow.com/questions/2990277/how-useful-is-turing-completeness-are-neural-nets-turing-complete

Why NOT Care About Sequence?

*So **unnecessary constraints** do not creep into our systems*

Controls Are Constraints

- **Constraints**: We need them! They **define our system**!
  - **Necessary Constraints**: Our system **behaves as expected**
  - **Unnecessary Constraints**: Restrictions that provide **negative-value**
  - **Missing Constraints**: Our **system fails** in some scenarios
- **Controlling** (constraining) for **Sequence**:
  - Want to **constrain** for "**necessary**" sequence
  - Want to **NOT constrain** for "**unnecessary**" sequence

How do we know if controlling a given sequence is "necessary" or "unnecessary"?
Hint: **Dependencies!**

There is **no such thing** as a "**neutral constraint**"
By definition:
Constraints restrict behavior for **desirable** (*positive*) or **undesirable** (*negative*) result

Exerting Necessary Control

- Want to **exert NECESSARY control**
  - Specifying **necessary detail is how we design/implement**!
    - …should not be violated!
    - …becomes our design/implementation ASSUMPTIONS!
- Want to Be **SILENT** on **unnecessary detail**
  - Is not "requirement" – we do not care
  - Do not want to restrict OPTIONS
  - Do not want to restrict THINKING
  - Do not want to add unnecessary ASSUMPTIONS

Exerting necessary control is how we **Design and Implement**!
It **establishes behavior**, **expectations**, and allows us the safety to **make valid assumptions** when reasoning about the system!

Avoiding Unnecessary Control

- Want to **avoid UNNECESSARY control**
  - Unnecessary controls create **unnecessary assumptions**
    - ...which are eventually violated
    - ...which restrict your options
    - ...which limit your thinking
  - Unnecessary controls create **unnecessary constraints** (controls *are* constraints!)
    - ...which **disallow efficiencies and technologies** within the compiler and CPU

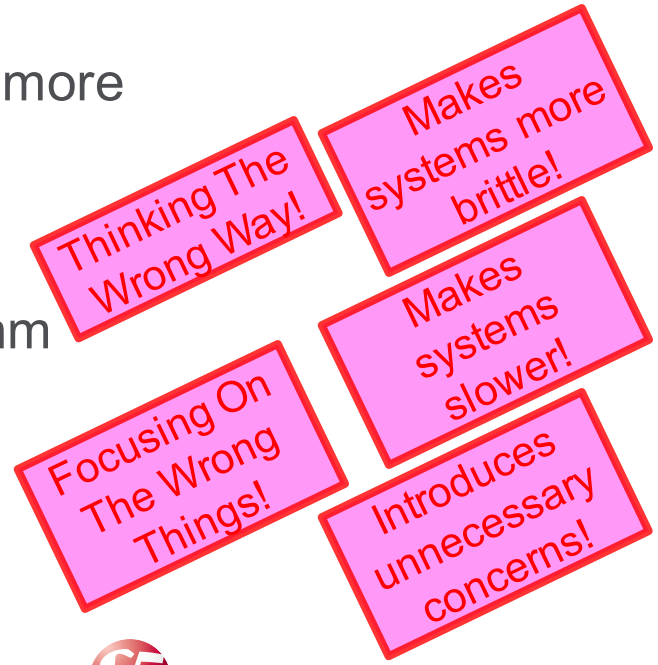
You're not leveraging the compiler and CPU!
(You disallow them from fully doing their jobs!)

Penalties from inflexibility!
(Benefits from flexibility!)

The **More Control You Exert**, the more likely **you hurt yourself and your algorithm**
- …through **misplaced and incorrect assumptions**
  - …that eventually get violated
  - ...and which disallow efficiencies

Why Managing Instruction Order Is Bad

- **Efficiency** Penalties
  - We **refuse to leverage** the **technologies in the CPU**
  - We **disallow optimizations** the compiler provides
  - **Future** system **performance/maintenance** scales **worse**, not better
- **Correctness** Penalties
  - The algorithm is **more likely to be wrong** (more corner cases)
  - We **blind ourselves to alternatives** that scale increasingly better with more resources, larger data sets, and new technologies
- **Effort** Penalties
  - *(usually),* it is **more work to encode more assumptions** in the algorithm
  - *(usually),* it is **more work to hold more assumptions** in our brains



Penalties All Around!

Assumptions: Avoid When Possible

- Most programmers have **too many *(assumptions)***
  - We must **ALWAYS** manage **"some"** assumptions
- You **SHOULD NOT CARE**!
  - about many things, including **instruction order**
- Should: Make **NO ASSUMPTIONS** regarding **"order"**!
  - Should accept that "Instruction Order" is highly ***fickle***, ***erratic***, and seemingly ***capricious***.
    - We should **not want to control** it
    - We should **not want to care** about it
      - *Can design/implement to where we do not care about order!*

If **no** dependencies exist, then **"order" is irrelevant!**

We DO NOT design and implement based on "instruction order". *(We don't care!)*
We **design and implement** based on **dependencies**.

Example Scenario: (Order) Assumptions In Design

Assumptions creeping into our design, into our implementation

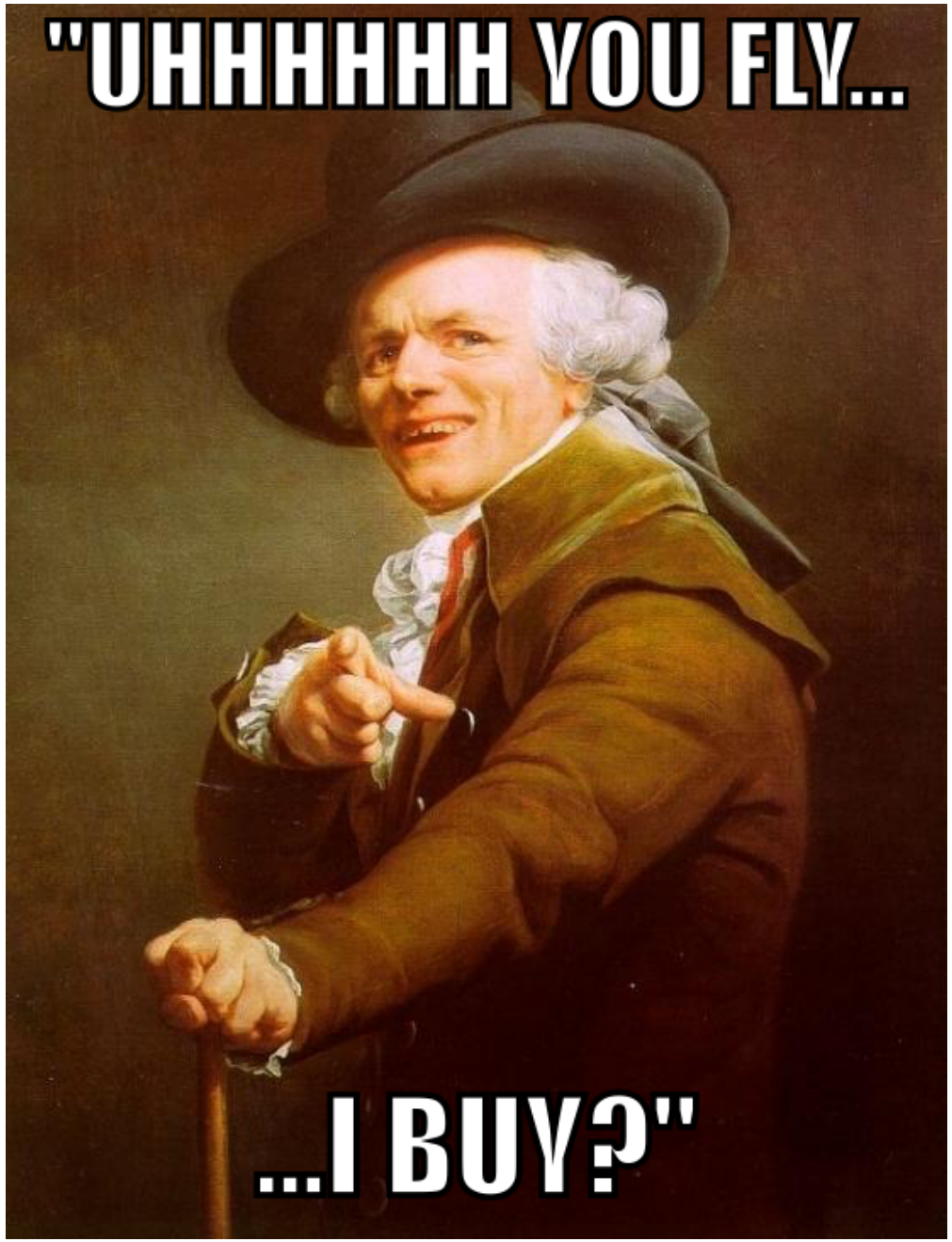
“You Fly, I Buy”

...A scenario (with assumptions)

- Example Problem Statement:

“I will pay for the pizza, you go get it.”

- **How many assumptions** will we make?



Algorithm

...or "Design", or "Control-Flow", or "User-Story", or "Implementation", or ...

1. @Me specify, "Supreme w/extra olives"
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

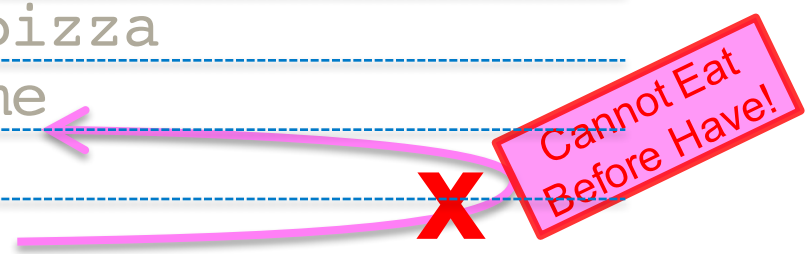
What is the **necessary order**?

For which should we **NOT CARE** regarding order?

Order That Matters

...Necessary constraints (warranted coupling establishing order)...

```
1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10.@Me eats pizza
```



What is the **necessary order**?

For which should we **NOT CARE** regarding order?

On “1. @Me Specify…”

…*Necessary* constraint, or *Assumed* constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

Can give you money first!

Can phone you with order after you leave!

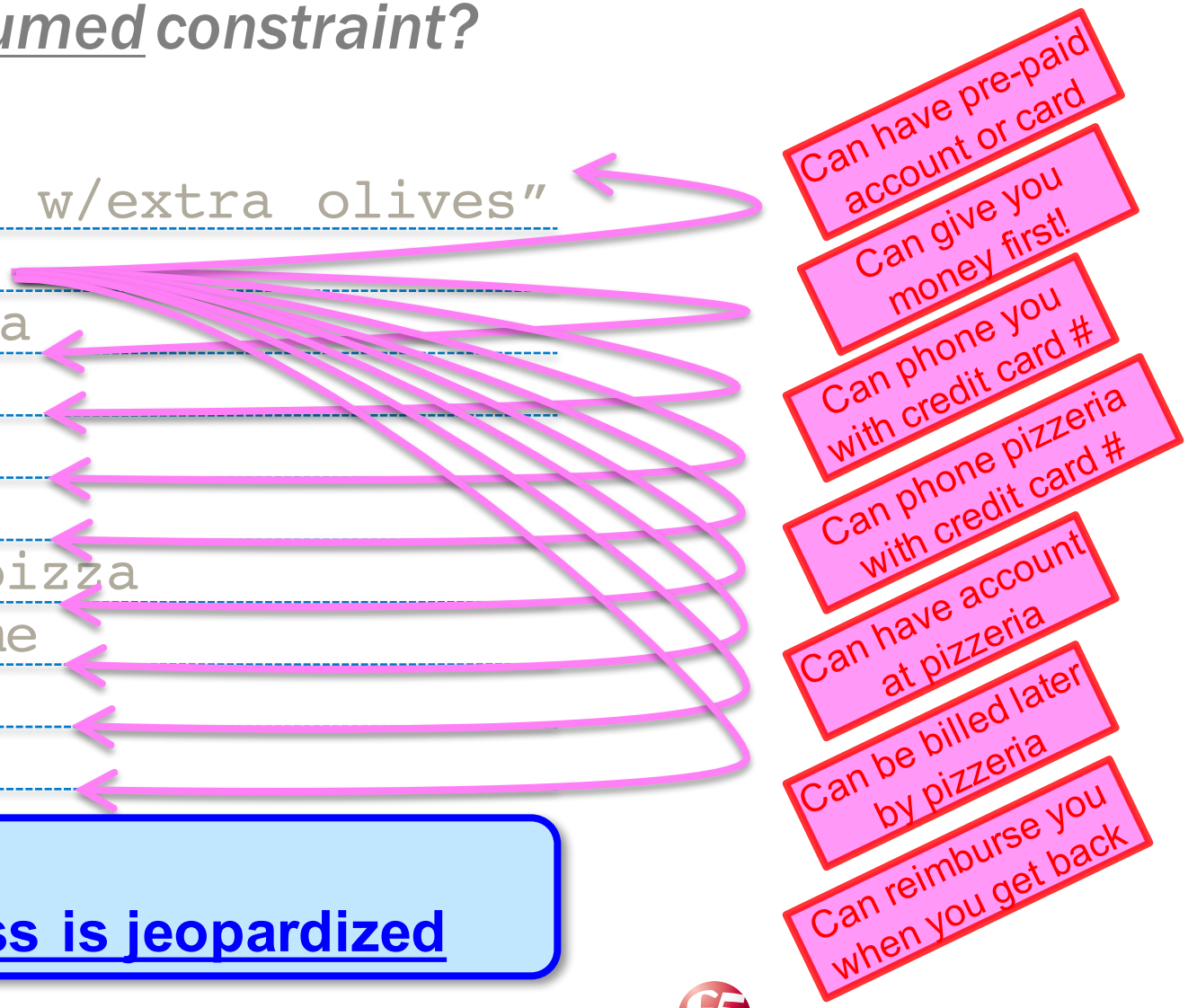
Why care about sequence?

- ONLY if you **get the wrong pizza**

On “2. @Me Give You Money”

...*Necessary* constraint, or *Assumed* constraint?

```
1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10.@Me eats pizza
```



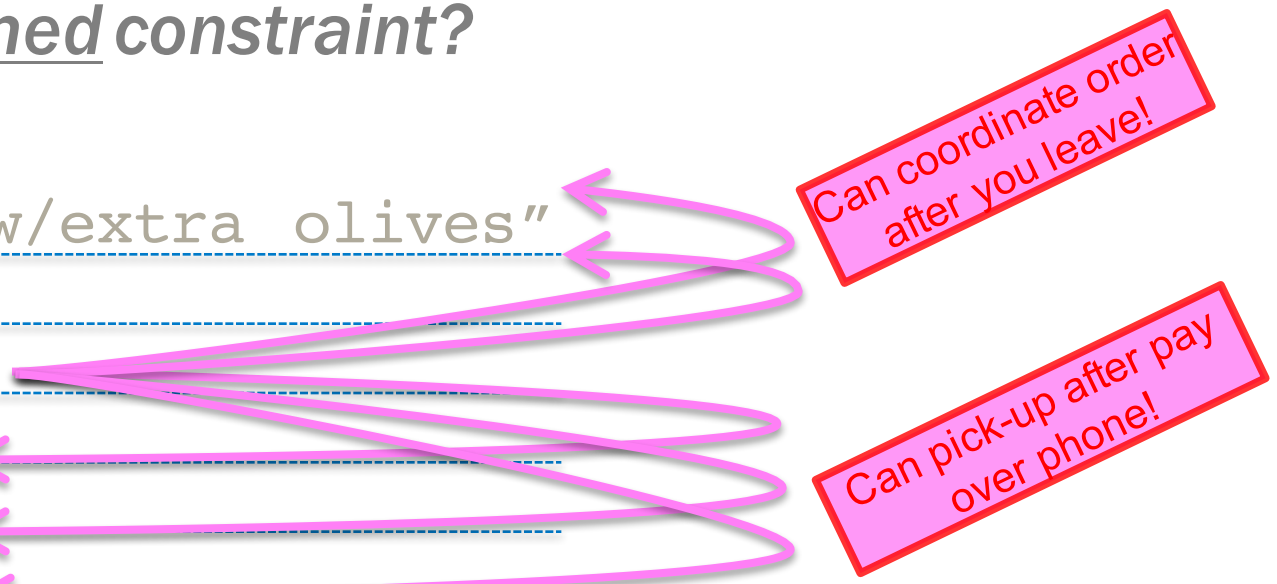
Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “3. @You Drive To Pizzeria”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



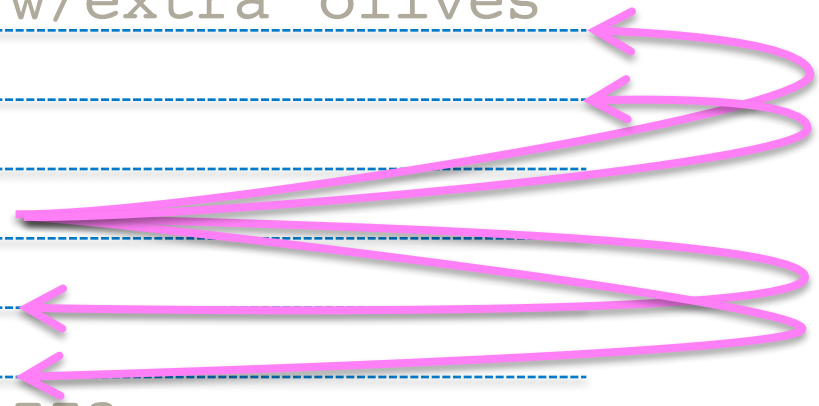
Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “4. @You Order Pizza”

...Necessary constraint, or Assumed constraint?

```
1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10.@Me eats pizza
```



Can order from home!

Can order on the way to pizzeria!

Can use pre-paid coupons, gift cards, or rewards cards!

Pizzeria may pre-make “standard” or “specials” pizzas!

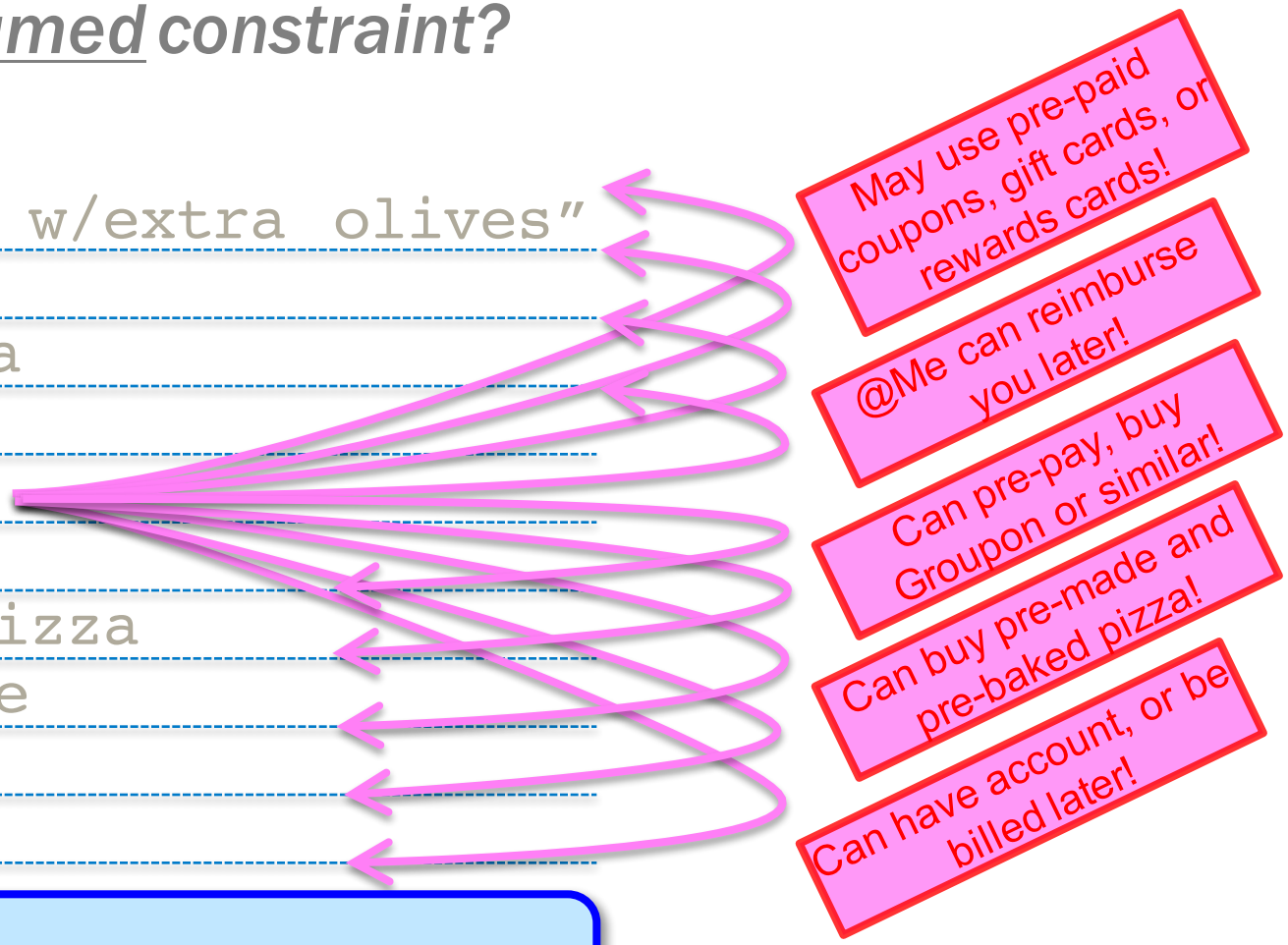
Why care about sequence?

- ONLY if you **get the wrong pizza**

On “5. @You Pay For Pizza”

...*Necessary* constraint, or *Assumed* constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



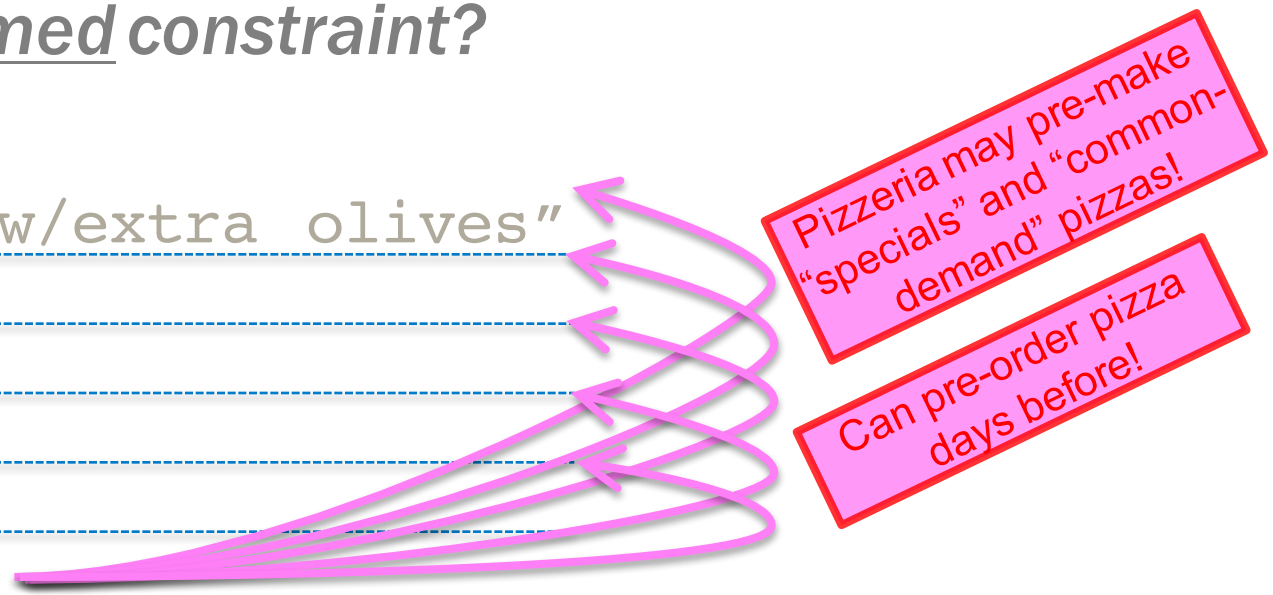
Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “6. @Pizzeria Makes Pizza”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. **@Pizzeria makes pizza**
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



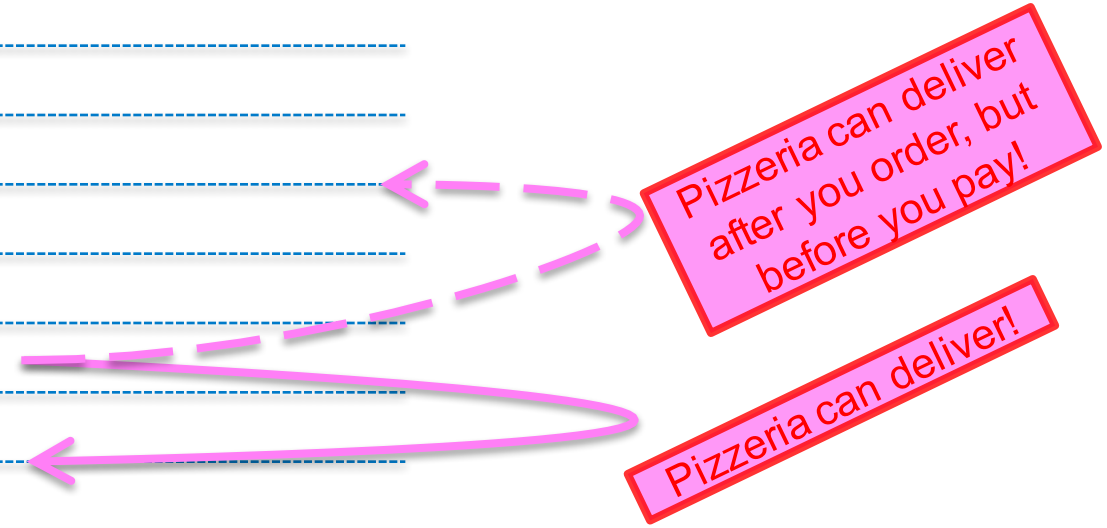
Why care about sequence?

- ONLY if you **get the wrong pizza**

On "7. @Pizzeria Gives You Pizza"

...Necessary constraint, or Assumed constraint?

```
1. @Me specify, "Supreme w/extra olives"
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10.@Me eats pizza
```



Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “8. @You Bring Pizza To Me”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

Pizzeria can deliver! (Remove instruction?)

Pizzeria can deliver! (Remove instruction?)

Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “9. @Me Has Pizza”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

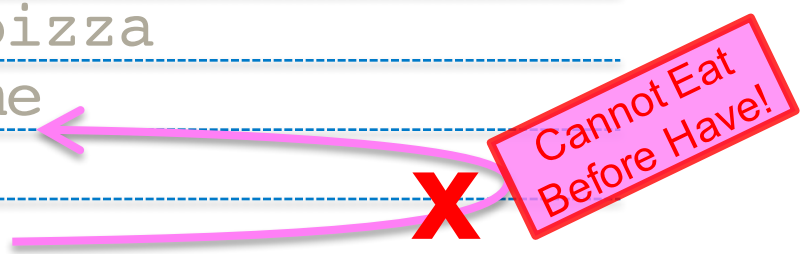
Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “10. @Me Eats Pizza”

...*Necessary* constraint, or *Assumed* constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



Why care about sequence?

- ONLY if you **get the wrong pizza**
- ONLY if **transaction success is jeopardized**

Scenario Review: We DID Consider...

- We considered the **algorithm**
  - The "**Goal**"
  - **Sequence-of-steps for how** the "Goal" is achieved
  - **Dependencies among instructions** within algorithm
    - Including **flexibility** for "**Instruction Re-Ordering**"

We focused on ONE algorithm!
There are an **infinite number** of **alternative approaches** (designs) to this problem!

- **In Considering "Dependencies"** among instructions:
  - Identified **corner-cases forced** us to **define-terms** regarding:
    - The **meaning of** specific **"state"**
    - The **meaning of** specific **"operations"** (or "activities")
  - "**What-if" logic** and scenarios **forced** us to **identify** "required" **instruction-order**

Algorithm requires:
- What "**state**"?
- What "**operations**" upon state?
- What "**dependencies**" among state-changes?

Scenario Review: We DID NOT (YET) Consider...

What About…

System "Tuning" might be factory-configured or user-configured

1. **Preferences** *(Optimization Bias)*
   - **How much** are we **willing to relax** some things **to get more** of another thing about which we care more?

(*example*), User-Latency requirements might be sacrificed in some "special circumstances"; Compatibility requirements might be sacrificed for some versions, or configurations

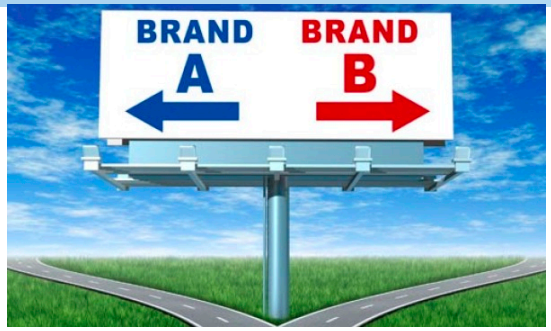
2. **Constraints**
   - **What "limits"** are we **willing to compromise** at some level?
     - (i.e., exploration of ***"soft-limits"*** and ***"hard-limits"***)

System is validated and controlled for characteristics that the user does not care about

3. **Over-Specifications**
   - What "**controls/limits**" are **unnecessary**? (We **mistakenly *ASSUMED***?)
     - What **ASSUMPTIONS restrict** our system from achieving more optimal/desired levels-of-performance?

What About "Preferences"?

For what do we optimize?



Can "Rank Order" to perform "Optimization Bias"!

Optimizations can influence *"selected-instruction-order"*!

- Prefer ***"Hot"***
  - Select pizzerias closer-to-home
  - Order after you get there (not ahead-of-time)
  - Select "Pick-Up", not Delivery
- Prefer ***"Good"***
  - Select pizzeria across town with better pizza
- Prefer ***"Cheap"***
  - Select low-cost pizzeria
  - Select pizzeria with "specials"
  - Select pizzeria for which we have coupons
  - Select pizzeria for which we have "rewards-card"
  - Select "Pick-Up", not "Delivery" *(so no $ for tip!)*
- Prefer ***"Fast"***
  - Select pizzeria closer-to-home
  - Select pizzeria that "pre-makes" pizza
  - Select pizzeria with faster production pipeline
- Prefer ***"support-specific-business"***
  - Select pizzeria recently started by your cousin Vinni
  - Select pizzeria employing the person with whom you are trying to get a date
- Prefer …?

What About “Constraints”?

How are our choices limited?

The Big Bang Theory (2007 -)

Constraints
can influence
“possible-instruction-order”!

- **Vehicle** available
  - Limits pizzeria selection based on distance, motivates Delivery not Pick-Up
- **Money** available
  - Limits pizzeria selection based on price-range, current-deals, on-hand coupons, frequent-buyer punch-card, Pick-Up not Delivery
- **When:** Day-of-week, Time-of-day, “The Big Game Day”
  - Limits pizzeria based on who is open, customer-load (e.g., long wait), etc.
- **Dietary restrictions**, preferences
  - Limits pizzerias to those with vegetarian/vegan, gluten-free, etc., or those offering the pizza type desired

Biggest Assumption: Over-Specification (1 of 2)

"Unwarranted Constraints"

- Scenario Purpose:
  - **@Me pay, but @Me not leave the house**
- Over-specification:
  - **@You will go get it**
- Actual Specification:
  - **@Me not leaving the house**
- Impact:
  - Should "Delivery" be considered (in addition to "Pick-Up")?
  - Should "Take-and-Bake" be considered?
  - Should we "Pre-Order" before The Big Day?
  - Should we have frozen-pizza "on-hand" for The Big Day?
  - Should we make pizza at home (from box, from scratch?)
  - Should friend just bring pizza when coming over?
  - Should you take a job at a pizzeria for free pizza?
  - Should you make friends with "Delivery Guy" who can get you all the free pizza you want?
- Other Over-specifications In This Scenario
  - @Me specify vs. @You specify pizza-type
  - @Me order vs. @You order
  - @Me pay, vs. @Me reimburse @You later (or @Me billed later by @Pizzeria)
  - Take&Bake pizza (provides flexibility regarding constraints "time", "hot")
  - Plan Ahead (have frozen pizza "on-hand")
  - …etc.

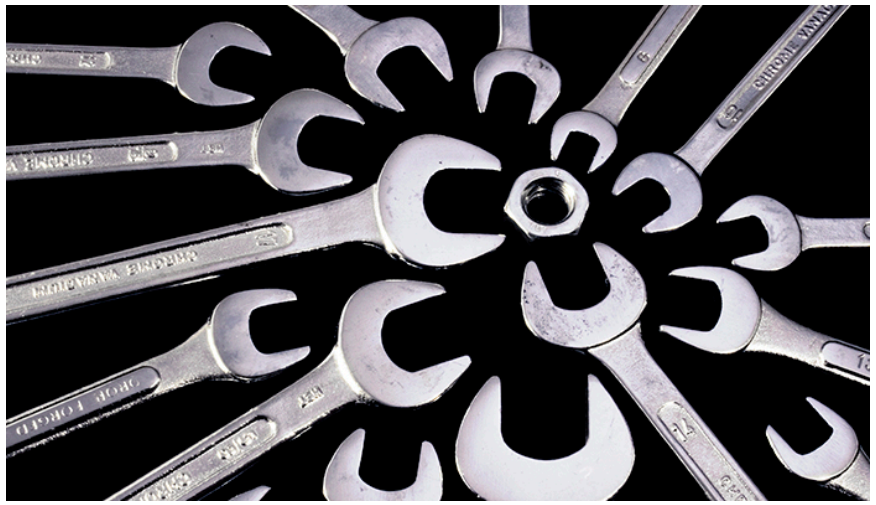
Any assumption unrelated to purpose is not a "real" specification/constraint!

Usually "YES", something else is worth considering!

Over-specifications are **identified** by (*continually*) re-visiting **"First Principles"** (i.e., the system's purpose)

Biggest Assumption: Over-Specification (2 of 2)

"Unwarranted Constraints"



Over-Specifications

- **Limit options**
  - For **no benefit**
  - For **negative benefit**
- **Confuse/Hide** the actual **constraints**!
  - **Harder to discover** elegant patterns
  - **Harder to reason** about (understand) behavior
- **Slow** our systems
  - by adding **unnecessary constraints**
- **Break** our systems
  - when unnecessary constraints are (eventually) violated

(*often*),
ENORMOUS Benefits
from relaxing constraints!

Over-Specification:
Erroneous constraints
should **always** be
identified, and removed!

- **Real constraints** – may be relaxed in some scenarios *(maybe)*
- **Erroneous constraints** – always provide *negative* value

Summary: Influences On Instruction Order (1 of 2)

- Instruction order influenced by:

1. **Requirements**: Sequence demanded through correct algorithmic execution
2. **Preferences**: Optimization biases expressing preferred tradeoffs
3. **Constraints**: Limitations based on what is possible
4. **Translation**: Communication friction where concepts are lost or changed during conversion
   - **REALITY** Hits!
   - Must translate to execute!

The same "System Requirements" can be expressed through different algorithms (each with "algorithm requirements")

May be compile-time, or runtime!

Watch for "over-specifications" ("Phantom Constraints")

Translation, and opportunities for misunderstandings ALWAYS EXIST!

OHZ NOZ! We still need to go through the compiler!

(Hidden) System Failure Scenarios:
Assuming "universal *concept* and *execution* agreement" among all parties.

Summary: Influences On Instruction Order (2 of 2)

- Instruction order influenced by:

1. **Requirements**: Sequence demanded through correct algorithmic execution
2. **Preferences**: Optimization biases expressing preferred tradeoffs
3. **Constraints**: Limitations based on what is possible
4. **Translation**: Communication friction where concepts are lost or changed during conversion

- **REALITY** Hits!
  - Must translate to execute!

Software Developer designs!

Compiler optimizations (speed/size/cross-platform, instruction set compatibility), sometimes by programmer design/algorithm, by system configuration

Mandated CPU cores, threads, execution units, or by system configuration

Many-stage Compiler translation to object code, CPU executes (pipelined) instructions through the CPU cache

Imperative vs. Sequential Devices

"Do This" vs. Control Structures (Necessary Order)

Programming Paradigms

C++ is Multi-Paradigm Language! Can do all of them!

- Adapted from: https://en.wikipedia.org/wiki/Comparison_of_programming_paradigms
  - **Imperative**: Statements directly change program state *(e.g., "Do This, then Do That")*
  - **Structured**: Style of Imperative with more logical structure
  - **Procedural**: Modular programming (procedure call), derived from Structured
  - **Functional**: Computation is evaluation of functions, avoiding state and mutable data
  - **Event-Driven, Time-Driven**: Program flow determined by events
  - **Object-Oriented**: Datafields treated as objects manipulated through methods
  - **Declarative**: Defines computation logic without defining control flow
  - **Automata-Based**: Programs are a model of a finite-state-machine or formal automata

Derived from

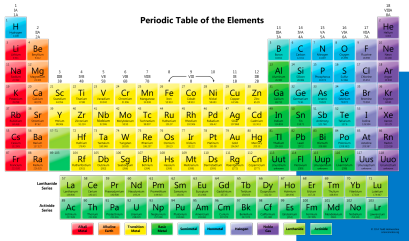
Imperative Devices are controlled through **Imperative** programming

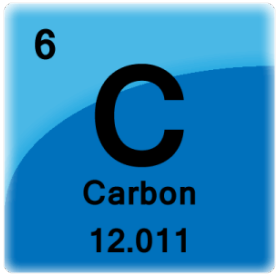
Instruction Order Is Crucial!

Sequential Devices are controlled through **Structured** or **Procedural** programming

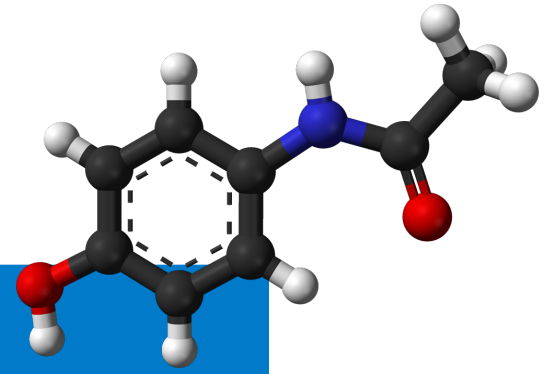
Instruction Dependencies Is Crucial!

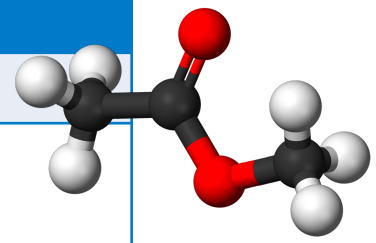
Contrast Imperative vs. Procedural

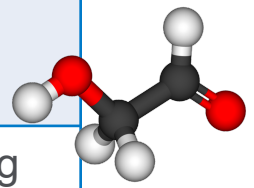


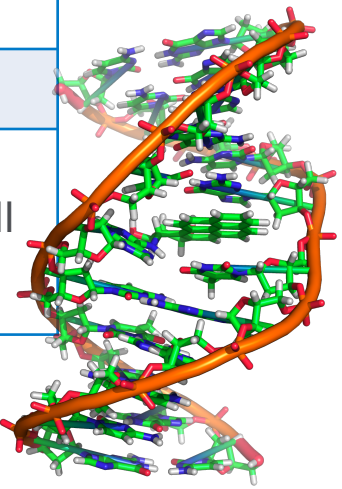


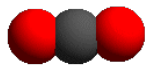
	Imperative: "Do This"	Procedural: Control Structures
Building Block	• **Instruction**	• **Control Structure**
Sequence Is Established By	• (Instruction) **Order**	• **Logical Dependencies**
Practical Application	• Interesting to explain **atomic-level reasoning**	• Interesting to explain **all life** and **processes we observe**
Metaphor	• **Atom** (association of indivisible smallest parts)	• **Molecule** (association of interacting compounds)
Explains	• **Table Of Elements**	• **All compounds that actually exist**
Basis For	• **Nuclear Chemistry** (including the most expensive way to "boil water" ever invented)	• **Electron Chemistry** (including all human-observable processes and all organic life)

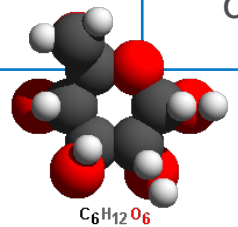


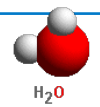


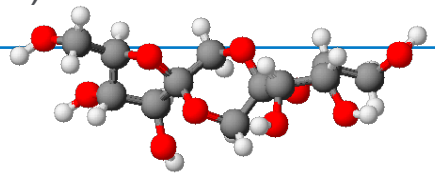






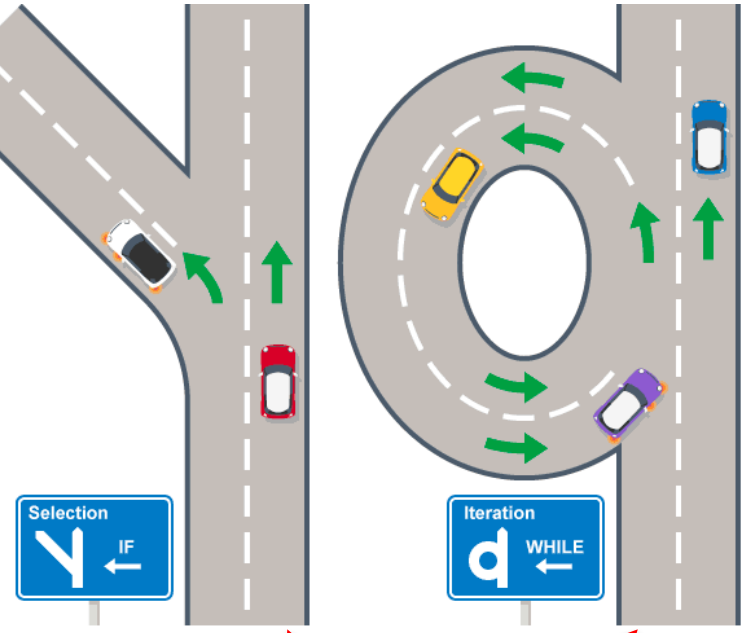






Sequence Of What?

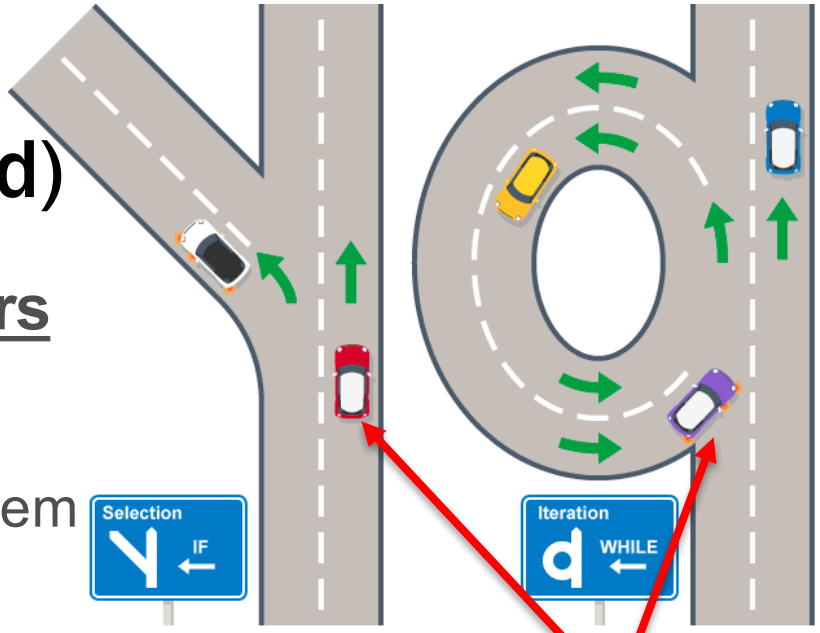
- Sequence of what?
  - ~~Sequence of instructions~~
  - ~~Sequence of data~~
  - Sequence of **dependencies**
- When we speak of, “Sequence”, are we talking about the “road” or the “cars-on-the-road”?
  - ~~Are we discussing the structure of our lines-of-code?~~
  - Are we discussing the **data-flow and logic-flow defined by the dependencies established** through our lines of code?
- Is “Sequence” the **road** (our lines-of-code), or the **cars** (that actually move/flow through our system)?



We **design and implement** our systems by defining and building the **roads** on which data and instructions will flow!

Sequence Is The Cars (Not The Road)

- We **Design the Road**, to (indirectly) **Influence the Cars**
- **The Road**: **Program Structure**, **Control Flow**
  - We write lines-of-code (we "build-the-road") to craft our system
  - Lines-of-code is NOT the "sequence"!
- **The Cars**: **Data-Flow**, **Instruction Sequence**
  - Data-and-Instructions "move/flow" to run our system (the cars)
  - Lines-of-code (the "road") is an INDIRECT INFLUENCE on "sequence"!



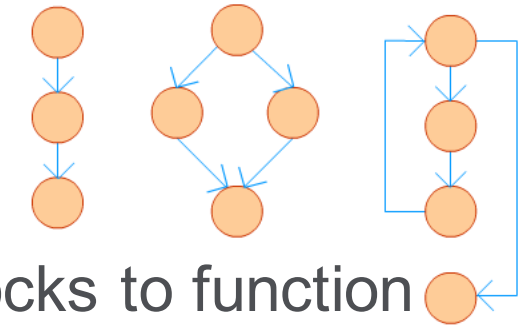
Sequence is the "Cars", which we (indirectly) control by defining the "Road"!

Can define any road you want!

Software Engineering establishes the structure (the roads) **to INDIRECTLY control data flow, logic flow** (the cars) that represent the processing we REALLY care about.

Algorithms Are Formed

- **Algorithms** are formed **entirely from building blocks**:
  - **Sequence** (ordered series of steps)
  - **Selection** (branched path, some not taken)
  - **Iteration** (repeat some instructions)



- **The "mechanisms"** that enable the building blocks to function (conditional values, counters) **establish the dependencies**
- We **ONLY care** about **"Sequence"**!
  - **Selection and Iteration** are merely convenient (logical utility) constructs to **assist** us **in defining sequence** (Turing-Complete algorithms)
  - Hard to otherwise **define an "infinite-loop"** sequence, or ***"select-among-option-set"*** sequence

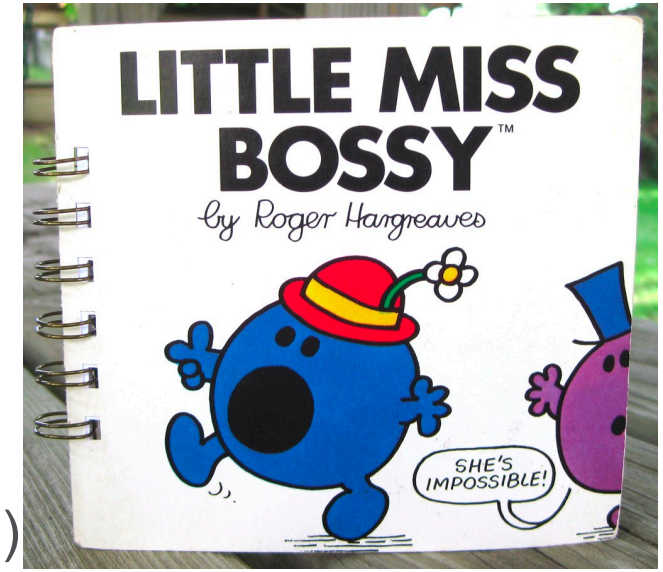
Algorithmic constructs provide efficient syntax to express sequence!

Sequence is defined by dependencies!

"Selection" and "Iteration" are merely constructs to assist in expressing (describing/defining) sequence!

Sequence is otherwise termed, **"Sequence of Dependencies"**
(NOT "sequence-of-data" nor "sequence-of-instructions")

Review: Imperative



- "**Imperative**" means we specify "**Do This!**", and it is done
  - **Minimal** (or no) **latitude**
  - **High control** (restriction) regarding behavior
  - **Very repeatable** (assuming no error, and no external state-changes)
  - *(Usually),* Highly **Inefficient**
    - You must "wait" until instruction is "done" before issuing next instruction
- Example:
  - Your **well-trained dog** is an imperative device
- Interesting Applications:
  - **Real-Time Operating Systems (RTOS)** demand exact control of each instruction cycle, and each resource allocation

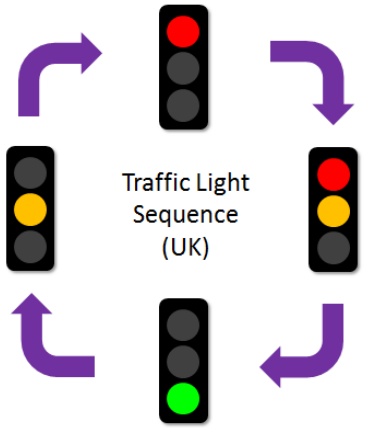
Today's CPUs don't work like this! **Superscalar** CPUs use **"pipelines"!**

Review: Sequential

- "**Sequential**" assembles **control-structures based on dependencies**
  - Can **reason about behavior** (order is "defined" by logical dependencies)
  - Can **reason about mutations** (e.g., "state-machine")
  - Can be **Turing-complete** (e.g., "Turing Trains" is Turing Complete)
    - Turing Complete: System is able to perform any calculation, given enough resources. *(Non-Turing-Complete systems are unable to handle a specific set of calculations, even with enough resources.)*
    - Turing Complete means algorithm can be expressed (whether-or-not it is performed)

We rely upon logical control structures!

- Example:
  - **Airplane "Auto-Pilot"** is a sequential device *(tends toward stable state by managing internal and external dependencies through well-defined algorithm(s))*
- Interesting Applications:
  - **Most real-world** Software Engineering **systems, system-integrations**



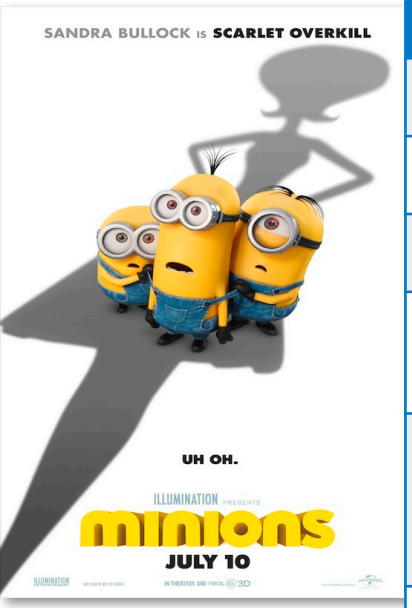
Imperative vs. Sequential Device

Note: Today's Distributed Web architectures scale through "*Agents*", not "*Minions*"

Dave

Agent Smith

Imperative: "Do This"	Sequential: Logical Steps
• Is "**Task-based**"	• Can be "**Goal-based**"
• Establishes "**next-step**" (only)	• Defines **logical procedure** (algorithm)
• **Oblivious** to logical dependencies	• **Relies upon logical dependencies**
• **Is Limited**: Requires constant (frequent) supervision and control	• **Can be used** as **organizational structure** (to build/scale systems)
• **Hard to work with**: Must specify many seemingly irrelevant details to address unexpected/surprising behavior	• **Easy to work with**: Operates with latitude within established constraints
• **CANNOT** scale	• **CAN** scale

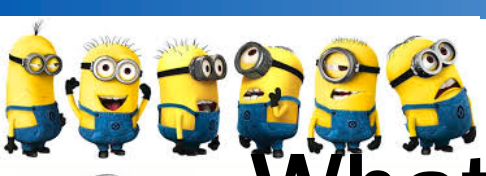


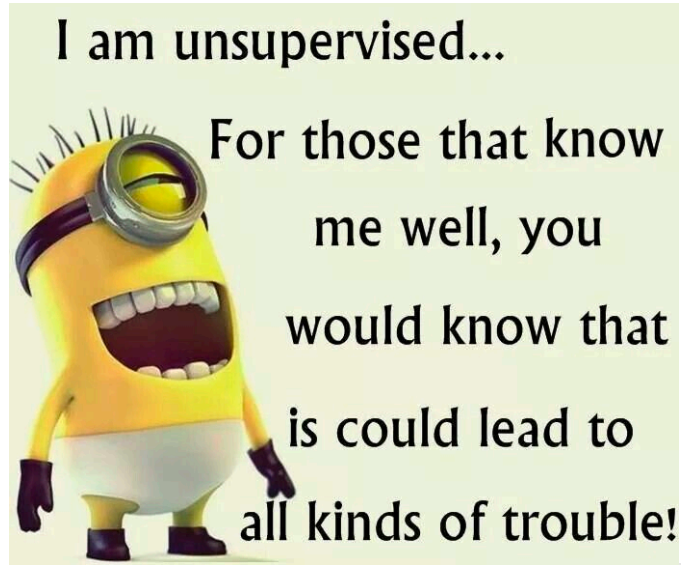
Minions (2015)

The Matrix Reloaded (2003)

What It's Like To Work With Imperative Devices

- You must:
  - **Queue all future tasks** *(the device will not)*
  - **Perform all scheduling** *(the device will not)*
  - **Identify all dependencies** *(the device will not)*
  - **Manage all dependencies** *(the device will not)*
  - **Identify opportunities** for efficiencies *(the device will not)*
  - **Respond to unexpected** external or system events *(the device will not)*
  - **Concurrently interface** with other components/systems *(the device will not)*
  - **Continually establish further constraints** when the device exhibits surprising behavior





Just shoot me now.

Exceptions reset to "baseline" state. There is no baseline state (chaos).

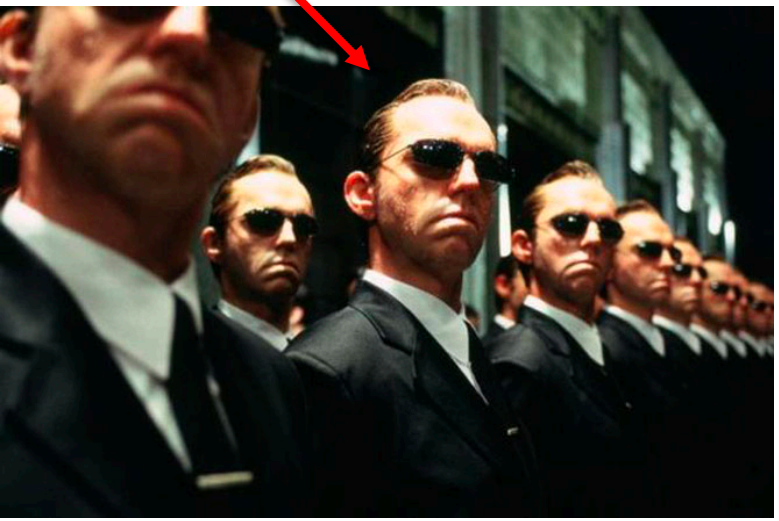
No context exists to throw exception, because all behavior is "custom one-off"

What It's Like To Work With Sequential Devices

- You must:
  - **Establish algorithm** based on **logical dependencies**
  - **Establish objectives/priorities** to guide device execution
  - **Establish constraints** for device behavior
  - **Establish rules** for **concurrent interfacing** with external components/systems
  - *That's it!*

Stable monitoring, idle mode

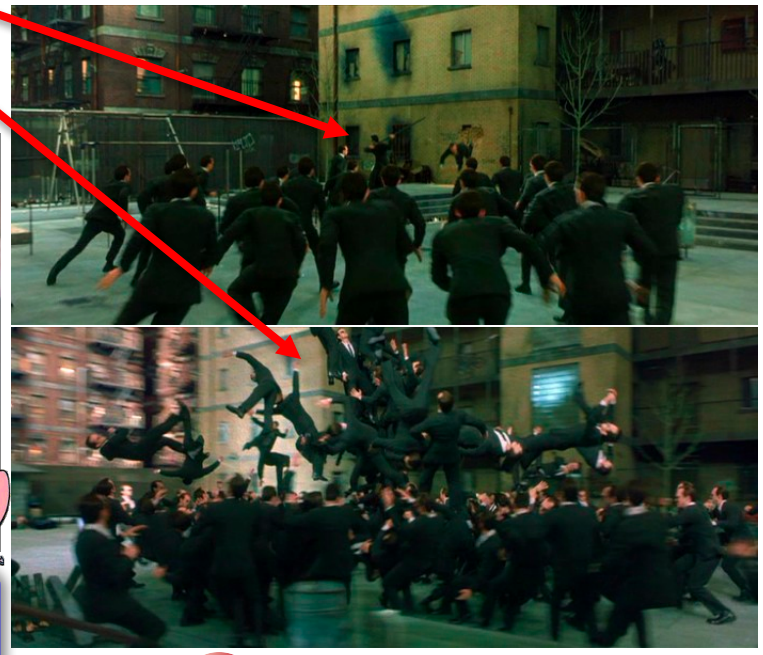
Self-organizing



Variable load-response based on internal metrics

Goal-oriented

Exception is thrown



Taking over the world is far easier.

Pinky & the Brain (1993-1999)

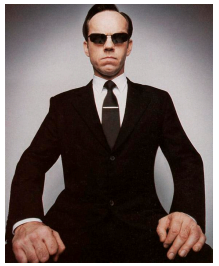
Contrasting Imperative vs. Sequential Devices

- **Imperative** devices:
  - Are "**Simple**": **Easy to create**, and to **understand**
  - At Scale: Approximate **Chaos**
    - Logical dependencies **not respected**
    - **Must tightly supervise interactions** among components that do not respect (*logical*) dependencies
  - **Failure to Supervise: Chaos** *(possibly death – yours or theirs)*

- **Sequential** devices:
  - Are "**Higher-Order**" devices **bounded by logical dependencies** (*algorithm*)
  - May be **"simple"**, **or "rich"** in behavior
  - **Scale well**
  - **Failure to Supervise: Predictable Results**, can tend towards "order"

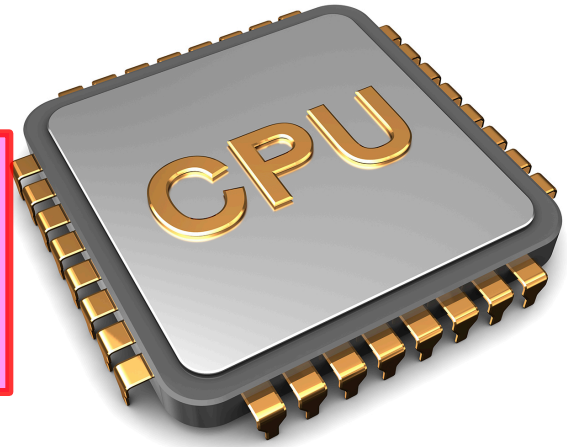
Goal: Is **easier to scale systems with Agents** that follow (established) procedure, versus Minions that must be constantly supervised. (*Chaos vs. Predictable Results*)

Minions (*imperative*) might be fun for entertainment value, and because they are unlikely to threaten you with collective bargaining; but **if you want to get work done** (**efficiently** and **correctly**), and **at scale**, then use Agents (*sequential*).



The CPU Is A Sequential Device

The Modern Super-Scalar Multi-Core CPU Is *Awesome*.



- Very rich **internal technologies**, self-managed
  - Multi-level **CPU cache** (loading, eviction, coherency)
  - **Superscalar CPU Instruction Pipeline** (many instructions started/retired per cycle)
  - **Speculative Execution** (instructions "eagerly" executed, might be aborted, re-tried, or results disregarded)
  - **Rich Microcode** (firmware updates, other optimizations)
  - …and *much more*!

CPU is very rich **Sequential Device**! (Not Imperative!)

Today's CPU will **trace internal** and **external dependencies** to

- parallelize execution, pre-fetch data, and speculatively execute instructions,
- while managing cache coherency *(dependencies!)* across threads, processes, and cores,
- while managing internal metrics for data access patterns and cache hits,
- to constantly optimize *(at runtime!)* based on a given execution sequence, data set, or runtime-observed system load characteristics.

Physical vs. Logical Sequences

What *actually* happened, vs. what you *assumed* happened

Physical vs. Logical Sequence

- **Exactly TWO** kinds of sequence exist:

1 Physical Sequence defined by **order** in which instructions "occur"

2 Logical Sequence defined by **logical dependencies** in an algorithm

- *"Occur"*
  - Are **specified** in source file?
  - Are **serialized** in object code?
  - Are **presented** through the Program Counter?
  - Are "**started**" (in the CPU)?
  - Are "**completed**" (in the CPU)?
  - **Results** are **propagated/written**?

Physical Sequence is ALWAYS ambiguous!

- *"Logical dependencies"*
  - Are **EXPLICIT** based on:
    - "**Well-Defined Behavior**"
      - ...those logic (control) structures for which the programming language ***EXERTS CONTROL***

Logical Sequence is ALWAYS Well-Defined!

Example 1: Physical Sequence

- **Source Code**: A **serialized form** of **physical sequences**

```
void MyTable::highlightRowCol(int mouse_x, int mouse_y)
{
  int row_index = computeRowIndex(mouse_y);
  int col_index = computeColIndex(mouse_x);

  bool is_row_locked = isRowLocked(row_index);
  bool is_col_locked = isColLocked(col_index);

  Color row_color = (is_row_locked) ? color_locked_ : color_unlocked_;
  Color col_color = (is_col_locked) ? color_locked_ : color_unlocked_;

  highlightRow_(row_index, row_color);
  highlightCol_(col_index, col_color);
}
```

- *(Erroneous)* **ASSUMPTION**: This is execution order.
  - ***Order appears*** to be ***explicit***
  - ***Order is tool*** we use to reason (so *some* guarantees *must* be present)
  - Assumption is reinforced by ***how programming is taught***, and *(misunderstanding)* **programming language rules**.

NOT true!

BOLD assertion (defended on next slide)

- *Given:*

```
class Color { /*...*/ };

class MyTable
{
private:
  Color color_locked_;
  Color color_unlocked_;
  // ...
private:
  void highlightCol_(int col_index, Color color);
  void highlightRow_(int row_index, Color color);
public:
  // ...
  int computeColIndex(int mouse_x) const;
  int computeRowIndex(int mouse_x) const;

  void highlightRowCol(int mouse_x, int mouse_y);

  bool isColLocked(int col_index) const;
  bool isRowLocked(int row_index) const;
  // ...
};
```

Misunderstanding Programming Language Rules

- **ALL** programming languages have **rules**
  - **Syntax and semantics**
  - **Well-defined behavior**
- **ALL** rules establish **logical dependencies**
  - **Are how we reason** in that language!
  - Is **basis** for **Well-Defined Behavior** in that language!
- **ALL** programming languages are **SILENT on physical sequence**
  - Is **"Undefined Behavior" (UB)**
  - Is that which the **language CANNOT control** (e.g., CPU, memory access latencies)

Leads to idioms, patterns, conventions

Are the mechanisms that make the language work!

- **NO programming language** mandates behavior for **physical sequence**, because that is **NOT POSSIBLE** (is not how compilers and CPUs work).
- If a programming language wants to **EXERT CONTROL**, it establishes **Logical Dependencies** (through which **Logical Sequences** can be expressed.)

ALL programming languages are designed to ACTUALLY execute on a CPU!

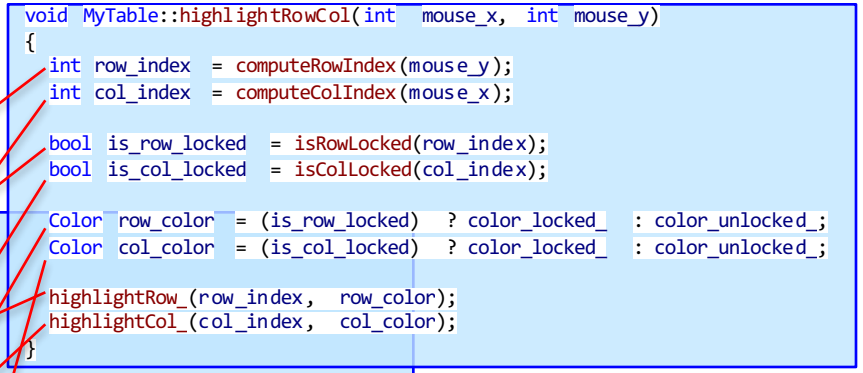
Revisit Example 1: Is Two Logical Sequences

(Previous)

In this case, logical sequences are totally unrelated!

```
void MyTable::highlightRowCol(int mouse_x, int mouse_y)
{
  int row_index = computeRowIndex(mouse_y);
  bool is_row_locked = isRowLocked(row_index);
  Color row_color = (is_row_locked) ? color_locked_ : color_unlocked_;
  highlightRow_(row_index, row_color);

  int col_index = computeColIndex(mouse_x);
  bool is_col_locked = isColLocked(col_index);
  Color col_color = (is_col_locked) ? color_locked_ : color_unlocked_;
  highlightCol_(col_index, col_color);
}
```



- **By tracing Logical Dependencies** (by tracing C++ Well-Defined Behavior)
  - **We Discover** (in this case): Two logical sequences are present!
  - These **separate logical sequences** have NOTHING to do with each other!
    - In this case, **could** even **be separate functions**!

Example 2: Physical Sequence

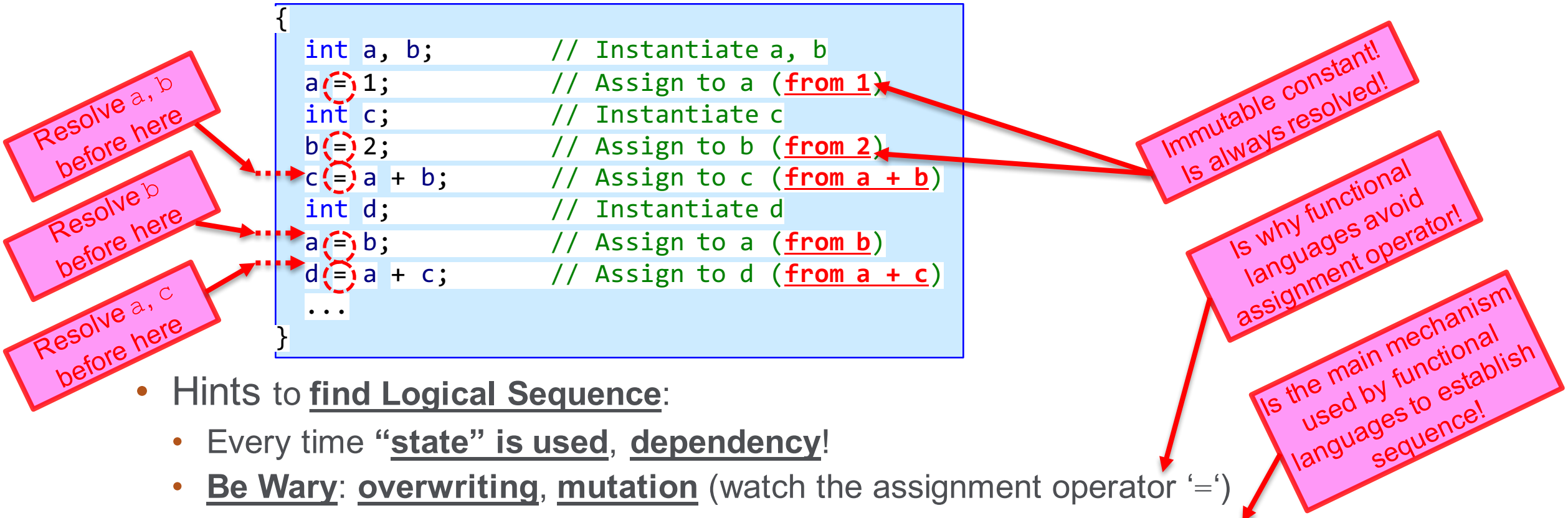
- **How** to **identify logical sequence(s)**?

```
{
  int a, b;          // Instantiate a, b
  a = 1;             // Assign to a (from 1)
  int c;             // Instantiate c
  b = 2;             // Assign to b (from 2)
  c = a + b;         // Assign to c (from a + b)
  int d;             // Instantiate d
  a = b;             // Assign to a (from b)
  d = a + c;         // Assign to d (from a + c)
  ...
}
```

- A **more abstract** (perhaps "real-world") **example**
  - Code seems …*indirect*
  - Variable names are …*vague*
  - Comments are …*unhelpful*

Example 2: Logical Sequence (inferring)

- **Logical Sequence** can **be inferred** from physical sequence (using programming language "Well-Defined" behavior)



- Hints to **find Logical Sequence**:
  - Every time **"state" is used**, **dependency**!
  - **Be Wary**: **overwriting**, **mutation** (watch the assignment operator '=')
  - **Dependency is implied** through **"nested-expansion"** (inputs to expressions, function-calls)

State: Hidden Dependencies!

- "**State**" is "**tricky**"
  - **WHEN** did you use it? Was it **CORRECT**?
    - Was it "**stale**"? *(too "old")*
    - Was it "**corrupted**"? *(update was in-progress)*
    - Was it "**too-new**"? *(race condition)*
- ***Two*** issues exist:
  1. **Compute State** *(e.g., expressions with operands)*
     - Were **operand values** "correct"?
  2. **Change State** *(e.g., overwrite/mutate previous state)*
     - **WHEN** did that occur? *(stale, corrupted, too-new?)*

`const` (read-only) makes this **simple**! (Value is always correct!)

Not too hard

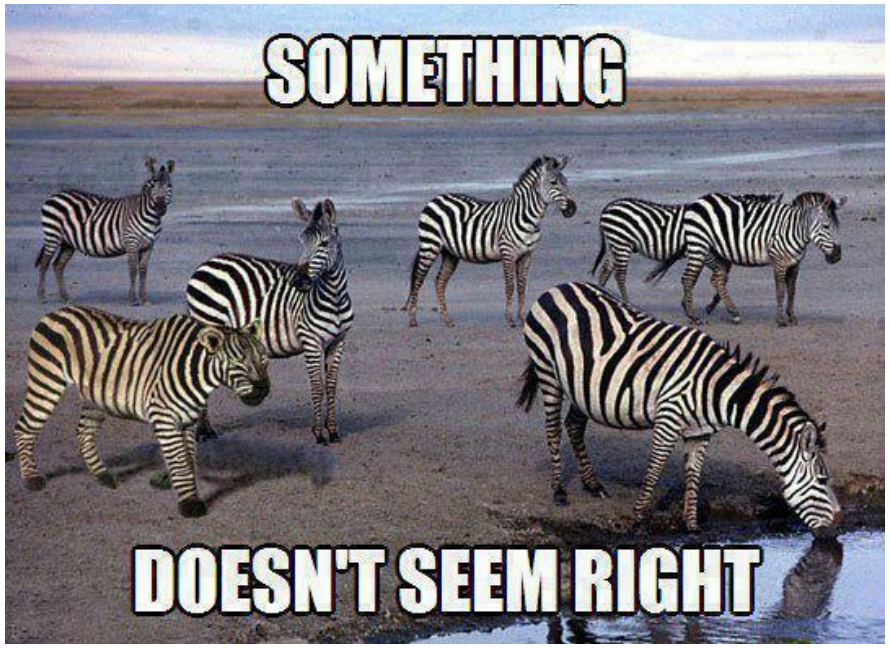
Much Trickier!

Anytime you **use state**, there **might be** hidden **dependencies**! So,
"Use state with care!"

Assignment '=' Is A Change In State

- Assignment `operator=()`: Tricky!
  1. **Dependencies** to compute **RHS**
  2. **Overwrite LHS** (Dependencies for WHEN did that occur relative to other reads/writes?)
- **Functional** programming **avoids** `operator=()`
  - **Avoids (2)**, but **still has (1)**
  - **Addresses (1)** by **functionally computing** all values from "First Principles"

    (i.e., "State Is Evil")



Functional Programming Mantra:

"State Is Evil,

all values are computed from First Principles"

“Resolving” Dependencies

- **State** (values)
  1. **Constant**: `42`
     - Easy! Done at compile!
  2. **Computed**: `a * b`
     ① Must complete evaluation before is used (i.e., “RHS”)
  3. **Overwritten**: `c = a * b`
     ① Must complete evaluation of RHS
     ② Must overwrite LHS
  4. **Mutated**: `c += a * b`
     ① Must resolve LHS
     ② Must resolve “delta” (e.g., “RHS”)
     ③ Must overwrite LHS

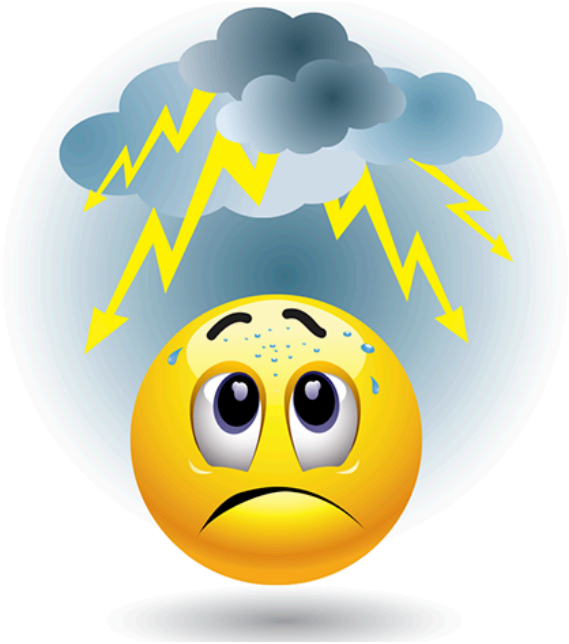
- **Side-Effects** *(tricky!)*
  1. Resolve “**before**” side-effect “**starts**”
  2. Resolve “**before**” side-effect “**completes**”
  3. Resolve “**after**” side-effect “**completes**”

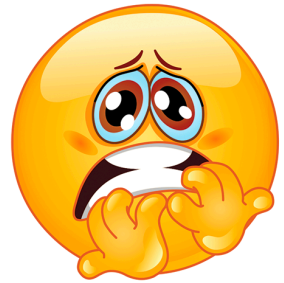
Functional Programming (tries to) avoid these!

Reasoning about dependencies is actually **pretty simple**

In practice,
Complexity increases (non-linearly) with a greater number of dependencies

State: "Nervous Meter"





Care Free!

No data object	`const` data object	Computed data object	Overwritten data object	Mutated data object

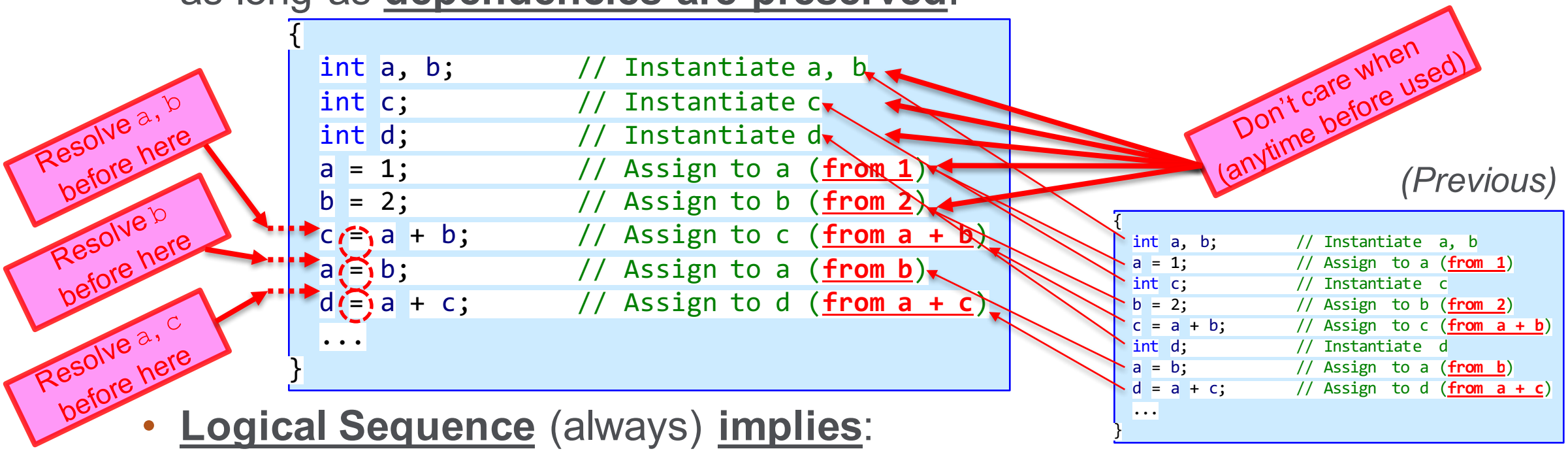
Very Nervous!

Functional Programming

All Programming

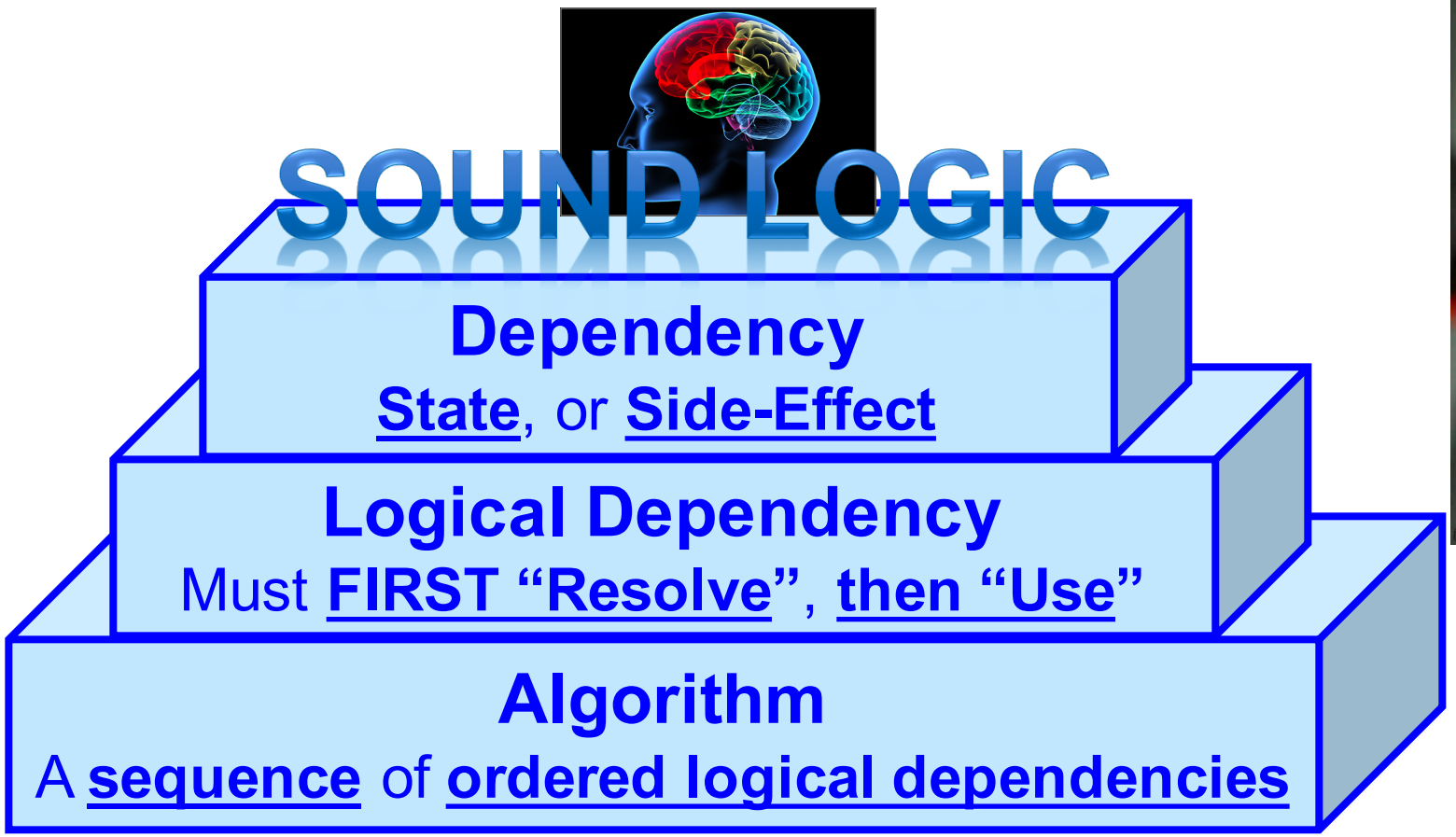
Example 2: Logical Sequence (preserving)

- **Logical sequence survives** *(arbitrary!)* physical sequence "**shuffling**", as long as **dependencies are preserved**.



- **Logical Sequence** (always) **implies**:
  - **State** must be "**resolved" before is used**
  - "**Zones" of *"Don't Care!"*** exist – we do not care about **physical** sequence order

Concept Review: Logical Sequence



Rule: **Any order** that **respects logical dependencies** is **logically equivalent**

Concept Review: Physical Sequence

- **Physical** Sequence: **Don't care**.
  - Is **arbitrary** and weird
  - **Moves around** for seemingly no reason whatsoever
    - At **compile-time**, at **runtime**
- When you hear the word "Sequence", IMMEDIATELY ask, "Logical Sequence"?
  - Because if it's **not a logical sequence**, **we don't care**.

Parks And Recreation (2009-2015)

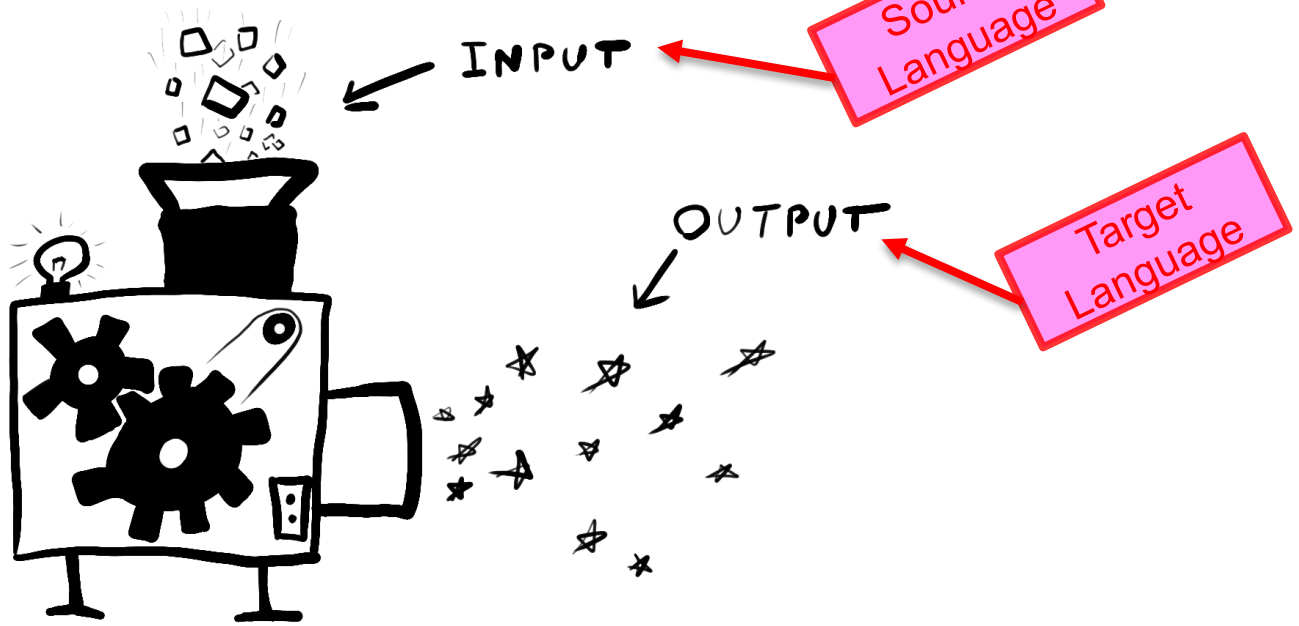
Physical Sequence:
No dependencies exist, so
order CANNOT be enforced.

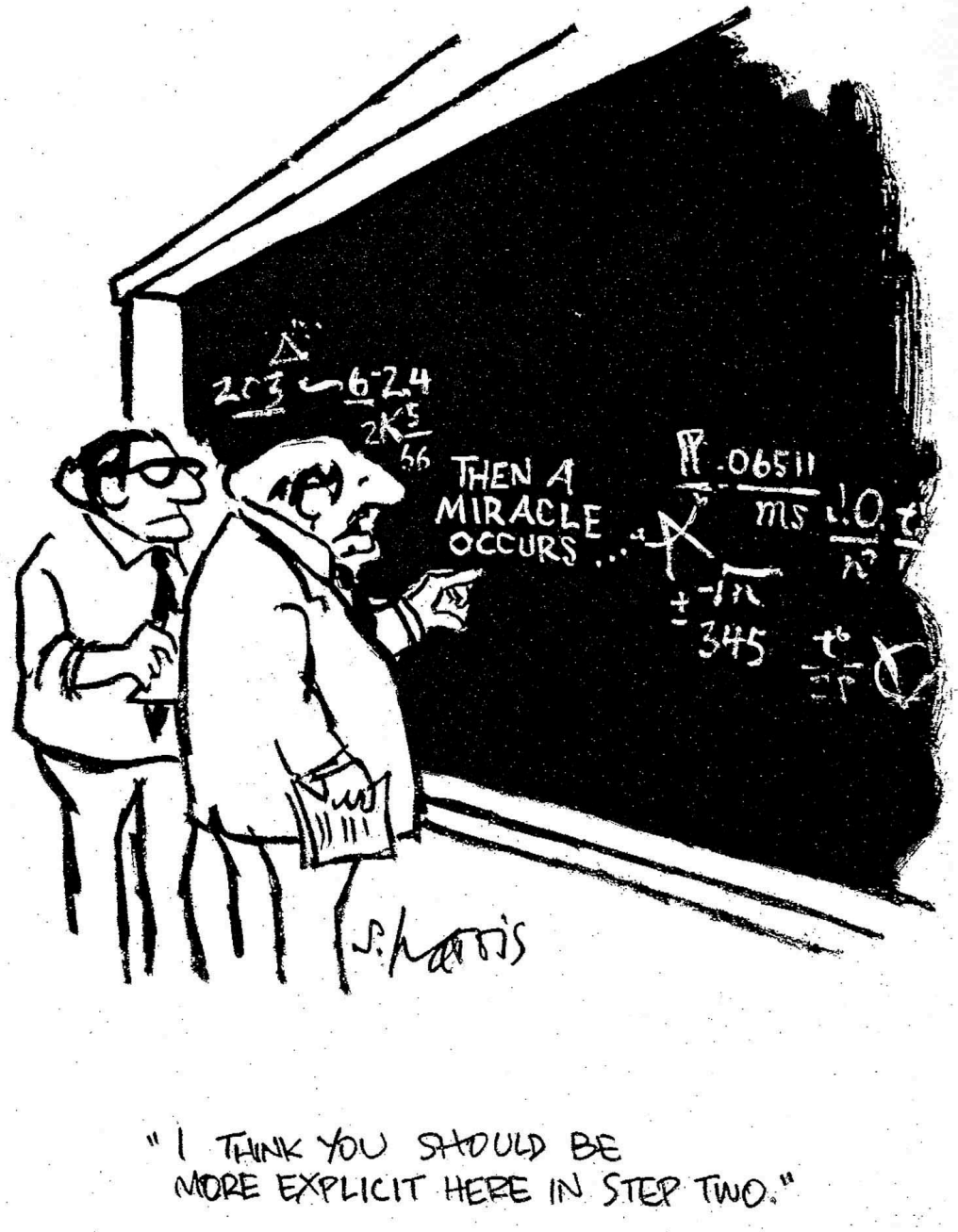
Compiler Internals: Optimizing Logical Dependencies

*Compiling **is** optimizing.*

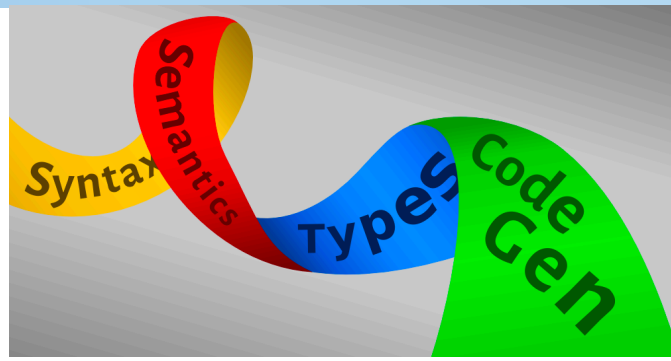
Compiling

- Compiler:
  - **Reads one language**
    (e.g., C++ source code)
  - **Writes another language**
    (e.g., CPU-specific machine code)





Compilation ASSUMPTION



- **ASSUMPTION** from Compile:
  - **Algorithm is SEMANTICALLY EQUIVALENT** ("unchanged") **between Source Language and Target Language** (e.g., algorithmic assumptions remain "in-tact")
- **REALITY**:
  - **Are different languages!** *(So semantics are different!)*
- Different languages have different…
  - **Priorities**
    - Different languages are optimized for different priorities, behavior
  - **Constraints / Limitations**
    - *(example):* Machine Language bounded by reality of architectural registers and instruction set within CPU
  - **Opportunities**
    - *(example):* CPU SIMD instructions allow multiple source code statements to be "collapsed" into a single (optimized) CPU instruction

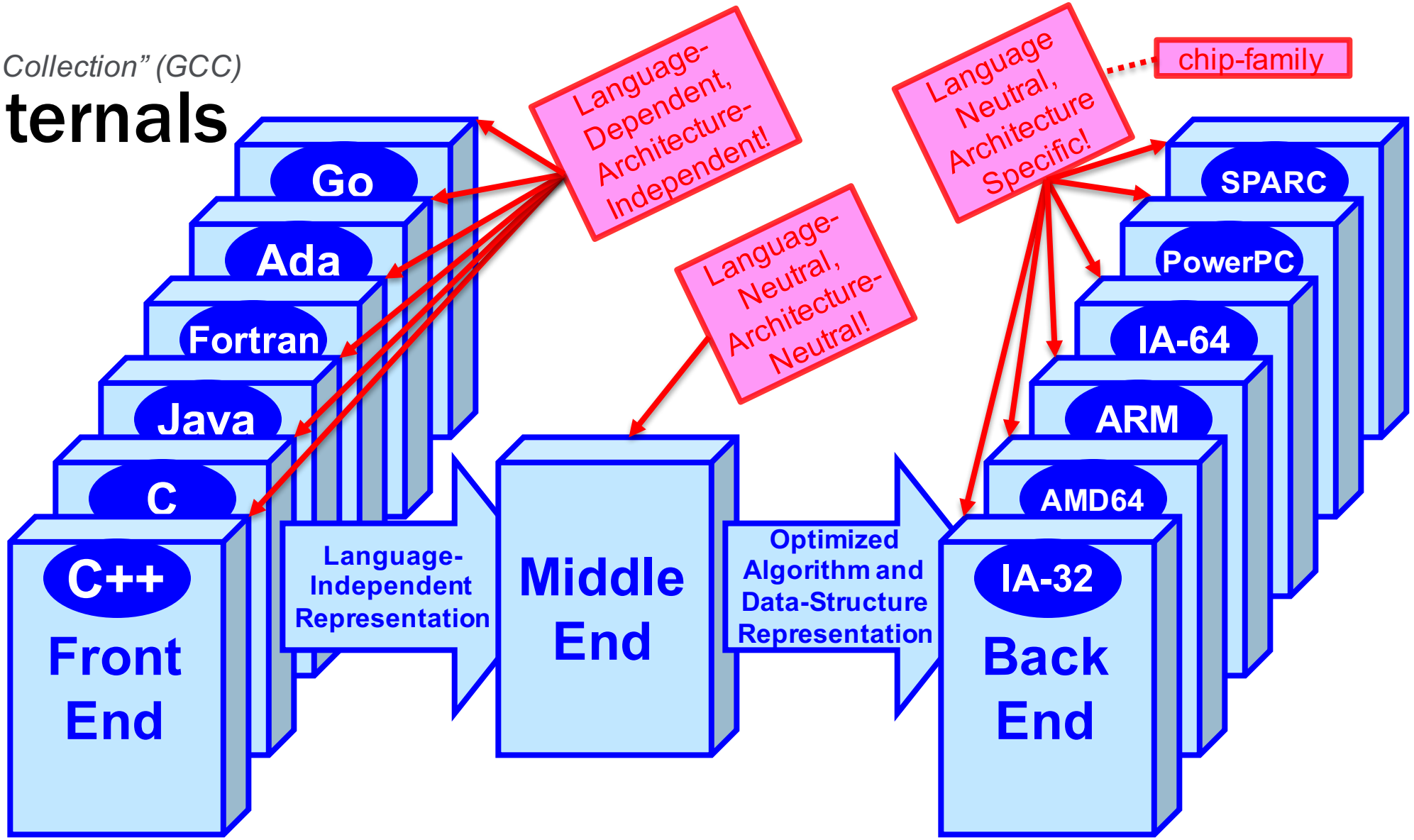
A compiler *never* produces an executable **identical** to your C++ source code *(it produces an **equivalent** program that's a lot better)*

"Gnu Compiler Collection" (GCC)

GCC Internals

Software Engineering Radio Episode 61: Internals of GCC: *(06-July-2007),* Morgan Deters discusses behind-the-scenes look at compilers and their inner workings using GCC as an example.
http://www.se-radio.net/2007/07/episode-61-internals-of-gcc/

In Memoriam:
Dr. Morgan G. Deters
16-Apr-1979 – 17-Jan-2015

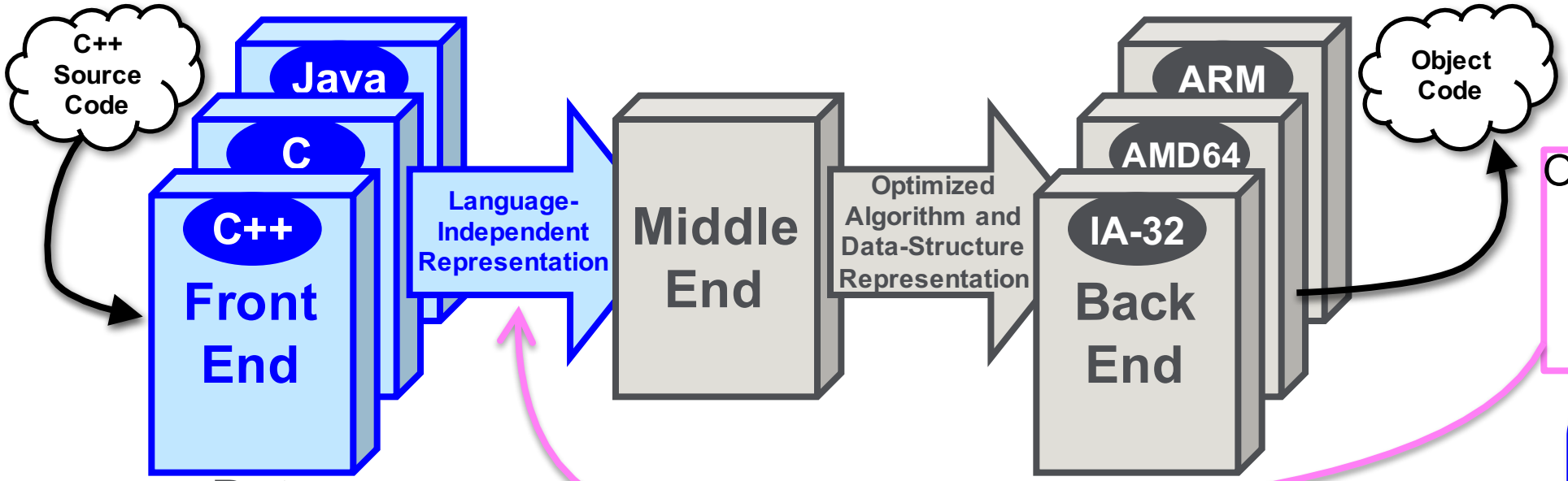


Goal: Accept algorithm with **language-dependent semantics**

Goal: **Efficient algorithm and data structure representation** (in space and runtime)

Goal: **Produce architecture-specific algorithm** (coupled to registers, instruction set, memory layout)

GCC Internals: "Front End"



Output:

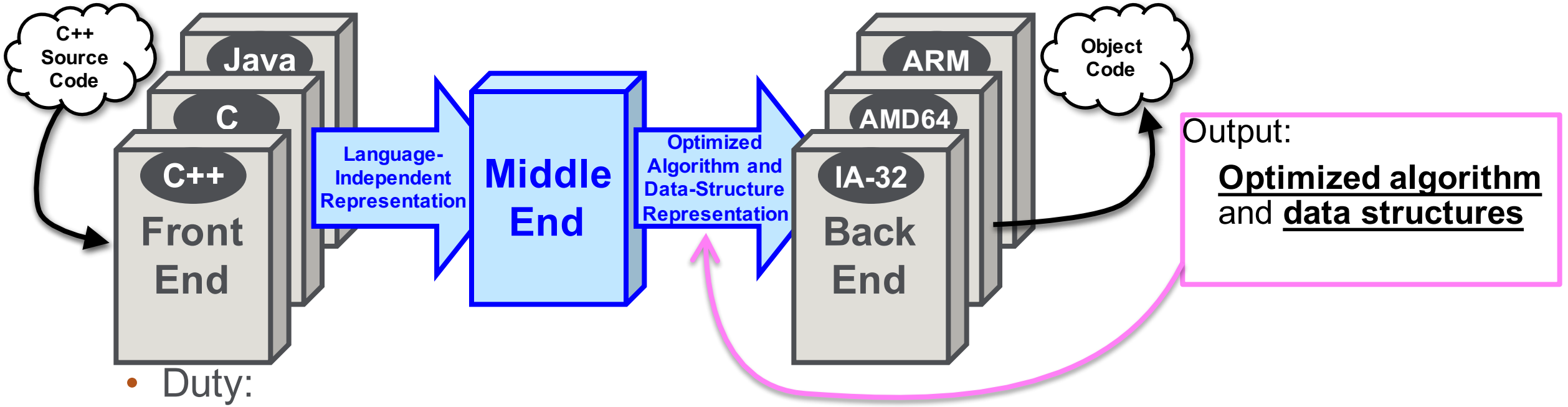
Abstract Syntax Tree (AST) of expressions and composite types

"Front End" actually **knows very little about what program does**!

- Duty:
  - **Lexical and semantic analysis** (specific to source code language rules)
  - **Validates syntactic structure** of input program
  - Some languages require "**many-passes**" (Java), C and C++ are (mostly) "one-pass"
  - **Emits diagnostics** about language conformance
  - **Creates internal data structures** for composite types, debugging information
  - **Builds initial AST** representation

many iterations

GCC Internals: "Middle End"



- Duty:
  - From AST produced by "Front End", **analyzes and transforms** the program
    - **Optimize Speed**: Object code should run as fast as possible
    - **Optimize Size**: Object code should take as little space (or resources) as possible

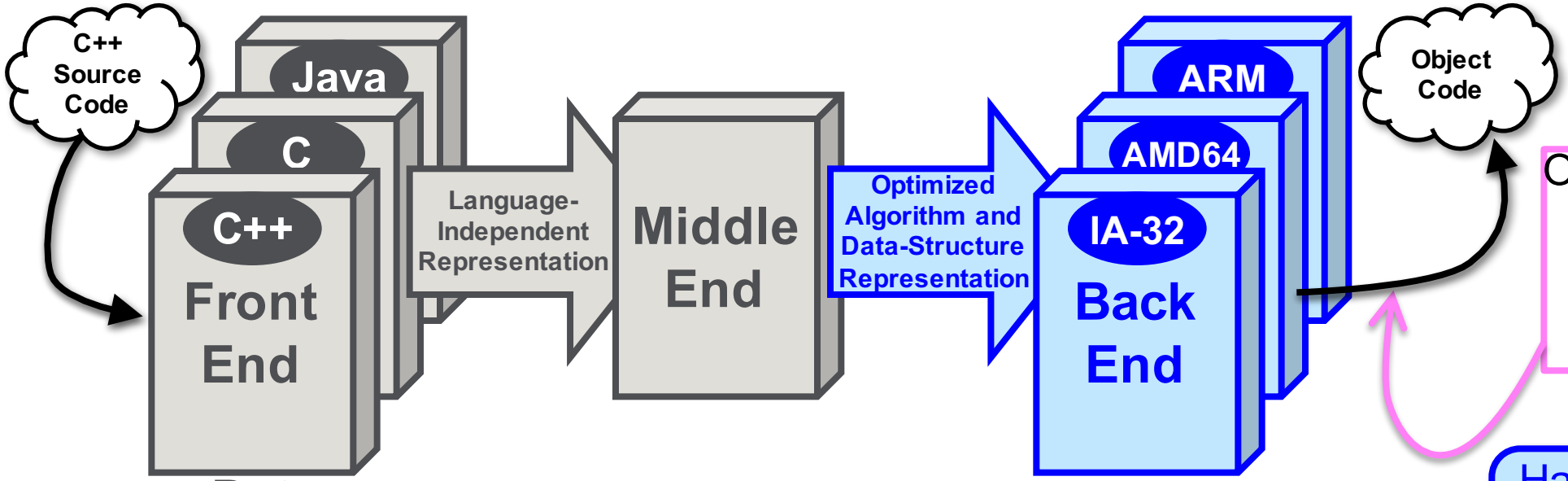
many iterations

Many dozens of passes going over AST and finding patterns!

Optimization is possible
because "understands" program
- **Control-Flow** analysis
- **Data-Flow** analysis

Optimizations are machine and target independent!

GCC Internals: "Back End"



Some compilers (including gcc) generate assembly code which is then fed into assembler for object code generation

Output:

Final translation of **machine code for target architecture**

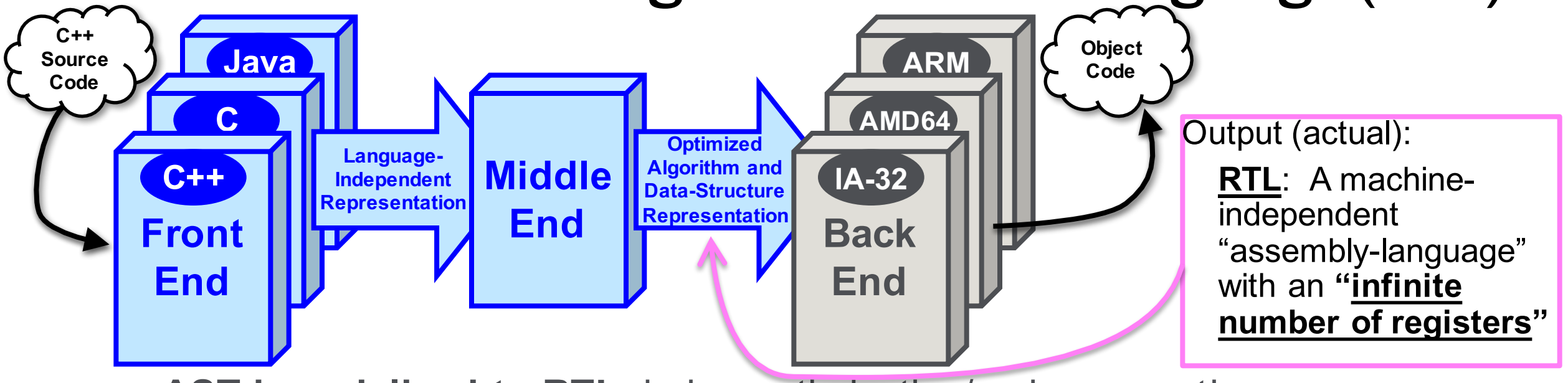
- Duty:
  - **Final mapping to object code** for target architecture
  - **Applies transformations** that take advantage of target architecture
    - **Register allocation** (maximizes the amount of program variables that are assigned to hardware registers instead of memory)
    - **Code scheduling** (instruction stream order)
    - Leverage super-scalar features of modern CPUs to **rearrange instructions** (so multiple instructions are simultaneously in different stages of execution)

many iterations

Has **detailed knowledge about the hardware** on which program runs

Linker collects all object files and libraries to build final executable

GCC Internals: Register Transfer Language (RTL)



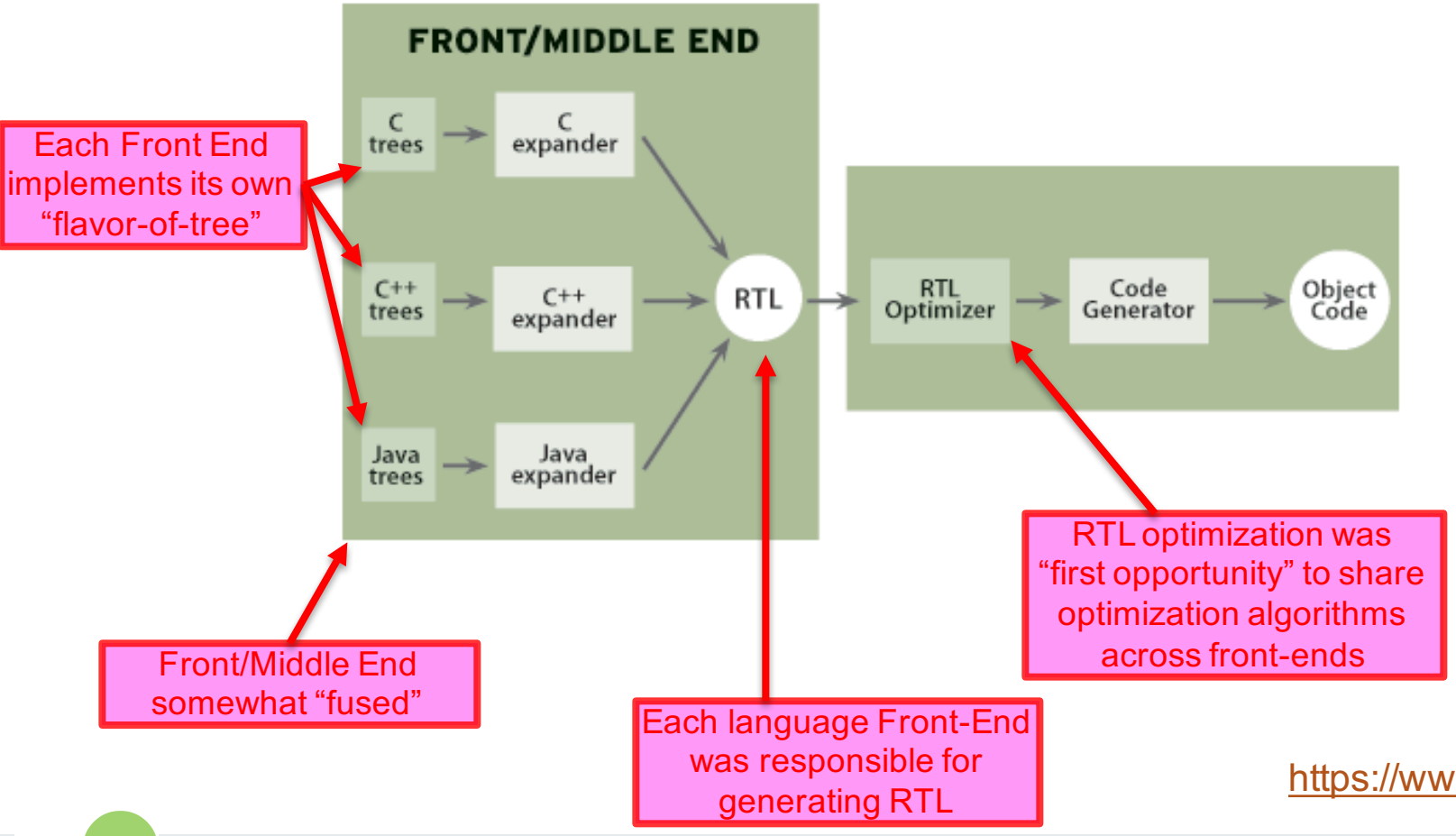
- **AST is serialized to RTL** during optimization/code-generation
  - **AST is a "stepping-stone"** to RTL
  - **Most optimizations occur in RTL** because RTL is programming-language independent (some optimizations possible in language-specific Front End, but not preferred)
  - **RTL optimization is in the "Back End"** with specific knowledge of target architecture

many iterations

RTL Enables:
- **Low-level optimizations**
- **Object-code generation**

GCC Internals: Investment (Red Hat & Community), ~2005

- **GCC 3.4** (*before* investment by Red Hat and GCC community, ~2005)



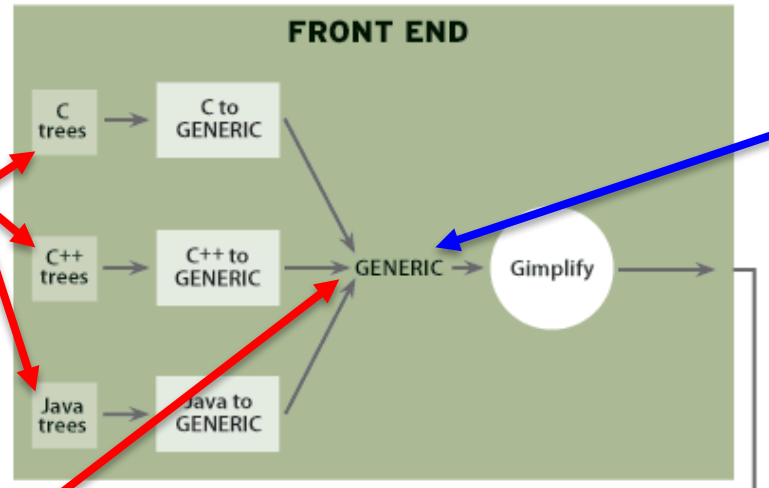
Desired Improvement:

- **RTL is "too-far-removed"** from source-code context **for some analyses and transformations**
- Want **language-independent representation so optimizations can be shared** across Front Ends, at higher-level context than provided by RTL

https://www.redhat.com/magazine/002dec04/features/gcc/
Diego Novillo

GCC Internals: Investment (Red Hat & Community), ~2005

- **GCC 4.0+** (*after* investment by Red Hat and GCC community, ~2005)



Trees contain detailed information about data types, variables, and control structures

Enables exploitation of common "high-level" optimization algorithms across Front Ends

C/C++ Front Ends convert directly to GIMPLE (might change in the future)

Solves complexity problems for discovery of data and control flow

GENERIC – A **language-independent** representation that comprises the **superset of all ASTs** in GCC

- Still structurally rich/complex like AST

GIMPLE – lexically identical to GENERIC, but **imposes structural restrictions on statements**

- Expressions of 3 operands max
- Temporaries introduced to hold intermediate values
- Control structures lowered into conditional jumps
- Function calls accept only variables (no nested-function-calls)
- Lexical scopes removed
- `if` predicate contains only a single comparison
- Exception regions converted into an, "on-the-side" representation

- Derived from SIMPLE representation used by McGill University's McCAT compiler

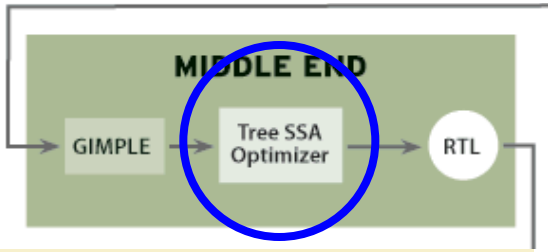
Simplified form to better generate optimized RTL

redhat.

https://www.redhat.com/magazine/002dec04/features/gcc/

Diego Novillo

Static Single Assignment (SSA)



- **SSA** is a **compiler formalism**: Representation used internally by compiler to (very explicitly!) track **how data flows** through the program
  - Is **essential to guide compiler** optimization decisions
  - GCC **implements on top of AST** (called ***Tree SSA***)
    - Uses Tree data structures (AST)
    - Uses SSA for data flow analysis

	GIMPLE program	SSA form
1	a = 3	a_1 = 3
2	b = 9	b_2 = 9
3	c = a + b	c_3 = a_1 + b_2
4	a = b + 1	a_4 = b_2 + 21
5	d = a + c	d_5 = a_4 + c_3
6	return d	return d_5

a_1 …first constant
b_2 ...second constant
c_3 ...third constant
a_4 ...fourth constant
d_5 ...fifth constant

Figure 5. Static Single Assignment Form of a Program

Tree SSA: *Based on versioning!*

1. Each time variable `X` is assigned a new value, compiler creates a new version of `X`.
2. The next time that variable `X` is used, compiler looks up latest version of `X` and uses it.

Representation is **internal to compiler** to reason about data flow!

- **Will not be seen** in generated code
- **Cannot be observed** by debugger

redhat.

https://www.redhat.com/magazine/002dec04/features/gcc/
Diego Novillo

Example Optimization: Constant Propagation

- *(very common),* Compiler **computes at compile-time** as many expressions as possible using constant values (C++11 extends to function calls with `constexpr`)

1. With program in SSA form, **all variables have version numbers**, so compiler **builds arrays indexed by version number**
2. All **expressions with only constant operands are deduced** at compile-time.

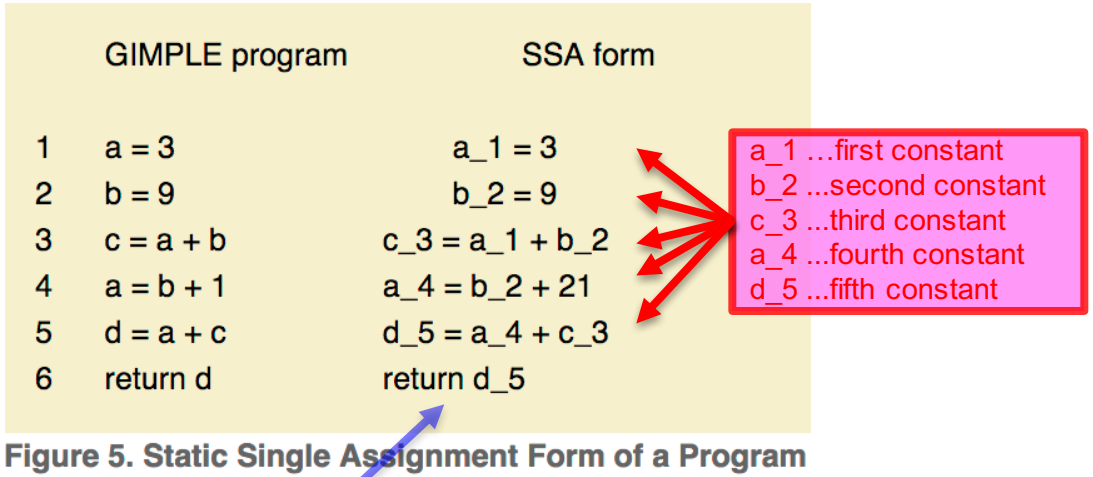


Figure 5. Static Single Assignment Form of a Program

Compiler creates array `CST[]` indexed by version

```
CST[1] = 3
CST[2] = 9
CST[3] = a_1 + b_2
CST[3] = 12
CST[4] = b_2 + 21
CST[4] = 30
CST[5] = a_4 + c_3
CST[5] = 42
```

```
1 a_1 = 3
2 b_2 = 9
3 c_3 = 12
4 a_4 = 30
5 d_5 = 42
6 return 42
```

After constant propagation, we discover return value is always **42**!
Compiler **removes all statements and returns 42**

WOW!

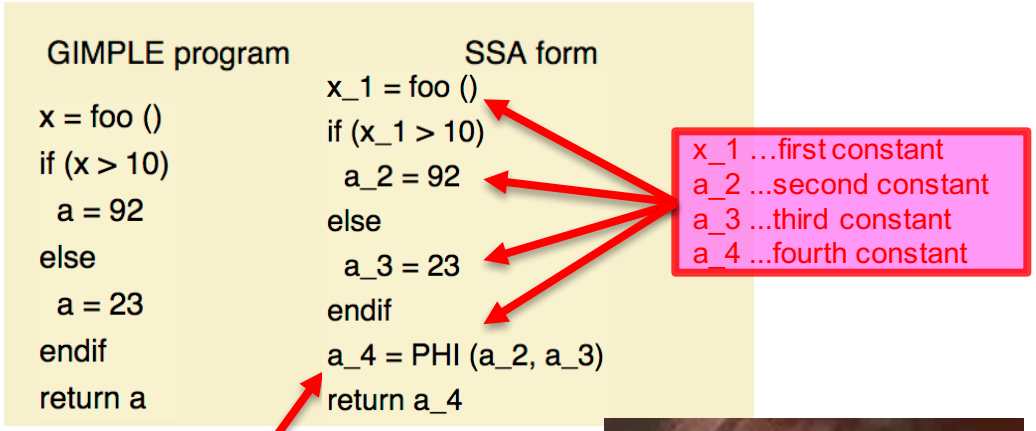
redhat.

https://www.redhat.com/magazine/002dec04/features/gcc/
Diego Novillo

Example Optimization: **Branch Logic**

- *(very common),* Compiler **does not know which version to use** for variable (because is result of runtime condition)

1. With program in SSA form, **create a (new) version that is the "merge" of conflicting versions** *(merge operation is called a PHI function)*
2. **Optimizers "see" the merge version**, which at runtime could be any of the conflict versions



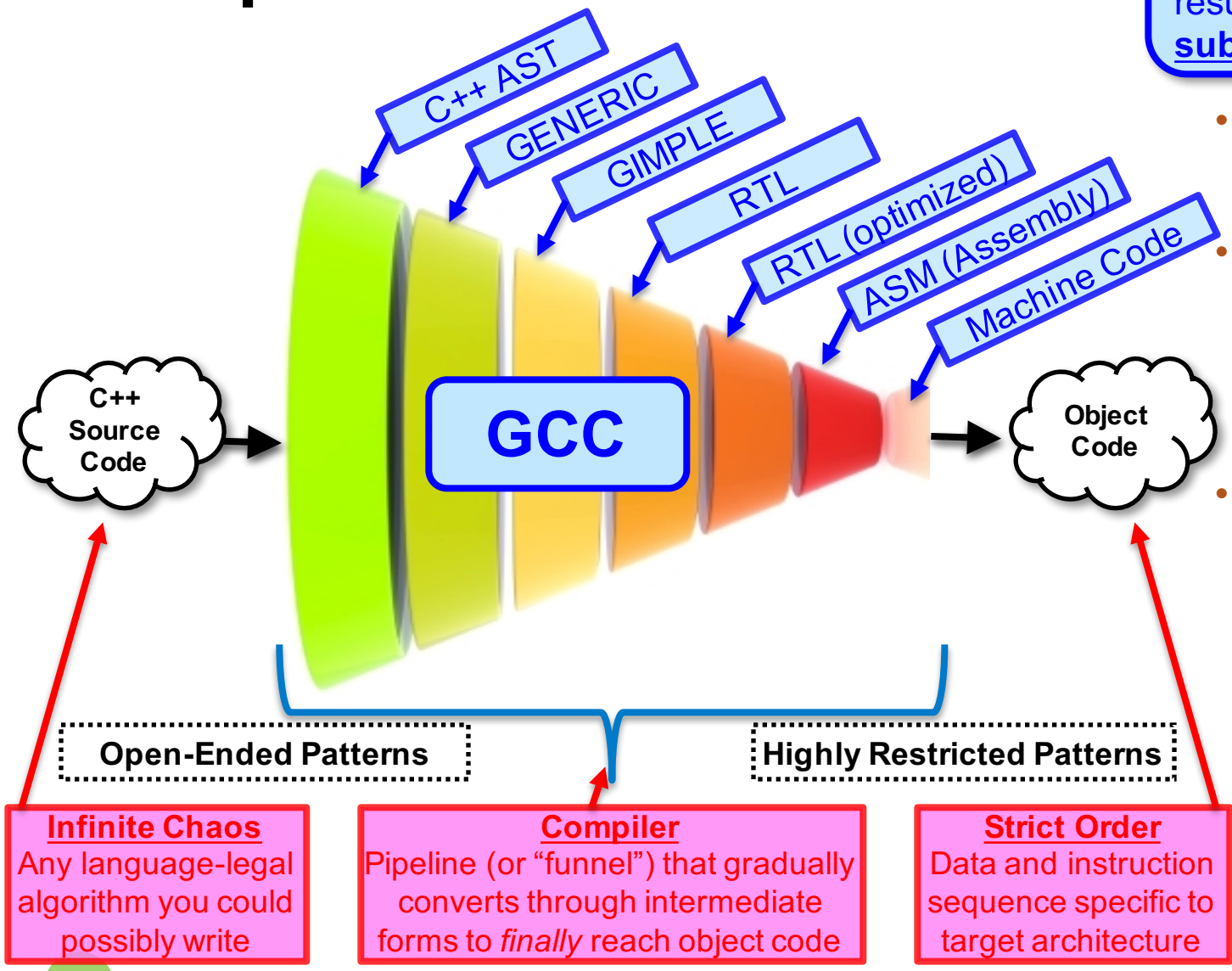
Compiler creates
a_4
that is the "merge" of conflict versions

Allows leveraging of the **CPU pipeline** for **parallel** and **speculative execution** of all conflict values!

https://www.redhat.com/magazine/002dec04/features/gcc/
Diego Novillo

The Lego Movie (2014)

Optimization Funnel



AST is unbounded (must accommodate any legal code construct), so discovering optimization patterns is hard. **Iterative passes replace patterns** with optimizations, resulting in increasingly more-bounded logic (where **subsequent levels can make greater assumptions**).

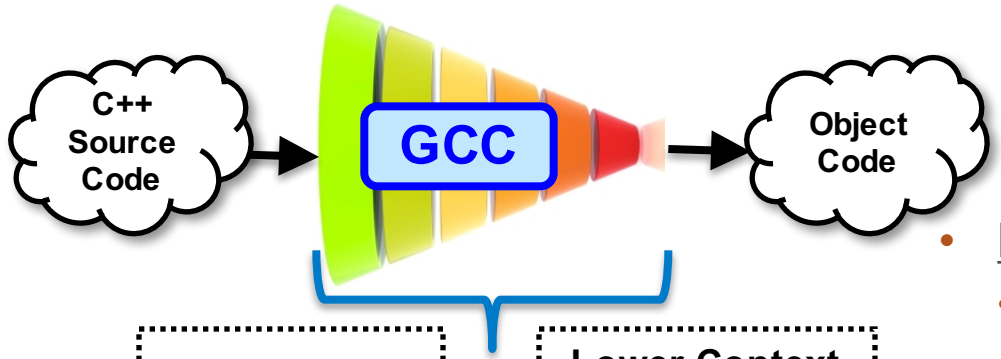
- **Optimization**: At each level…
  1. **Identify patterns and anti-patterns**
  2. **Substitute with "optimized"** patterns
- **Output** from each level:
  - **Is more "bounded"** with **increasingly "restricted" representation**
  - Allows **"next" level to make GREATER assumptions** for search/replace of patterns and anti-patterns
- **Next** Level:
  - **Increasingly loses source-code context** *(LOSSY!)*
  - **Gains architecture context** (physical limitations that exist in running system)
  - *How?*
    - **Dependencies are explicitly mapped**

LATER:
"Link-Time", "Whole-Program" optimization further manipulates algorithm (re-order instructions, pattern substitution)

Optimization At Different Levels

Good for optimization "close-to" source code

- **AST is "Higher Context"**
  - **More context** for search/replace patterns, anti-patterns
  - Search/replace **patterns are "bigger"** (like array references, data types, object references, control flow structures)
  - **Optimization is faster** (larger units are inspected/moved)
  - **Optimizations can provide significant (structural) benefits** across larger execution contexts



Higher-Context (closer to source code, more reasoning available)

Lower-Context (highly bounded input further restricted by specific target architecture)

Good for optimization "close-to" target architecture

- **RTL is "Lower Context"**
  - RTL optimization…
    - Is **informed by** very detailed **knowledge of target** architecture
    - **Leverages target** hardware
  - Optimization is "more specific"
    - Is **computationally expensive to identify "big" patterns** (slow compile)
    - **Does not have sufficient context to identify** (many types of) **"big" patterns**

Different types of patterns are possible for search/replace!

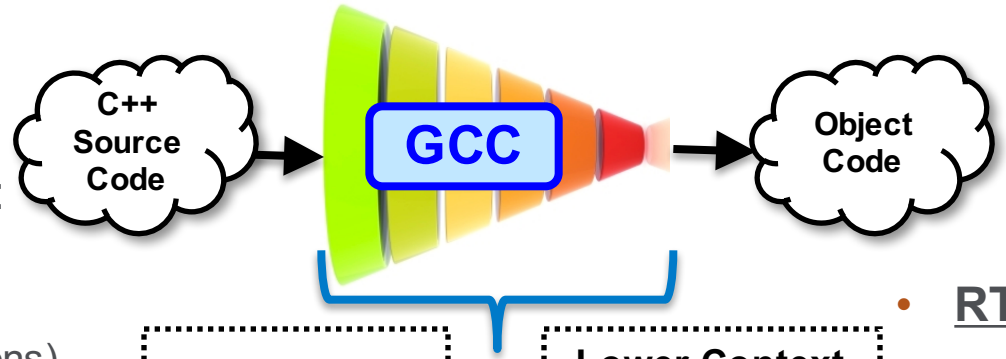
AST Optimization has:
- **NO knowledge of target** architecture
- **MUCH knowledge of source code** constructs

RTL Optimization has:
- **DETAILED knowledge of target** architecture
- **NO knowledge of source code** constructs

Optimization Opportunities At Different Levels

Maximize efficient patterns!
Remove inefficient patterns, anti-patterns!

- **AST optimization** opportunities:
  - **Algebraic simplifications**
  - **Constant folding** (execute literal expressions, `constexpr` expressions)
  - **Redundancy elimination**
    - **Loop-invariant code motion** (move code outside loop-body if computes same value each iteration)
    - **Common sub-expression elimination** (compute redundant sub-expressions once)
    - **Partial redundancy elimination** (remove computation from execution path to minimize computation redundancy)
  - "**Inline**" functions (whether `inline` was requested or not)
  - **Dead Code Elimination** (DCE)



Optimization is:
- **Iterative within** each level
- **Iterative across** levels

Maximize CPU resource utilization!
Minimize CPU resource contention!

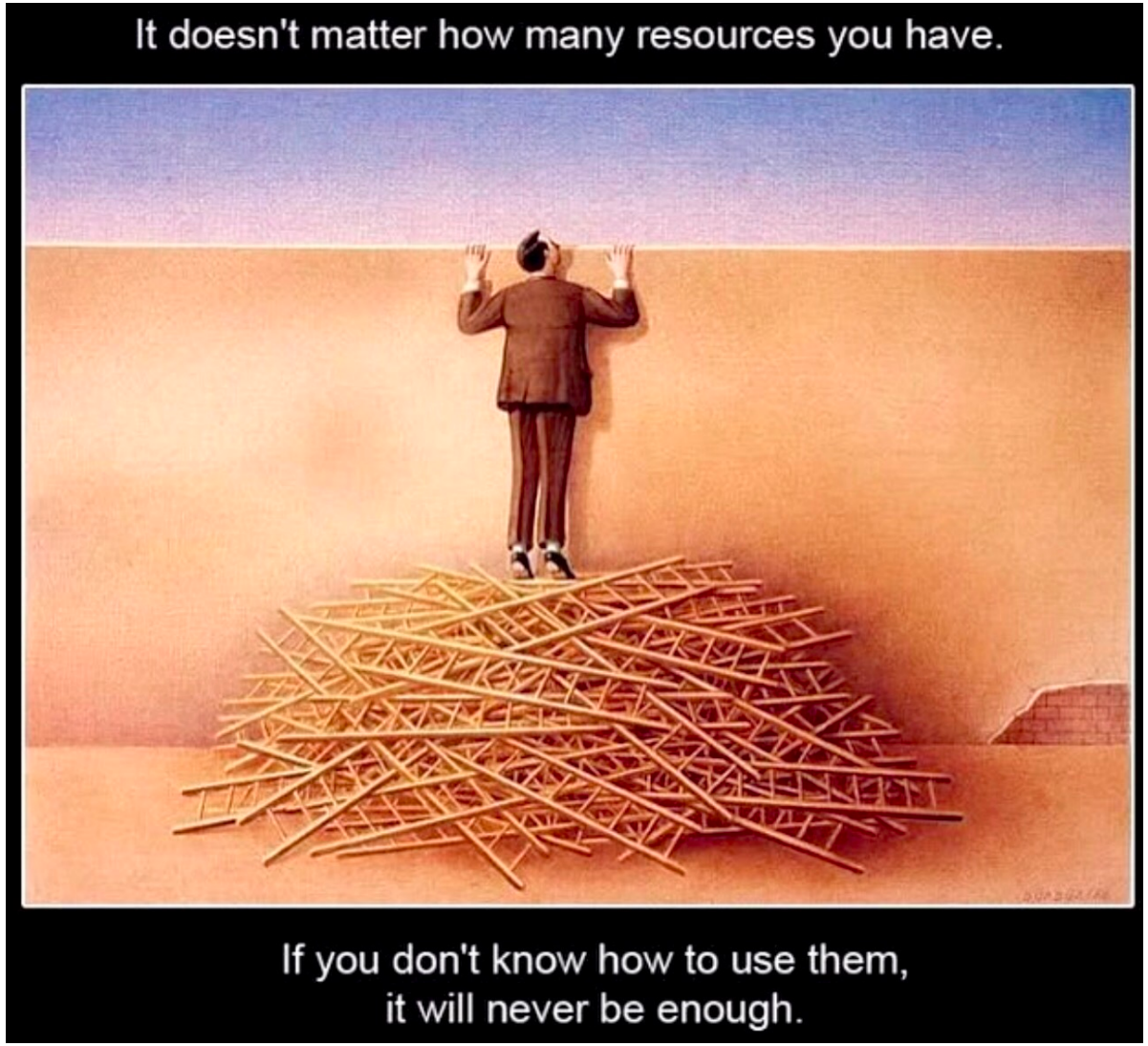
- **RTL optimization** opportunities:
  - **Register allocation** (maximize number of program variables assigned to hardware registers)
  - **Code scheduling** (leverage CPU super-scalar features to rearrange instructions so multiple instructions are simultaneously in different stages of execution)
  - **Instruction Substitution**
    - *(example),* replace multiple RTL instructions with single CPU-specific SIMD instruction
  - **Fine-grained tradeoffs** within target architecture (such as preference for speed, size, instruction set portability, etc.)

Program Execution

What's going on *inside* the CPU

Program Execution: Important Concepts

1. **Process Image**
   - **Data** and **Text** Segments
2. CPU **Registers**
   - **Architectural** Registers
   - **Dynamic** Registers ("Register Renaming")
3. Multi-Level **CPU Cache**
   - **Locality**: How The Cache Works
   - **Levels**: **L1/L2**, **L3** (L4…?)
4. CPU **Instruction Pipeline**

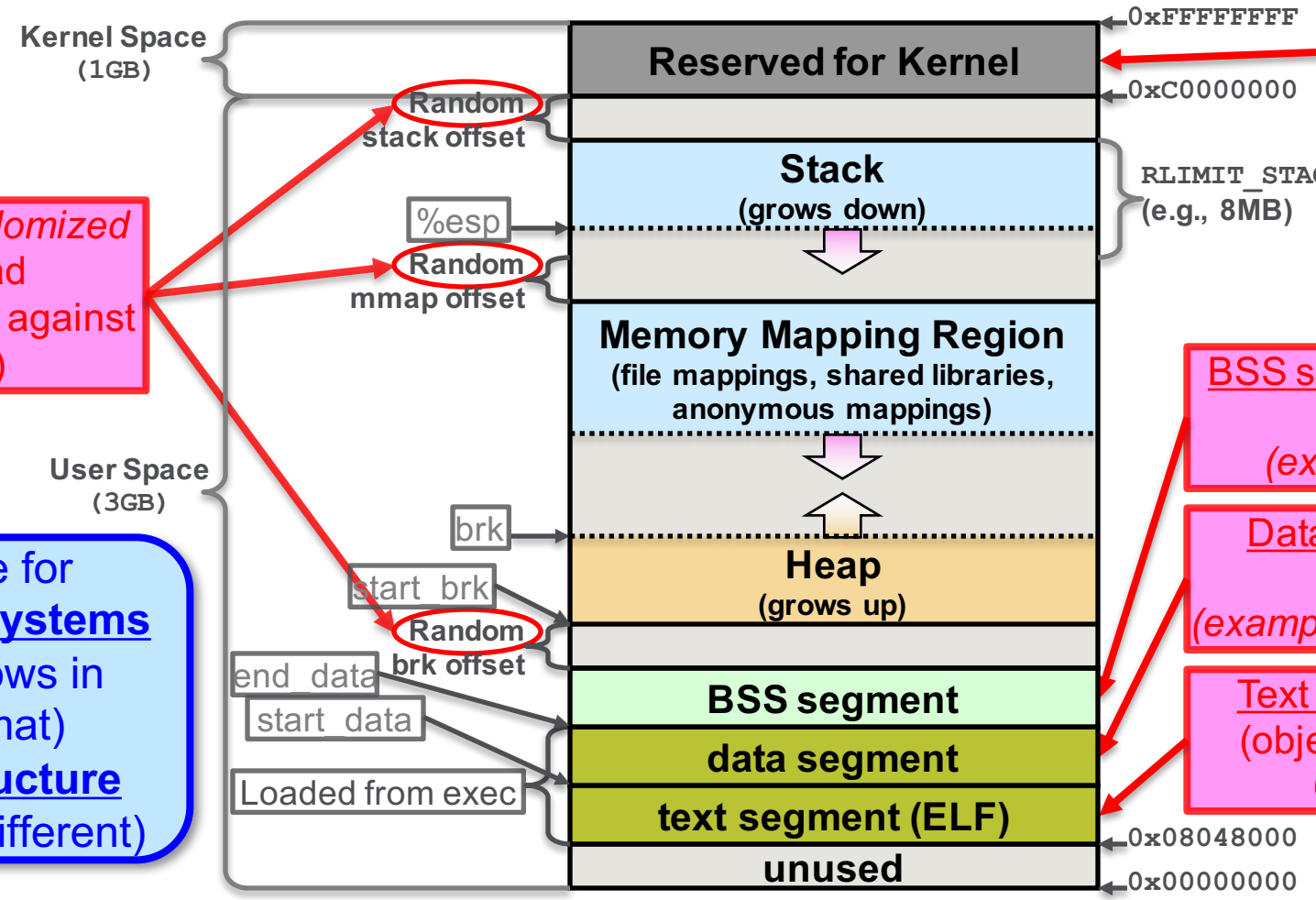


Review: The Process Image

Is a **Sequence** of **text** (instructions) and **data**!

Linux (32-bit) Process Memory Layout

Executable and Linkable Format (ELF)



User code CANNOT read/write to these addresses (causes segmentation-fault)

Addresses are *randomized* on system load (to improve security against viruses, etc.)

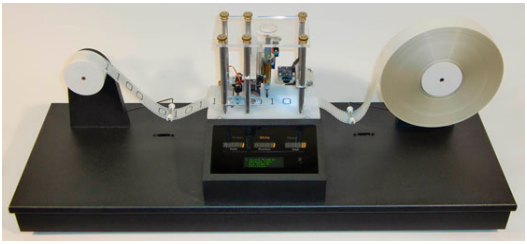
BSS segment: Uninitialized static data objects, filled with zeros
(example): `static char* p;`

Data segment: Static data objects initialized by programmer
(example): `static char* p="foo";`

Text segment: Process instructions (object code) for target architecture (in this case in ELF format)

Process Image for **other operating systems** (e.g., MS-Windows in PE/COFF format) have **similar structure** (placements are different)

Executing Instructions (in the CPU) (1 of 5)



```
forever:
```

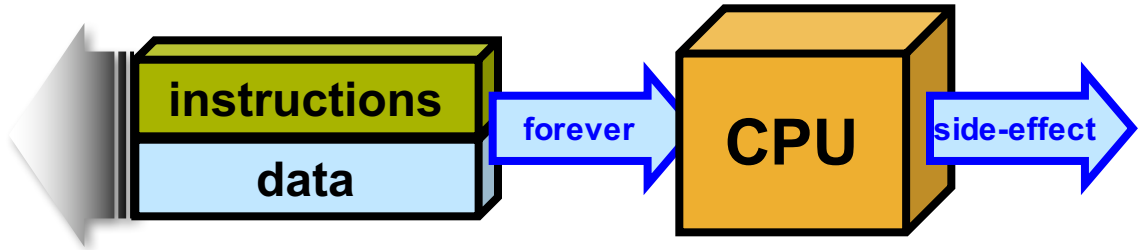
① Load next instruction

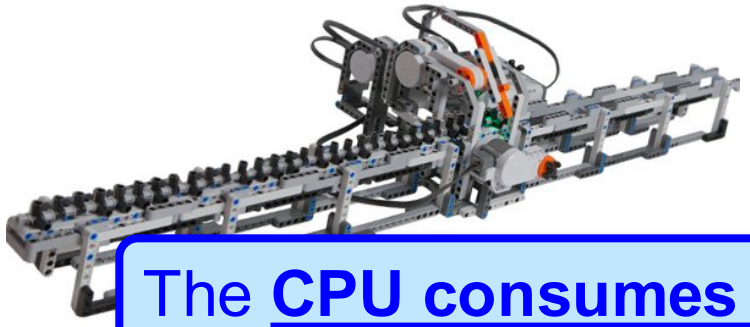
② Load operands

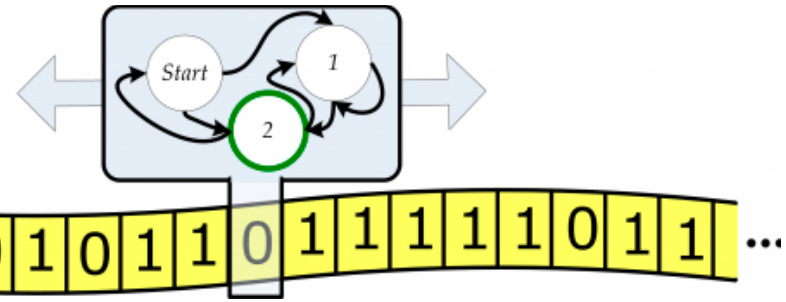
③ Compute!

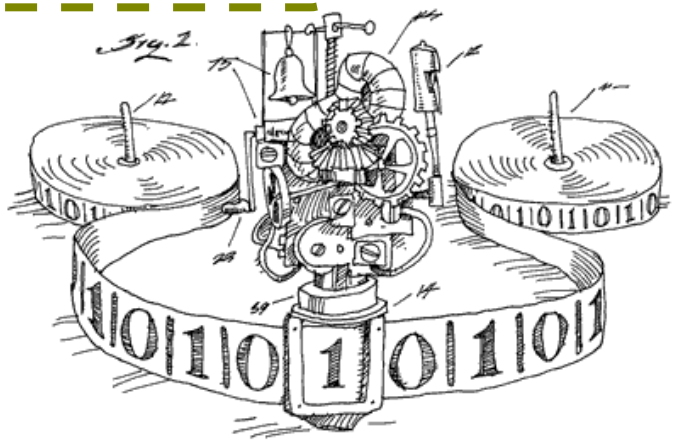
④ goto ①

The **Running Program**: A **stream of instructions and data** *into* the CPU processor core









The **CPU consumes instructions-and-data** (producing results and side-effects)

Executing One Instruction

- **Instruction is "loaded"**: `opcode operands`

All instructions look like this!

"Increase register `ebx+0310` *by* `42`*"*

```
add [ebx+00000310],42
```

opcode *(operation-code)*

operand 0

operand 1

Each instruction follows a "life-cycle" *(example):*
Fetch, Decode, Execute, Memory-access, Writeback

The "classic RISC pipeline"

Opcodes are defined by the **CPU microarchitecture** *(example):*

- `dec` ...decrease value by 1
- `inc` ...increase value by 1
- `sub` ...subtract value from value
- `add` ...add value to value
- `mov` ...copy second onto first
- `cmp` ...compare two values
- `jmp` ...jump to address
- `je` ...jump if compare was "equal"
- `jne` ...jump if compare was "not-equal"
- `jg` ...jump if compare was "greater-than"
- `jl` ...jump if compare was "less-than"
- ...etc.

Executing Instructions (in the CPU) (2 of 5)

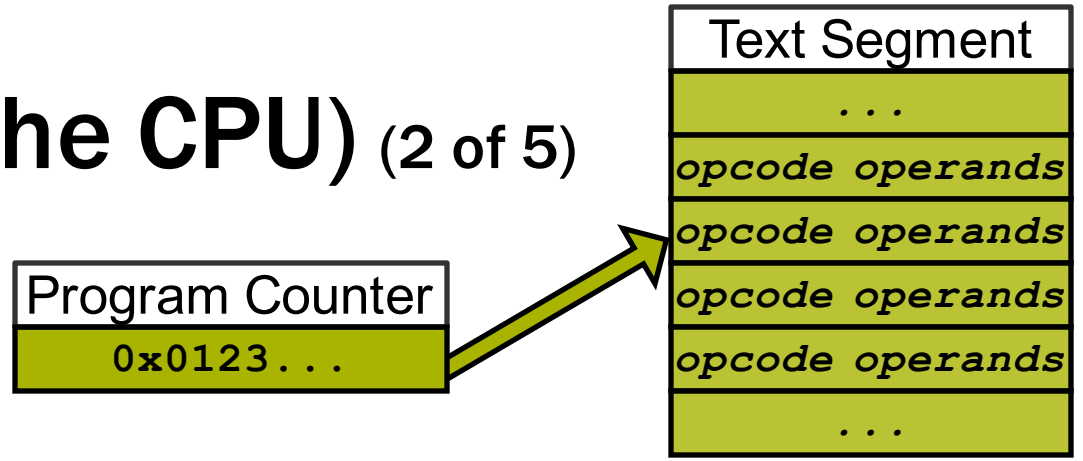
```
forever:
```

① **Load next instruction**

② **Load operands**

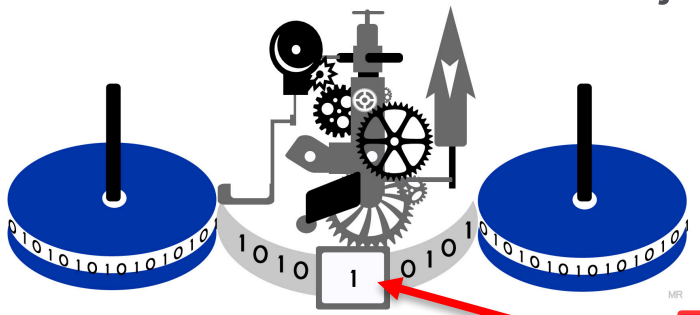
③ **Compute!**

④ goto ①



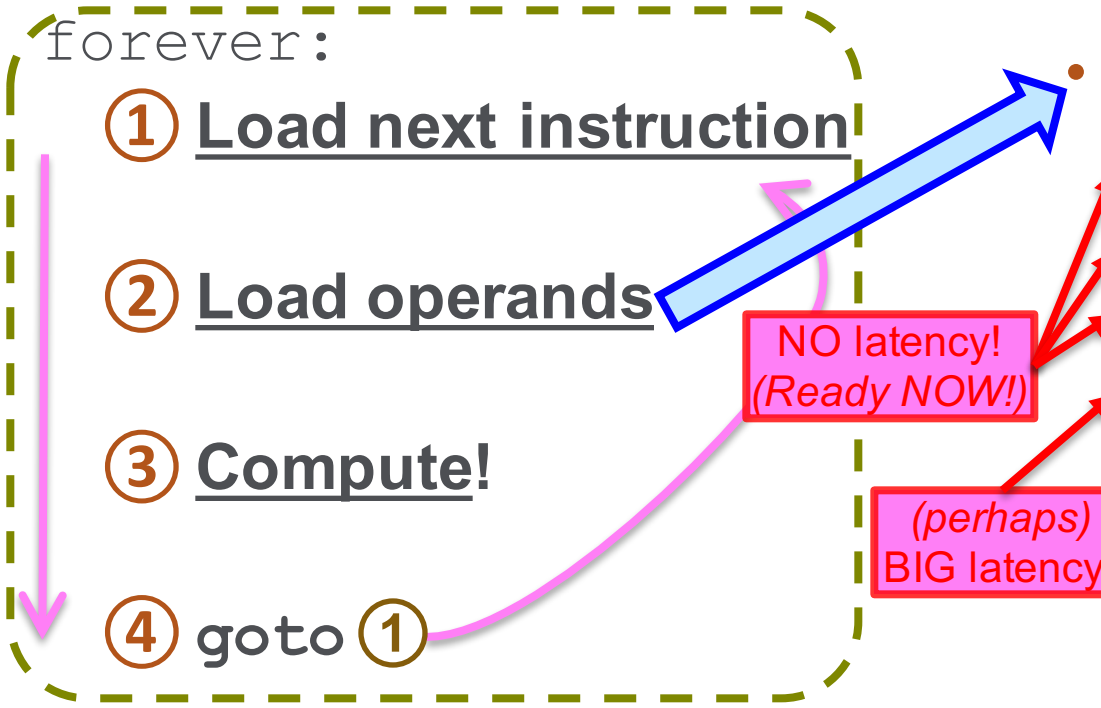
- The **Program Counter (PC)** is a special register containing the "current-instruction" being executed
  - Is "**incremented**" for **each instruction "loaded"**
  - **Program may** explicitly "**increment" (overwrite) PC**
    - *(example):* "**branch-logic**" runtime-condition establishes the "next-instruction" to "jump-to" (execute)

Runtime determination for the "next-instruction" to execute!



Program Counter (PC) contains the "current-instruction" being executed.

Executing Instructions (in the CPU) (3 of 5)



- Operand may be:
  1. **Implicit** (may operate on "well-known" register)
  2. **Embedded** in the instruction (e.g., "42")
  3. Identified by **"register-name"**
  4. Identified by **memory-address**

NO latency! *(Ready NOW!)*

(perhaps) BIG latency!

(perhaps) small latency to load into register

Instruction **stalls** while waiting for the operand

OHZ NOZ!

- *What if* conflict in cache?
- *What if* must wait on data bus?
- *What if* paged to disk?
- *What if* must wait for a device controller (locked by another process) to become available?

Loading operand from memory address is a highly **variable** *(unpredictable!)* **runtime**-expense

Desired: Low Latency

- Operand may be:
  - **Implicit** (may operate on "well-known" register)
  - **Embedded** in the instruction (e.g., "42")
  - Identified by "**register-name**"

LOW latency

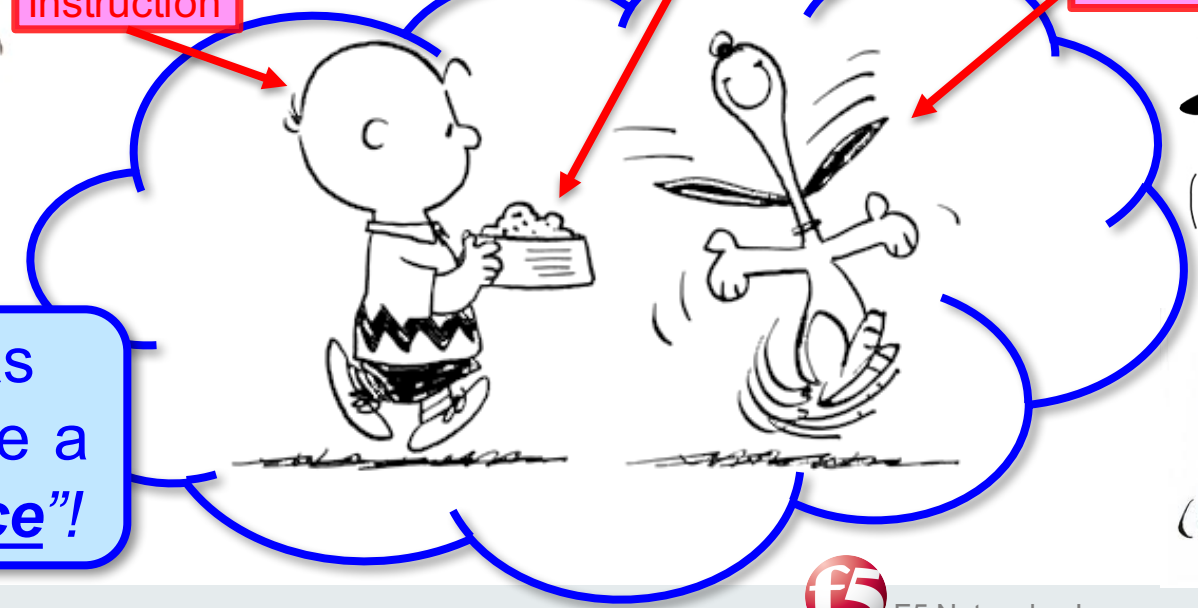
  - Identified by **memory-address**

HIGH latency

Instruction operands

Machine Instruction

CPU



Every time a CPU instruction has **low latency** operands, somewhere a cute puppy does its *"Happy Dance"!*

Lowering Latency: The CPU Cache

- **Function of any cache**: To **buffer the transfer of data** between a faster and slower device.
- CPU Cache: For **each memory-access** operand, CPU will FIRST inspect the **CPU (L1) cache**
- **Multi-Level**:
  1. Look to **L1** *("hit" or "miss"?)*
  2. Look to **L2** *("hit" or "miss"?)*
  3. Look to **L3** *("hit" or "miss"?)*
  4. (newer CPUs) Look to **L4** *("hit" or "miss"?)*
  5. …etc.

- *"hit" == DONE*
- *"miss" == KEEP_LOOKING*

If the cache "works", it MUST have a copy of the **data object** that we want **to access *next***

How The Cache Works: Locality

1 **Temporal Locality:**
The "next" data object accessed is *likely* **one that we recently accessed**

(example) loop-counter!

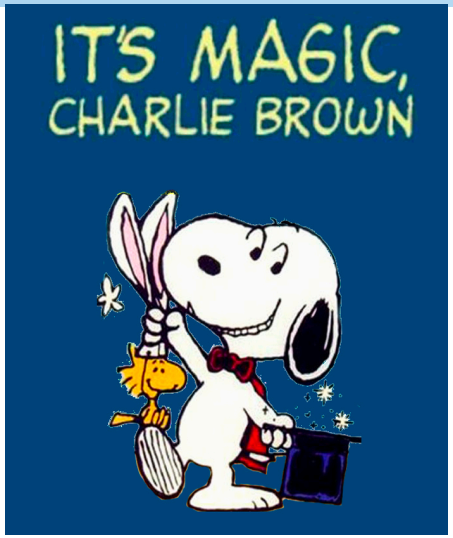
2 **Spatial Locality:**
The "next" data object accessed is *likely* **at a memory address near** one that was recently accessed

(example) iterating contiguous array!

- We gain "**efficiencies**":
  - By "fast-access" to **local-cached copies** of far-away state
  - By **delaying** (time-shifting) updates (for when we are not busy)
  - By **eliding** (collapsing) updates (to lower total activity)
  - By **pre-fetching** yet-to-be-requested state (the *cache line*, *CPU heuristics*)

Temporal and spatial **locality is how all caches work**, *everywhere!*
(not just CPU caches)

Critical Cache Issue: Dependencies!



- A "cache" implies **multiple copies** of a data object **exist**!
  - **Where** are the copied data objects?
  - **Can we trust** a given data object?
  - **How do we change** the "authoritative" data object?
  - **When** do we update the (stale) data object copies with the change?

The CPU **auto-MAGIC-ally** does all of this for you!

The cache ONLY WORKS if we respect **DEPENDENCIES**!

Coherency **TAKES TIME**! Your algorithm and cross-thread contention can establish EXPENSIVE dependencies (which THRASH the CPU)!

CPU Cache Coherency: A consistent (universal) view of state, no matter how many copies exist in what locations

CPU Cache **MAGICALLY WORKS**. If you write your own software-system cache (for any purpose), don't forget to take care of all these details.

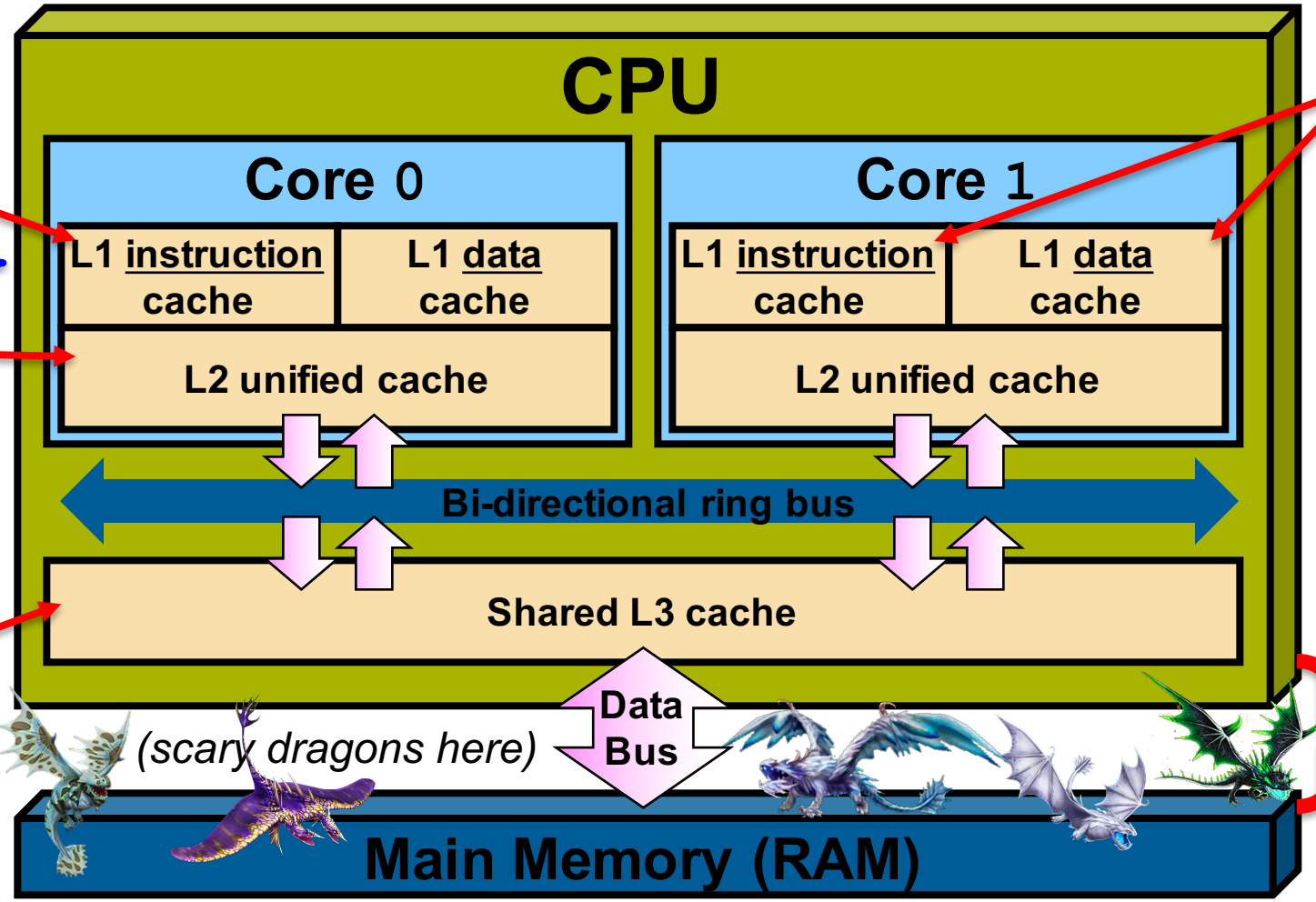
The CPU Cache L1/L2 Partnership

(example) "i-Series" CPU Architecture

L1/L2 Partnership

Primary Cache (L1) maximizes "cache-hit" rate

Secondary Cache (L2) minimizes "miss-penalty"



Separate i-cache, d-cache speeds up instruction fetch/decode, and data fetch/store

Also, efficiency: different access patterns for instructions and data *(instruction writes are rare)*

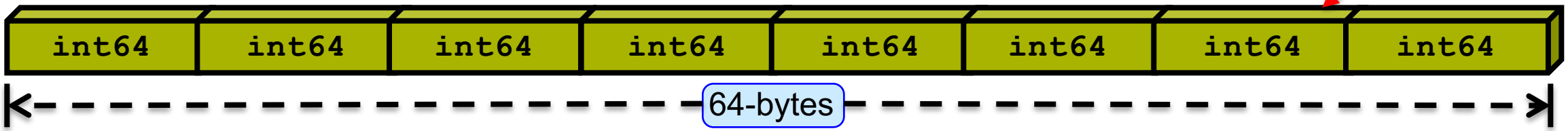
L3 is CPU-specific (usually a pool shared across cores)

Going Off-Chip: The CPU Cache Line

- The CPU only **transfers "fixed-size-chunks"** to/from main memory (the "**cache line**", or "**cache block**")
  - **Architectures specify** between 16 to 256 bytes, but most typical is 32, **64**, or 128 bytes.
  - Many **(performance, efficiency) reasons** for using cache line
  - Device and memory controllers are **optimized for the cache-line size**
  - CPU multi-level cache internally manages **whole cache lines**

Most desktop and mobile CPUs (x86, AMD64, ARM) use a **64-byte cache line**

(example 64-byte CPU cache line)



The CPU **only transfers** to/from main memory in "**chunks**" called the CPU **cache line**

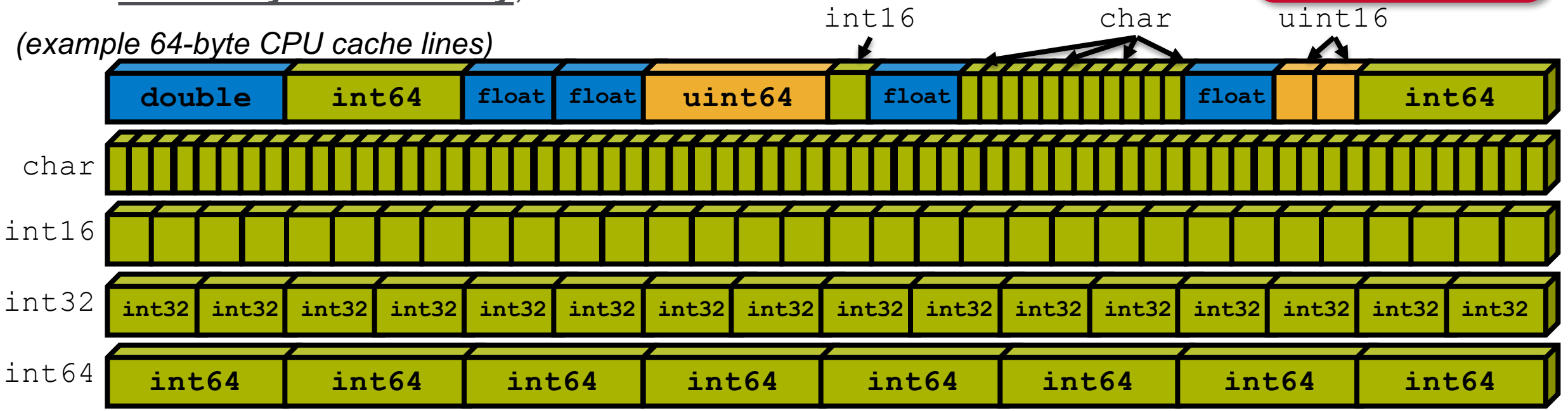
Going Off-Chip: The CPU Cache Line

- Ask for whatever you want, you get a **whole cache line**
  - ...**ask for a byte**, *get* a whole cache line
  - ...**overwrite a byte**, *dirty* a whole cache line
  - ...**store a byte to memory**, *transfers-and-overwrites* a whole cache line

CPU Cache will magically maintain **cache coherency** among all cache levels, across all cores, and into main memory!

Cache coherency (propagation) **TAKES TIME**!

(example 64-byte CPU cache lines)



Concern: **Two threads "fighting"** over *writes* to the **same cache line** (CPU cache will *correctly resolve*, but may be **TIME-EXPENSIVE**)

Runtime variable!

Leveraging The Cache Line

It exists!

- **Know** the **cache line exists**
  - **You're using it** "for free"
  - Be able to follow hallway conversation referencing **cache line terms**

Don't be abusive!

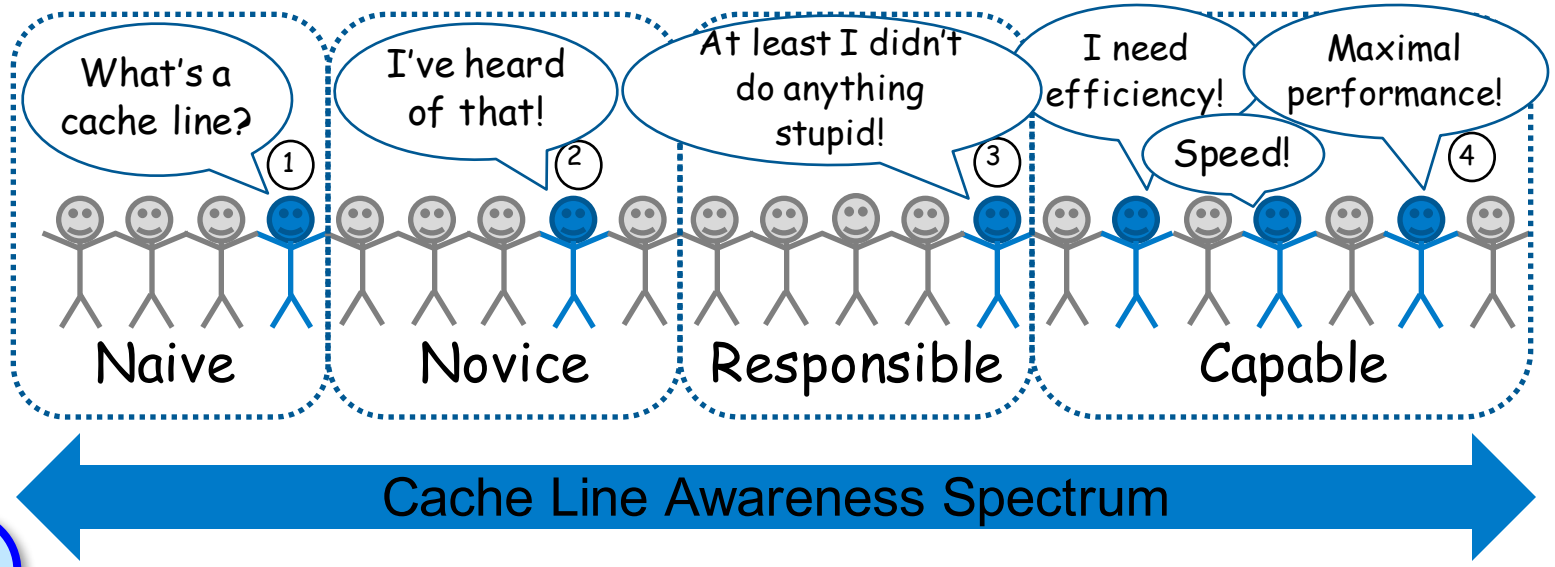
- **Avoid problems fundamentally in conflict** with how the cache line works
  - Consider **Data Alignment**!
    - avoid breaking data objects across cache line boundaries
    - avoid co-locating into same cache line data objects concurrently updated from separate threads

Leverage when you can!

- **Design** for **cache line efficiency**
  - **Consider cache line boundaries** for
    - **data structure layout**
    - **algorithmic access** of data objects

The "Cache Line Awareness Spectrum"

We are C++!
This is our duty



Naïve: Usually works

- Amazing advances by compiler and CPU to mask our transgressions

Novice: A "Good Place To Start"

- With small effort, can understand CPU cache, role of cache line

Probably, Most Programmers Should Be Here

Responsible: The Goal

- Know enough to identify fundamentally flawed approaches
- Reason about cache line in critical scenarios

Capable: Cache Line is a Design Element

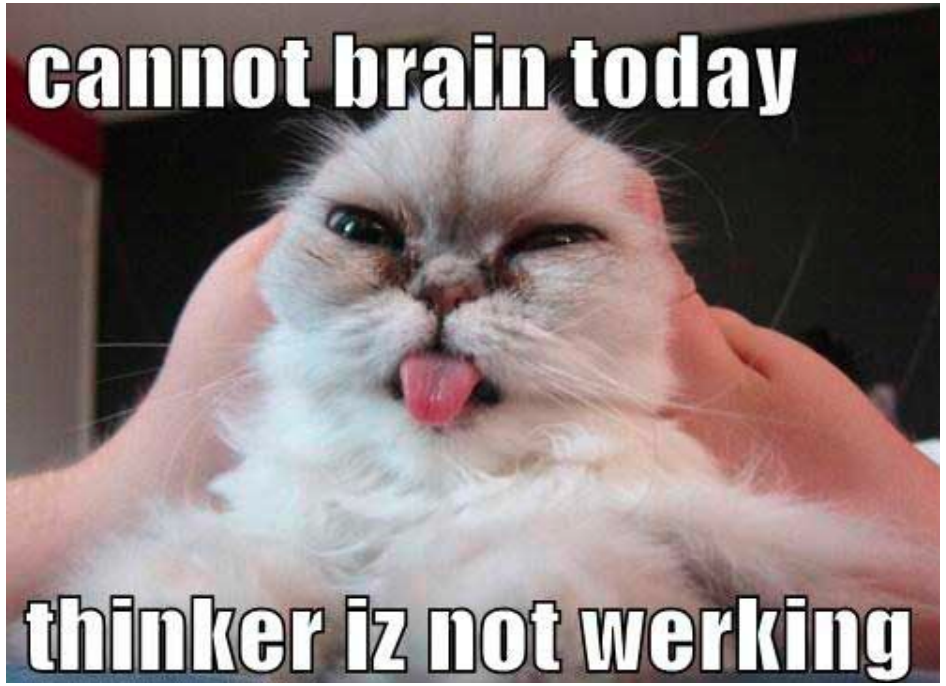
- Greater success, freedom implementing performance and efficiency-sensitive systems
- Easier "scaling" and system evolution (some problems never present)

Certificate Of Achievement

Print, sign, and place on your, "I Love Me!" wall

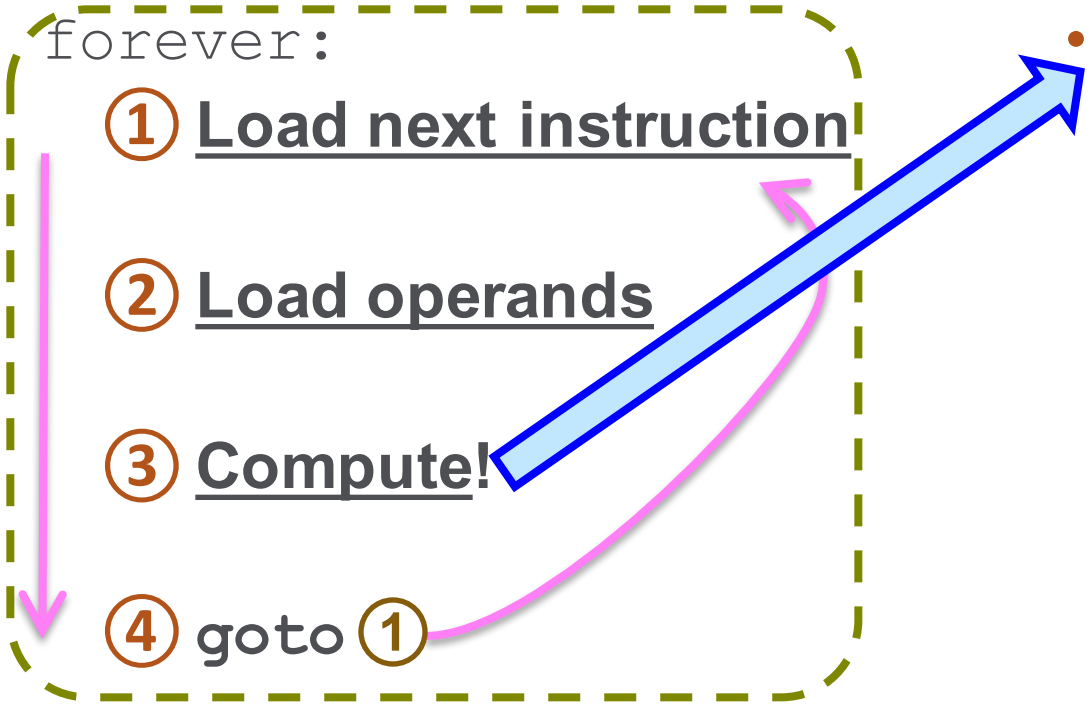
Review: Operand Fetch Latencies

- **Latencies exist**
- The **CPU cache** helps **lower latencies**
- Latencies are **runtime-unpredictable**



Operand fetch latencies are ***unpredictable*** due to the chaotic runtime nature of the CPU cache, its current state, other threads, other processes, memory alignments, device controllers, and PAST data-specific processing patterns and activities

Executing Instructions (in the CPU) (4 of 5)

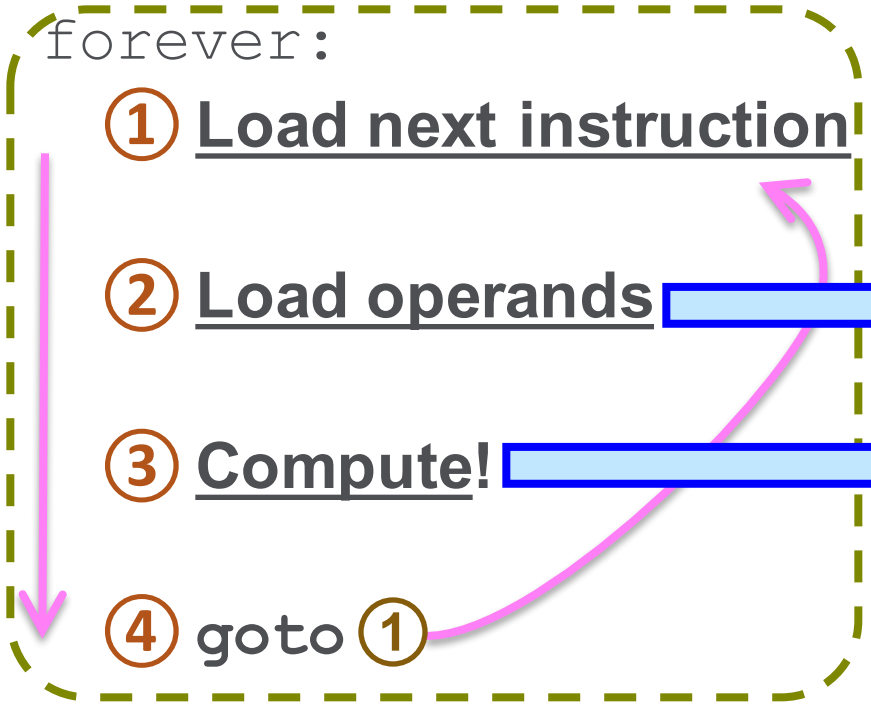


- **Opcodes** require **variable computation time** (number of CPU cycles) depending on:
  - How the CPU is optimized
  - Operation complexity
  - Contention with other resources within the CPU
  - CPU cache status, conflicts, invalidations
  - Distance among dependent registers/resources
  - Seemingly unrelated things, like:
    - specific data values being processed
    - other active cores/threads/processes/chips
    - *previous*, or current system device activity
    - type and level of bus activity
    - room temperature *(yes, really!)*
    - Phase-Of-The-Moon

Time for opcode completion is **unpredictable**: based on "guidelines" and "probabilities", varies with many *(uncontrolled)* system characteristics

With vendor support and extensions, C/C++ *can* control for well-defined timing (such as for real-time operating system [RTOS] applications)

Executing Instructions (in the CPU) (5 of 5)



Takes TIME!

(Runtime unpredictable!)

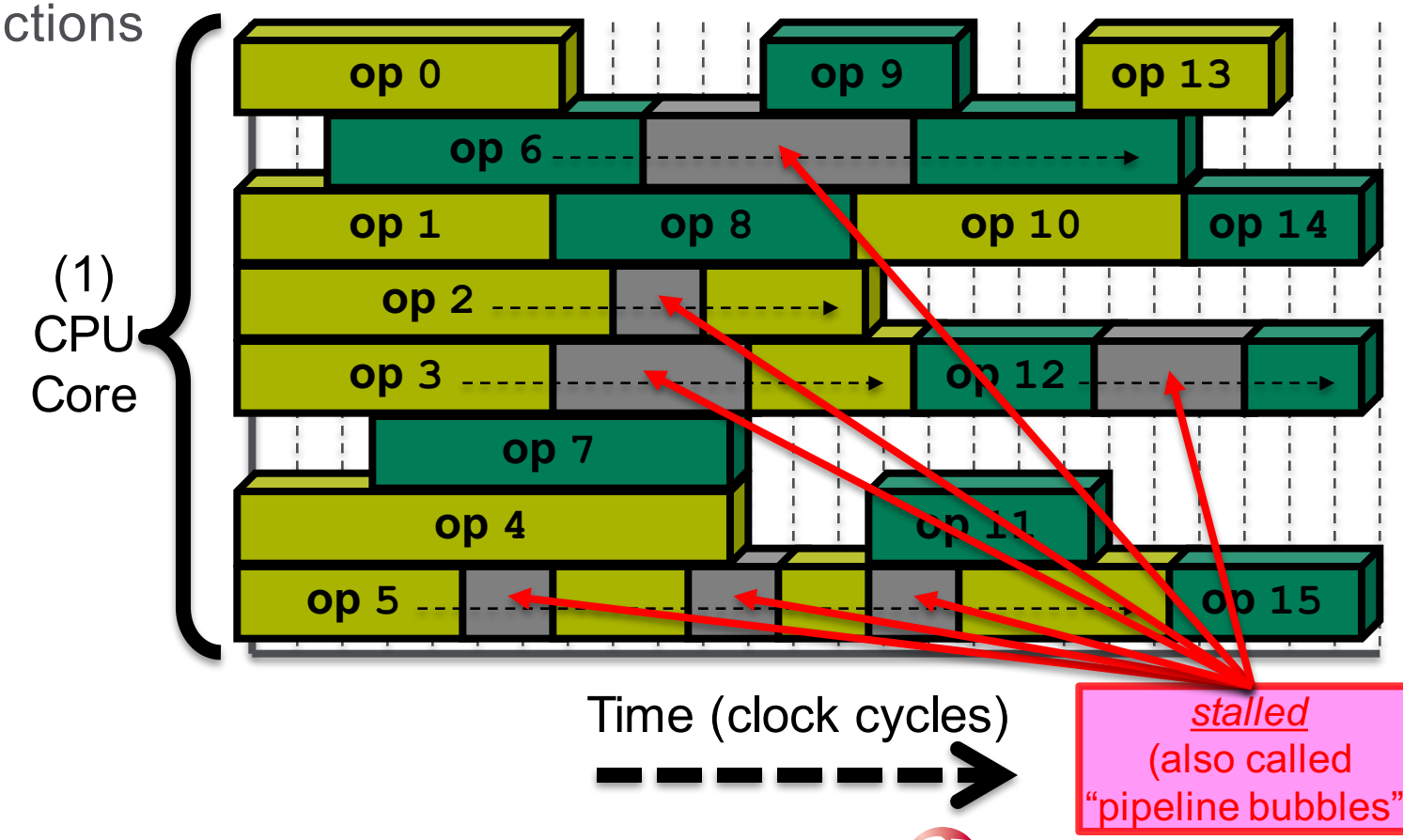
CPU employs the "**Instruction Pipeline**"
to **be productive while WAITING** on operations
that take *(unpredictable amounts of)* time!

CPU Instruction Pipeline

Today's CPUs, **each core**:
- several dozen simultaneously executing instructions
- can start a dozen each cycle
- can retire more-than-a-dozen each cycle

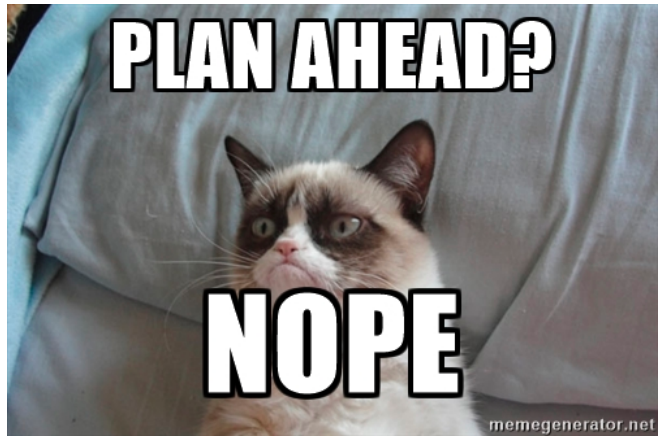
- **EACH clock** cycle (each core):
  - **Start MANY** instructions
  - **Make progress on MANY** instructions
  - **Retire/Abort MANY** instructions

"Superscalar Pipelining"
(multiple instruction pipelines in parallel)



With the CPU instruction pipeline, **MANY instructions** can "**make-progress**" during **EACH clock cycle** (while possibly MANY instructions are *STALLED* waiting for operands)

CPU "Code Re-Ordering"



- CPU will re-order instructions (at runtime)
  1. CPU will **fetch many** instructions
  2. CPU will **analyze dependencies** among instructions, and contention for CPU internal resources (registers, CPU "execution units")
  3. CPU will **re-order instructions** to minimize *"stalls"*
  4. CPU will **execute the reordered-instructions**
- CPU re-ordering algorithms:
  - Are **specific to CPU family/model**
  - May **change with** CPU microcode (firmware) **updates**
  - May **change at runtime** based on runtime CPU heuristics

Different processors make different tradeoffs (for energy consumption, throughput, response latency, etc.)

CPU code-reordering goals are for:
Execution Efficiency vs. **Performance** *(throughput)!*

Eager and Speculative Execution

ENTIRELY within the CPU processor **core**!

- **Will "look-ahead"** to FUTURE instructions
  - **Eager:** Begins execution of "later" instructions (for which no prerequisite dependencies exist), these are completed and committed
  - **Speculative:** "Guesses" likely branch, and begins execution
    - may simultaneously execute many/all branches!
    - aborts branches later determined as "not-taken"
- **Will "re-start/repeat"** instructions
  - If "**guessed" input**-operands were **wrong**
  - If **CPU cache** operand values are later **invalidated** during instruction execution (perhaps by other processor threads/cores)

A "future" instruction may complete first!

(proprietary) "**Prediction Logic**" is advanced *(intensive!)* processing **entirely within the CPU** to identify instruction and data access patterns **to optimize *(change!)*** cache and instruction execution based on what is "seen" during program execution

Delivers enormous **Real-World Benefits** in **Real-World Scenarios**!

CPU Transactional Memory will do LOTS of this! (expect more in the future!)

From where do the instructions come?
- The **program counter (PC)**
- (proprietary) **CPU "heuristics"**

What Goes On Inside Intel x86 Processors

x86 Internals for Fun & Profit • Matt Godbolt

GOTO Conferences

Subscribe 34,602

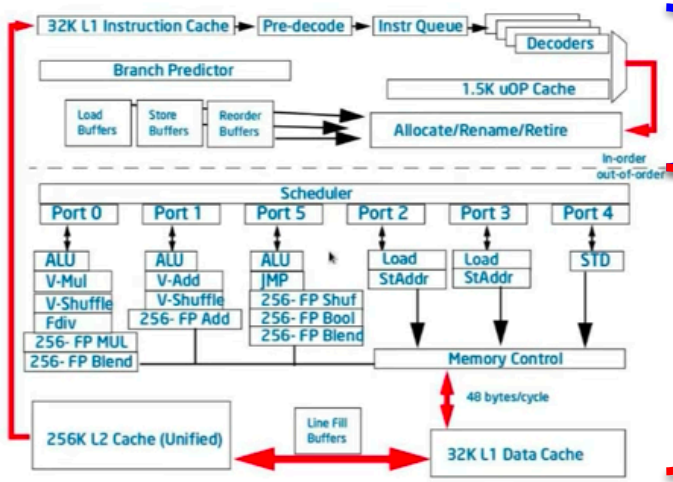
Add to Share More

Published on Aug 11, 2014
This presentation was recorded at GOTO Chicago 2014
http://gotochgo.com

https://www.youtube.com/watch?v=hgcNM-6wr34

What's all this then?

- Pipelining
- Branch prediction
- Register renaming
- Out of order execution
- Caching



In-order Execution

Out-of-order Execution

Interesting discussion:

- Compiler trickery with registers
- CPU branch prediction heuristics
- Example reordering of loop-body for BIG performance changes
- CPU micro-operation scheduling (within single instruction)
- CPU "renamer" for **on-the-fly dependency analysis** on registers

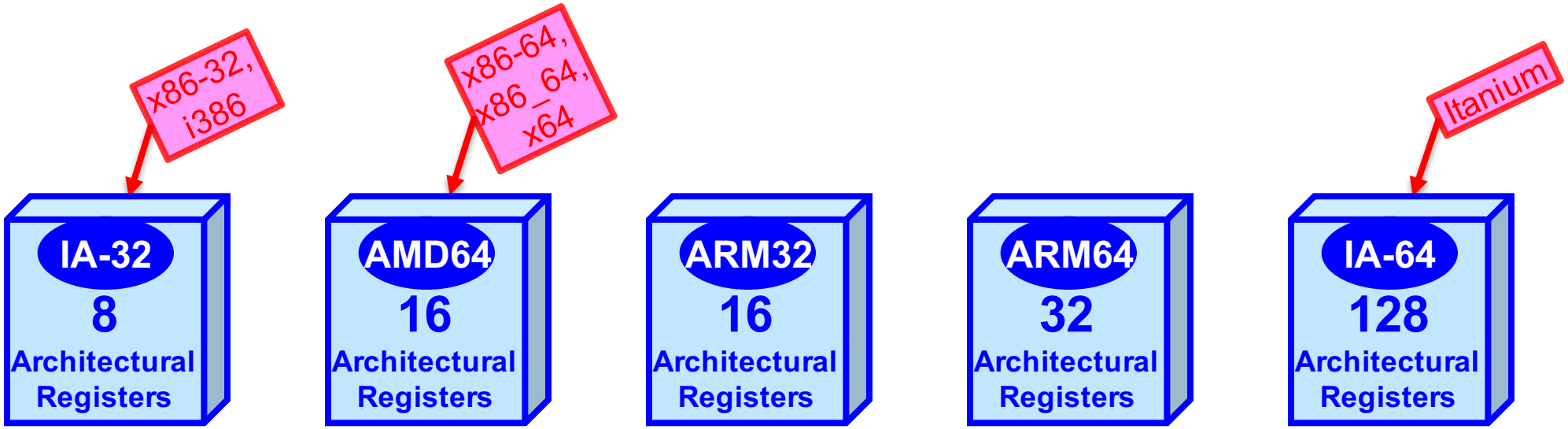
Competition For Resources Within The CPU

- **Execution units** – computational elements (e.g., for integral math, for floating-point math, pointer arithmetic, etc.)
- **Registers** – Accessible data locations used by execution units
- **CPU will "associate" and "re-order"** instructions to make best use of available execution-units, registers
- Simultaneously:
  - **Other processes** compete for **CPU cache resources**, **devices**
  - **Other threads/processes** compete for **execution units**, **registers** within one core (in Hyperthreading/SMT scenarios)
  - **Other instructions** compete for **execution units**, **resources** within one core

A whole lot of incestuous "stealing" and "sharing" goes on inside the CPU, **negotiated ENTIRELY at runtime**, influencing LATENCY and TIMING (and INSTRUCTION ORDER!)

CPU Architectural Registers

- Are **Well-known** within the architecture (e.g., "named" for "specific purpose")
- **Not many** (many reasons, "small-is-fast", "distance-is-time")

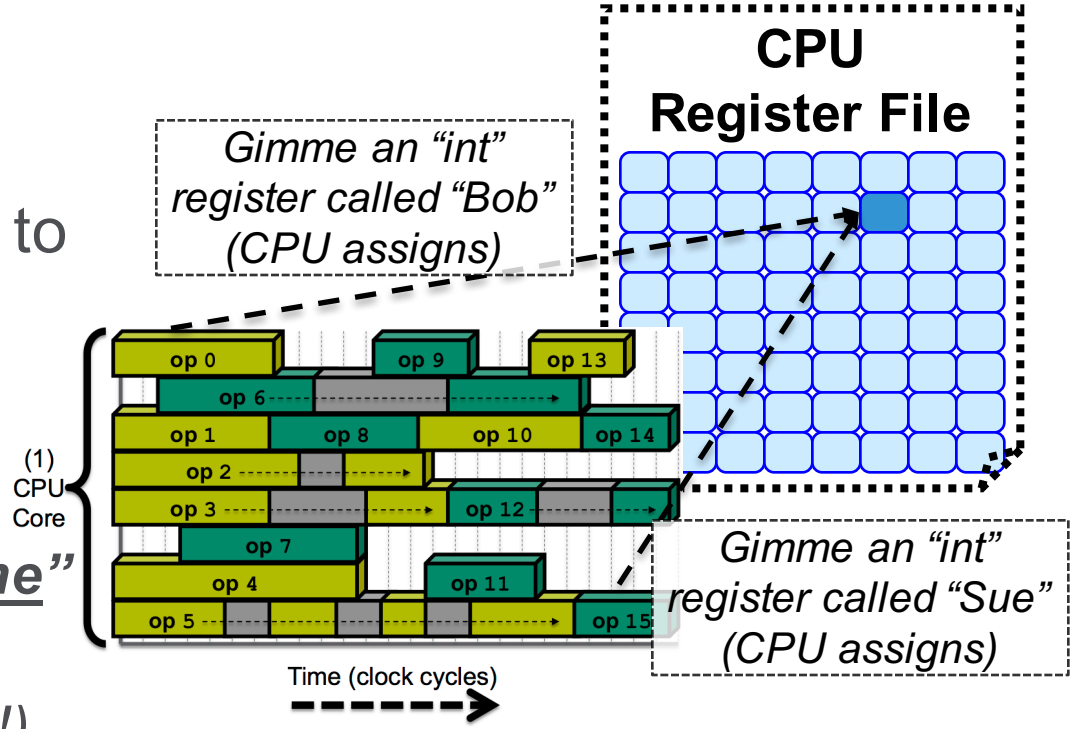


Compiler can specify architectural register, but **prefers to NOT** (because CPU is better at allocating scarce resources)

Similar to **(now deprecated)** C++ keyword `register` *(don't use it, you'll be wrong)*

CPU Register Renaming

- Many **MORE registers "exist"** within the CPU to support the instruction pipeline
  (2+ orders more, e.g., 100+ or more)
- **Register Renaming**: A technique
  ① **Compiler "asks"** for a register, assigning a ***"name"***
  (Register name has a scoped lifecycle!)
  ② CPU analyzes ***named-dependencies*** *(at runtime!)*
  ③ CPU **re-orders instructions** to **assign** architectural and other (available) "pipeline" **registers**
  (maximizing utilization, minimizing latencies)



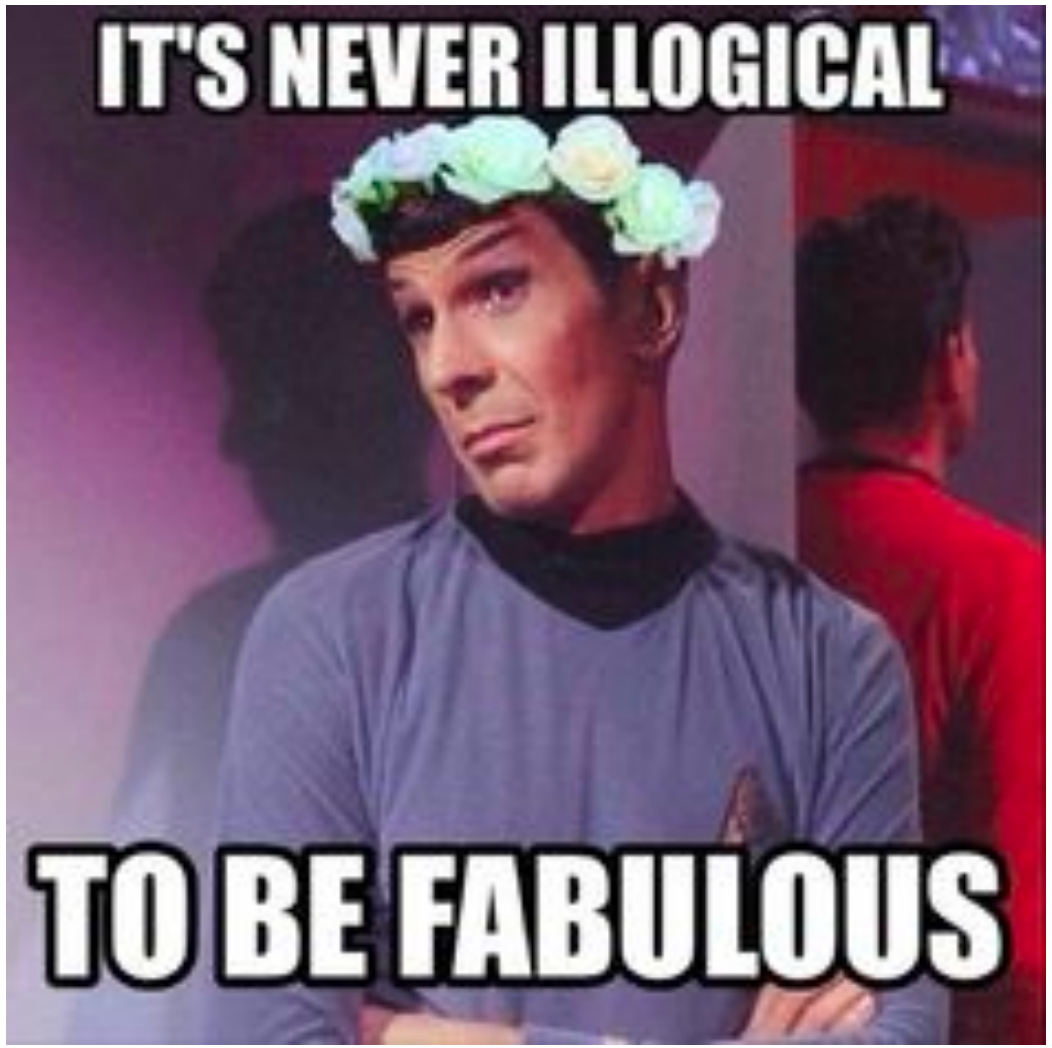
Register Renaming has been called a, "devious strategy" to allow the CPU to "reuse" registers by tracking the lifecycle of a "named-register" operand *(defining a **logical instruction sequence**!)*

Result: **Instructions "steal" registers from each other** *(based on register scoped lifecycle, and runtime competition for available resources)*

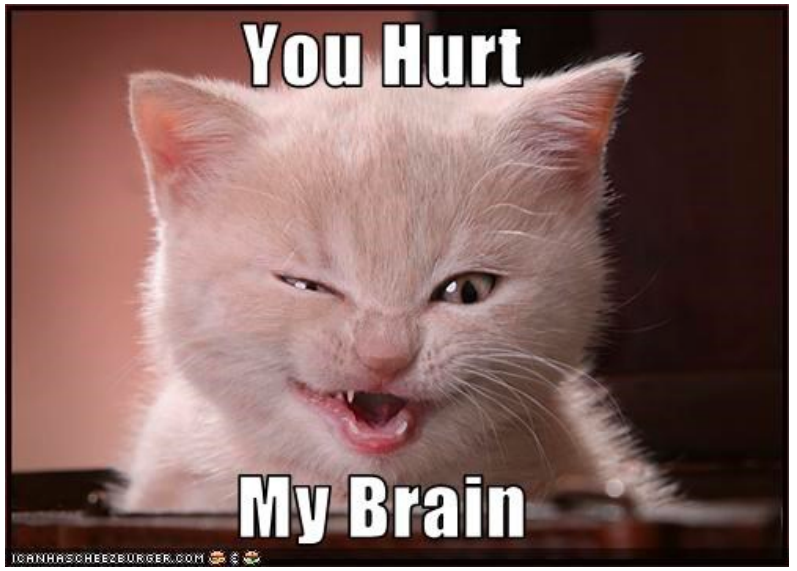
CPU Summary

Your CPU
Is
Fabulous!

Trust your CPU!
And your compiler too!
(But, verify…)



Runtime Summary



- Absolute **chaos**
- **Unpredictable** order
- **Competition** from threads, processes, cores, devices
- **Latency** and **instruction-order changes** due to:
  - (runtime) *values* of data objects
  - their *placement in memory* (memory alignment)
  - changes in (runtime) *branch-logic paths "taken"*
  - (runtime) Competition for resources *from outside your process*

Your algorithm **WILL RUN CORRECTLY**

CPU *Guarantees*:

① **CPU Cache Coherency** – consistent view of state across multi-level cache, and main memory

② **All dependencies are respected** *(enforced!)*

The C++ "As If" Rule

Conspiring to change your code *(unbeknownst to you)*

The C++ “As If” Rule (paraphrased)

- “The Chillin’ Rule”:

“It’s All Good.”

The C++ “As If” Rule (paraphrased)

- “The Cool Rule”:

“Relax. I got this.”

The C++ “As If” Rule (paraphrased)

- “The Casual Rule”:

“Dependencies respected. Everything else is casual.”

The C++ "As If" Rule (quoted)

C++11 Standard, §1.9/1

The semantic descriptions in this International Standard **define a** parameterized nondeterministic **abstract machine**. This International Standard places **no requirement on the structure** of conforming implementations.

<snip>, (emphasis added)

- The **C++ Standard** is **TWO things**:

1. **Explanation of (observable) behavior** *(which IS the standard!)*

   Must be respected! Is ***Well-Defined Behavior!***

2. **Definition of an abstract machine** to *explain* that behavior *(including sufficient structure to demonstrate the machine can be constructed)*

   Implementation is free to disregard! Is ***Artifact!***

The C++ “As-If” Rule (In Practice)

- The C++ compiler and CPU are ***permitted*** to perform **any and all code transformations** that do not change the program’s observable behavior.

Translation:
The **Compiler** and **CPU** can **do whatever they want**, as long as **logical dependencies are respected**.

Corollary:
We should ***expect*** the compiler and CPU to reorder the execution of our C++ statements at some level

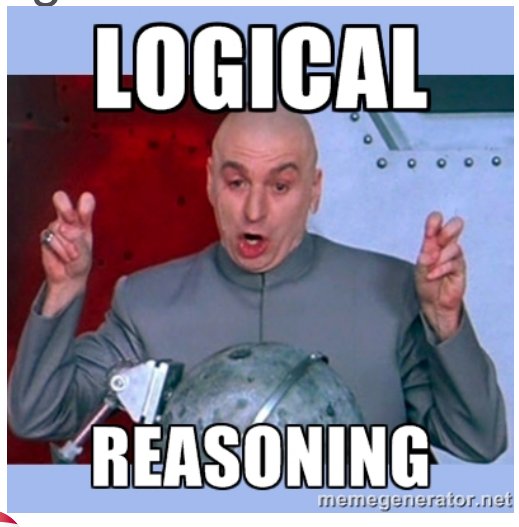
Compare "C++ As-If" With Other Languages

- **C Language** has the equivalence of the **"As-If" rule**
  - Lacks some subtle rules to gain intent (e.g., "copy elision")

Also designed to run on an actual CPU!

- Many **other languages** ("simpler"): Language specification merely **documents the intended behavior** of the implementations (including that which is explicitly unspecified) in the target intermediate representation specification
  - Might have a **"reference implementation"** that is "correct", where differing behavior of all other implementations are "wrong"
- **Some languages** strictly **specify formal language rules** without mentioning intent
  - Is similar to "As-If" rule when behavior is not explicitly specified

Intent != Behavior



C++ Expresses “Intent”

- std-discussion@isocpp.org
  - Richard Hodges, “Re: [std-discussion] throw std::exception with stack trace (portable)”
  - Sat-16-Apr-2016, https://groups.google.com/a/isocpp.org/forum/#!topic/std-discussion/A17G1ram9ns

<snip>,

“C++ is not like other languages. It expresses intent. The compiler transforms that intent into 'as if' code.

It is wiser to focus efforts on guaranteeing that the correct intent is specified.”

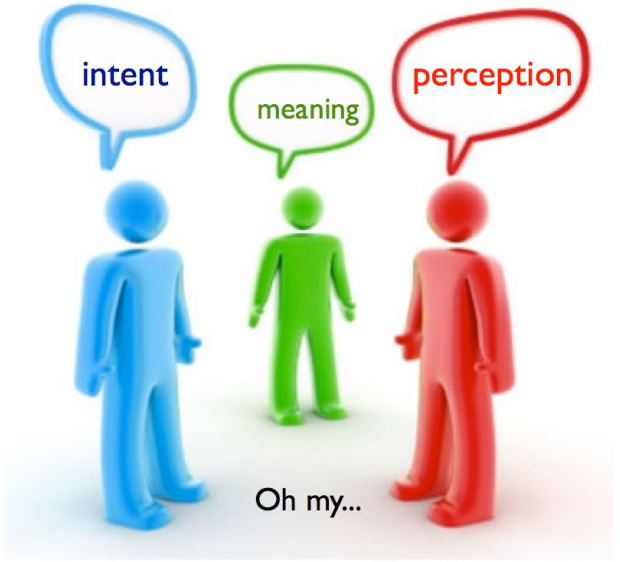
<snip>

C++ is designed to run *(efficiently!)* on an **actual CPU**!

Any “extra-sequence-guarantees” offered by other languages impose *(significant!)* efficiency penalties on the compiler, and CPU!

“Intent” is expressed through C++ Well-Defined behavior to **establish dependencies** within the algorithm.

Expressing "Intent"



- All **languages express "intent"** (somehow)
  - Language rules for "Well-Defined" Behavior
- **Intent allows "Optimizations"**
  - Respect "intent"; **Allow ANY AND ALL changes** that do **not undermine "intent"**
    - Copy Elision
    - Return-value optimization (RVO)
    - Function Inlining
- All languages **run on** an **actual CPU**
  - Possibly with many levels of "indirection", "**translation**", "interpretation"
  - Dependencies ("intent") are respected

Controlling for "intent" permits **MAXIMUM FREEDOM** to perform **optimizations**!
(Does not control for "artifacts")

Algorithmic "Intent" is expressed by establishing (logical) dependencies

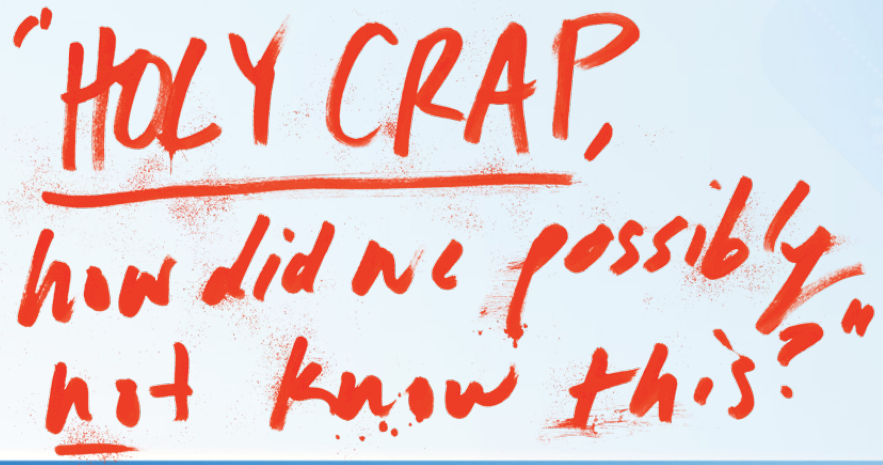
Influencing Instruction Order

When you *demand* more control

Controlling Instruction Order

- **Yes, You Can!**
- **Pros/Cons** controlling instruction order
- **How**

<insert content here>



Hyper-Threading (HT) and Simultaneous Multithreading (SMT)

Fighting Over Resources *Like Siblings*

HT/SMT: Fighting Over Resources Like Siblings

- What it is
- How it works

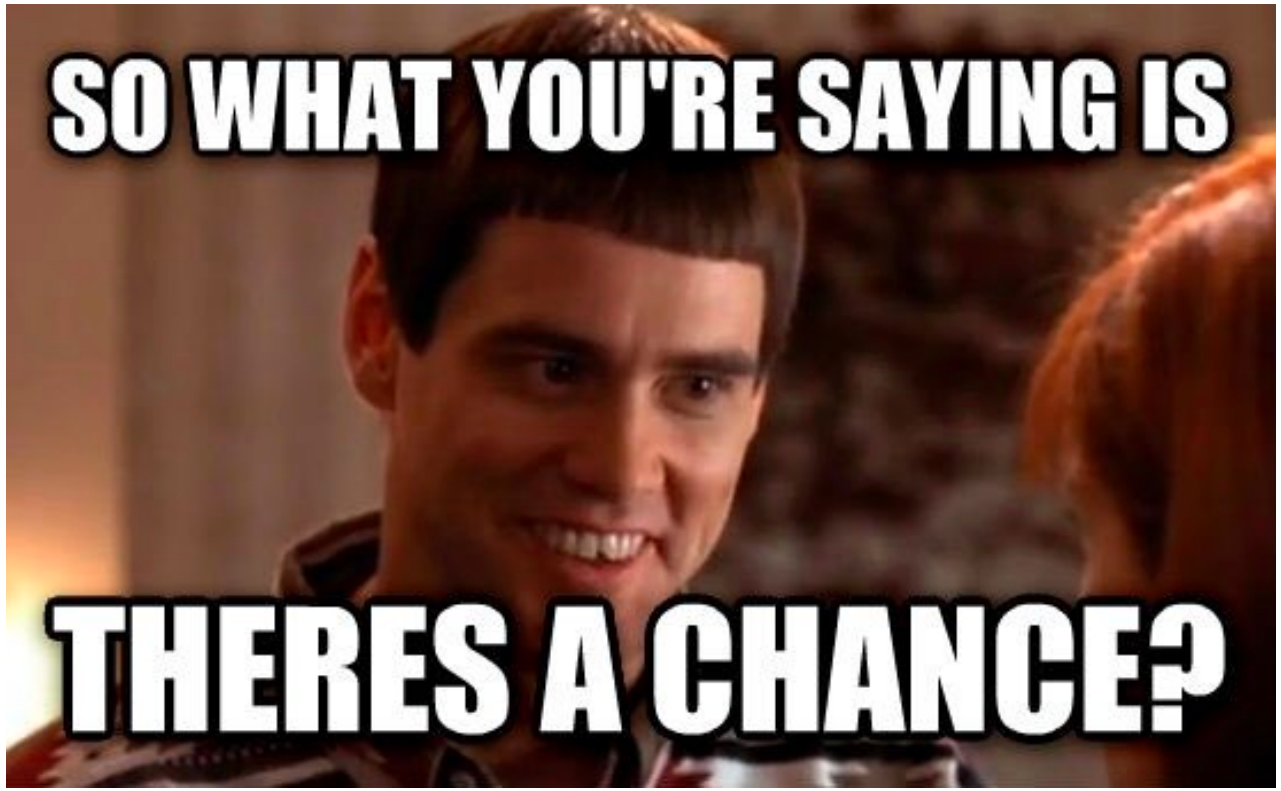
<insert content here>

Summary
The “take-away for today”

Stop Thinking Imperatively (don’t assume order)

Instruction Order Summary

- There is "**a chance**" that your statements **execute in the order you specify** in your C++ source code file.

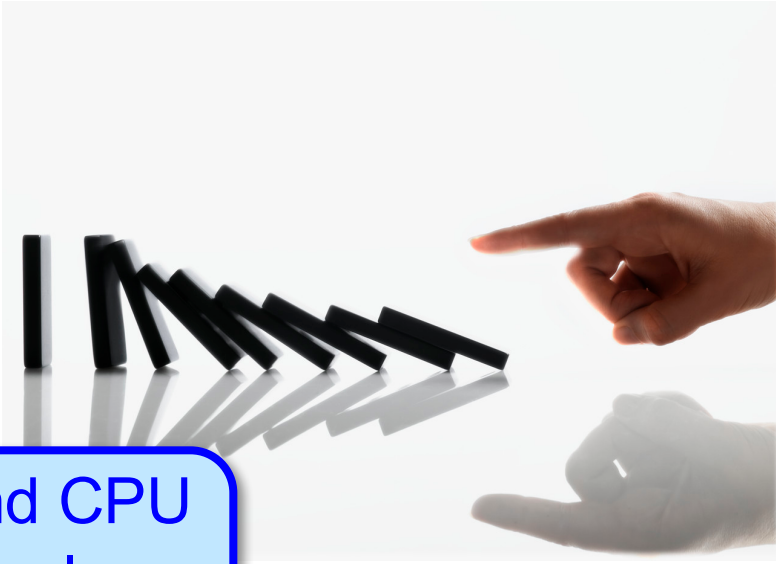


Dumb and Dumber (1994)

- …But ***not a very good one***.

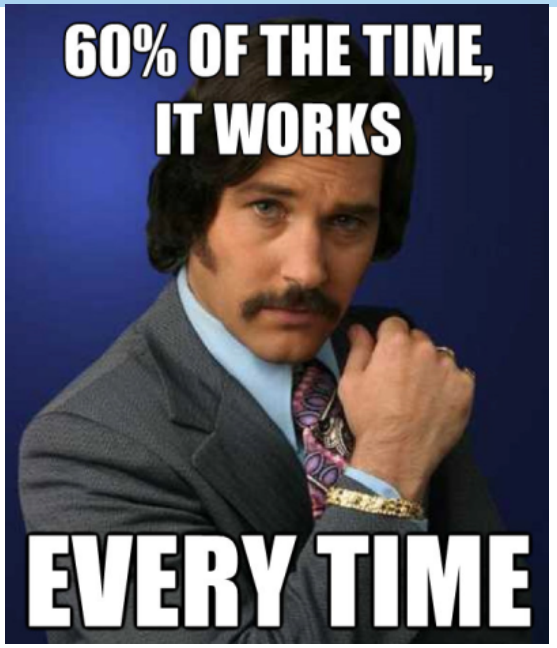
Logical Sequence: Defined by dependency

- **Any order** that respects dependency order is logically equivalent.
  - **Logical Sequence** is **defined by dependencies** *(we care!)*
  - We care **not at all** about any other order
- With explicit **focus on dependency management** in design and implementation:
  - Our single-threaded implementations are *faster*
  - Our multi-threaded designs are *easier*
  - Many irrelevant details disappear from discussion
  - We best exploit technologies within compiler and CPU



Granting **maximum latitude** to the compiler and CPU regarding order **leverages** the underlying technology to get the **fastest and most efficient** programs

We Control For Dependencies



- **Dependencies are the basis** for how we reason
  - **Dependencies are respected** by the compiler and CPU (or all is lost)
  - If **no dependencies** present, **order is uncontrolled**
- The **C++ Language**, the **compiler**, and the **CPU** are based on well-defined behavior to **manage dependencies** *(not "instructions", not "statements")*
  - They all **AGREE** on what are **"dependencies"**
  - They all **DISAGREE** on what is an **"instruction" or "statement"** *(no commonality)*
- Defining (or relying upon) **"physical order"** is **problematic** – don't try

Why would ANYONE think "instructions" (or statements") would execute "in-order"? It is **HIGHLY DESIRABLE** that the order be ***Uncontrolled***!

Dependencies: How We Design, Implement

1. Make **No Assumptions regarding "order"**
   (dependencies are respected, "order" is not)

2. **Over-Specifications restrict efficiencies**
   (specify only those minimal dependencies required)

3. *C++ As-If Rule:*
   The **Compiler** and **CPU** can **do whatever they want**,
   as long as **logical dependencies are respected**

A Bonus: By focusing on dependencies, our design often changes
- More **scalable**, **flexible**, **efficient**, **simpler** *(fewer assumptions)*

Essence Of Parallel, Concurrent

Best Practice:
Go **Parallel when you can**,
Concurrent when you must

1 *Essence of **Parallel**:*
No Dependencies
(is about leveraging the hardware)

Design to be simple and obvious!

2 *Essence of **Concurrent**:*
Interacting Dependencies
(is about managing complexity)

Design to algorithm desired!

Well-Defined Behavior

Well-Defined Behavior
- Comes from ***logical*** sequence
  - Defined by logical dependencies

Respected! Enforced!

Undefined Behavior is **always repaired** by establishing **logical dependencies**!

Undefined Behavior
- Comes from ***physical*** sequence
  - No dependencies exist, so order *cannot* be enforced

Rodney Dangerfield

November 22, 1921 – October 5, 2004

Patron Saint of Physical Sequence

The Lego Movie (2014)

Questions?