ECEN 3753: Real-Time Operating Systems

Diagramming and Unit Testing

Why do we diagram?

- To abstract what's important
- To analyze critical connections/info
- To look for errors
- To check for completeness
- To communicate important info

There is no single reason, nor a single best solution.

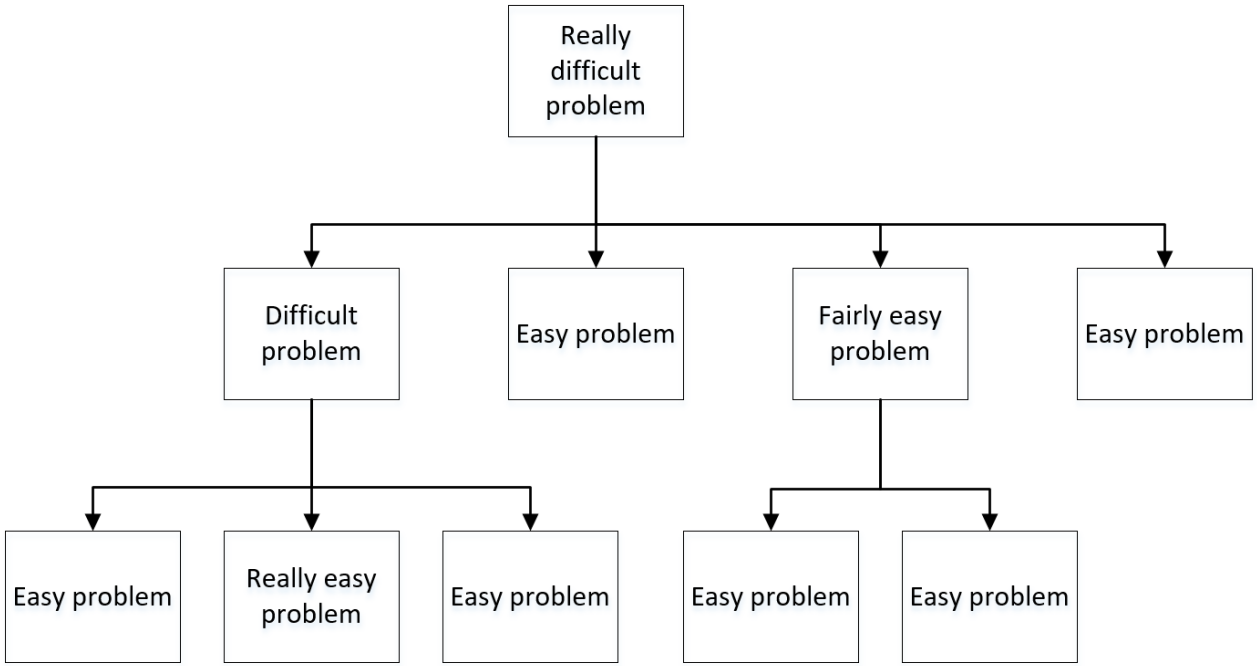
Some popular types of diagrams

As listed in the main lecture series:

- **Structure Diagram**: Repeatedly divides a problem into smaller and smaller pieces
  - Useful for communicating top-down or bottom-up design
- **Data Flow Diagram**: Data and Transformations on data
  - The basis for what we call a Task Diagram, where:
    - Tasks are the "Transformations"
    - Shared data and Inter-Task Communications are the "Data"
  - Helps us see where data interfaces might be more obvious for unit test boundaries
- **State (Machine) Diagram**: States and transitions/triggers
  - Sometimes very useful to help analyze different states of critical data that may align with triggering of data transformations
- **Sequence Diagram**: How parts of a system do their part (computation or signaling) to fulfill a use case's demands
  - *Optional* example perusal:
    - SoftwareIdeas.net: Sequence-Diagrams

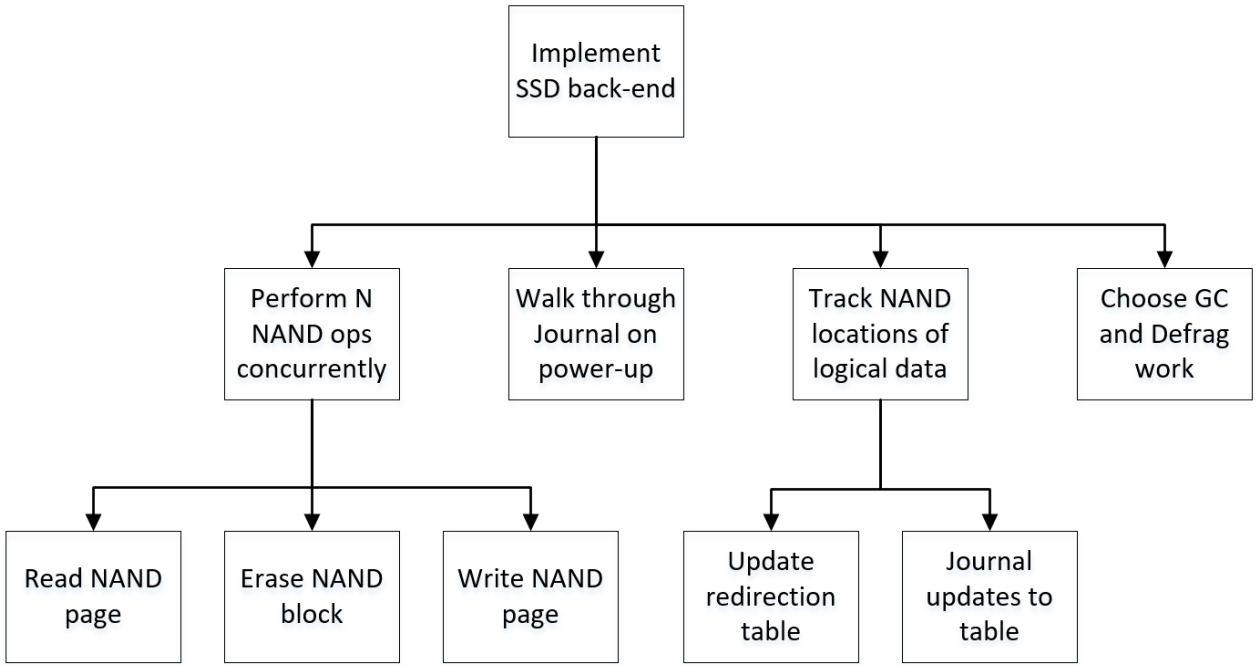
Generic Structure Diagram

While the general form of this may not seem like a big deal, it is a very powerful tool, regardless of whether you approach a problem from the top or from the bottom.



SSD Back-end Structure Diagram

For instance, that same structure shape can describe much of what is required to provide an SSD's back-end (the part that actually stores data, as compared to communicating with the server or PC it's plugged into).



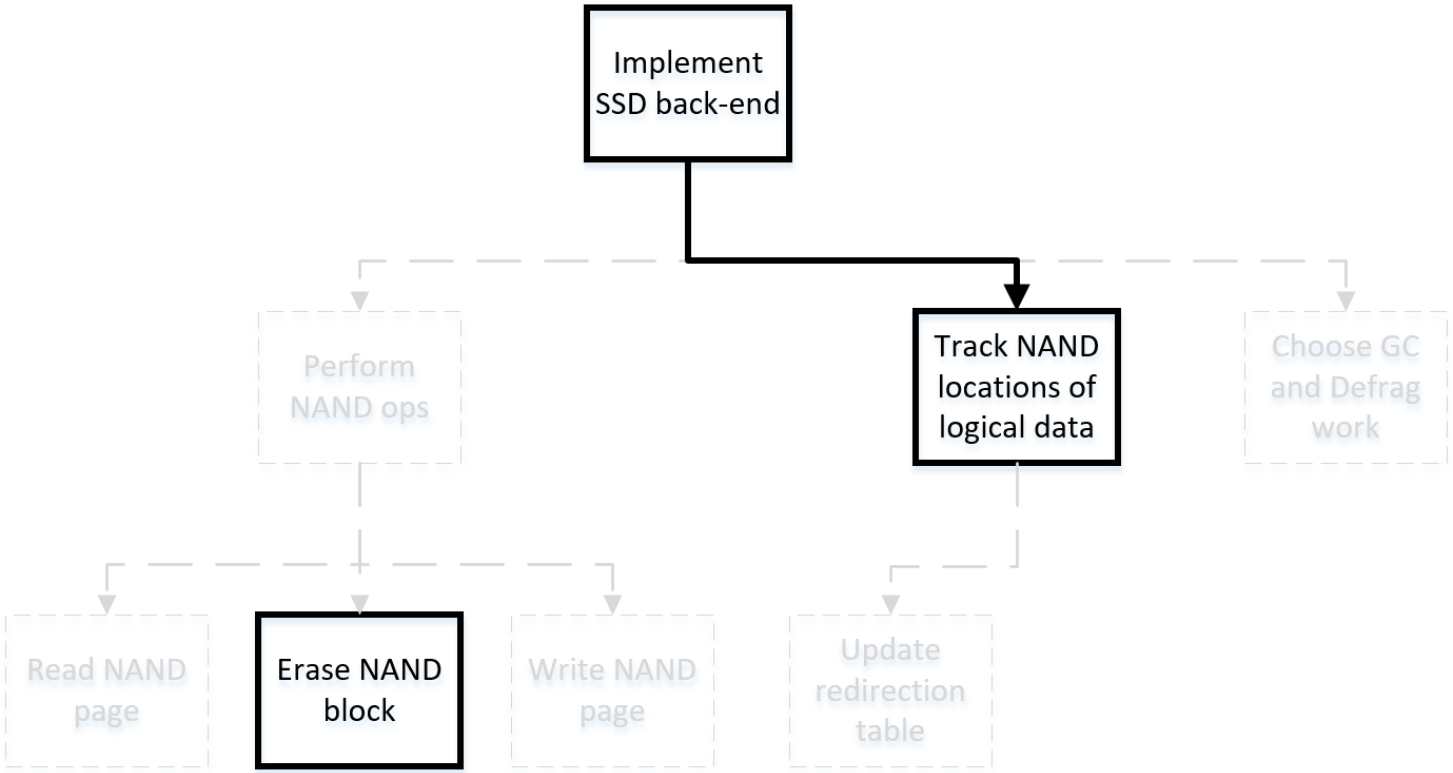
Top-Down vs. Bottom-Up Design

Neither is the *only* goal.

- Some people are more comfortable thinking in abstractions without needing to know the details at the bottom of the diagram.
- Others are more comfortable connecting to some low-level hardware or interfacing to a chip spec, without needing to know how the other parts of the system need to use that or do other work.

Most systems will require some top-down, and some bottom-up design, regardless of personal preferences.

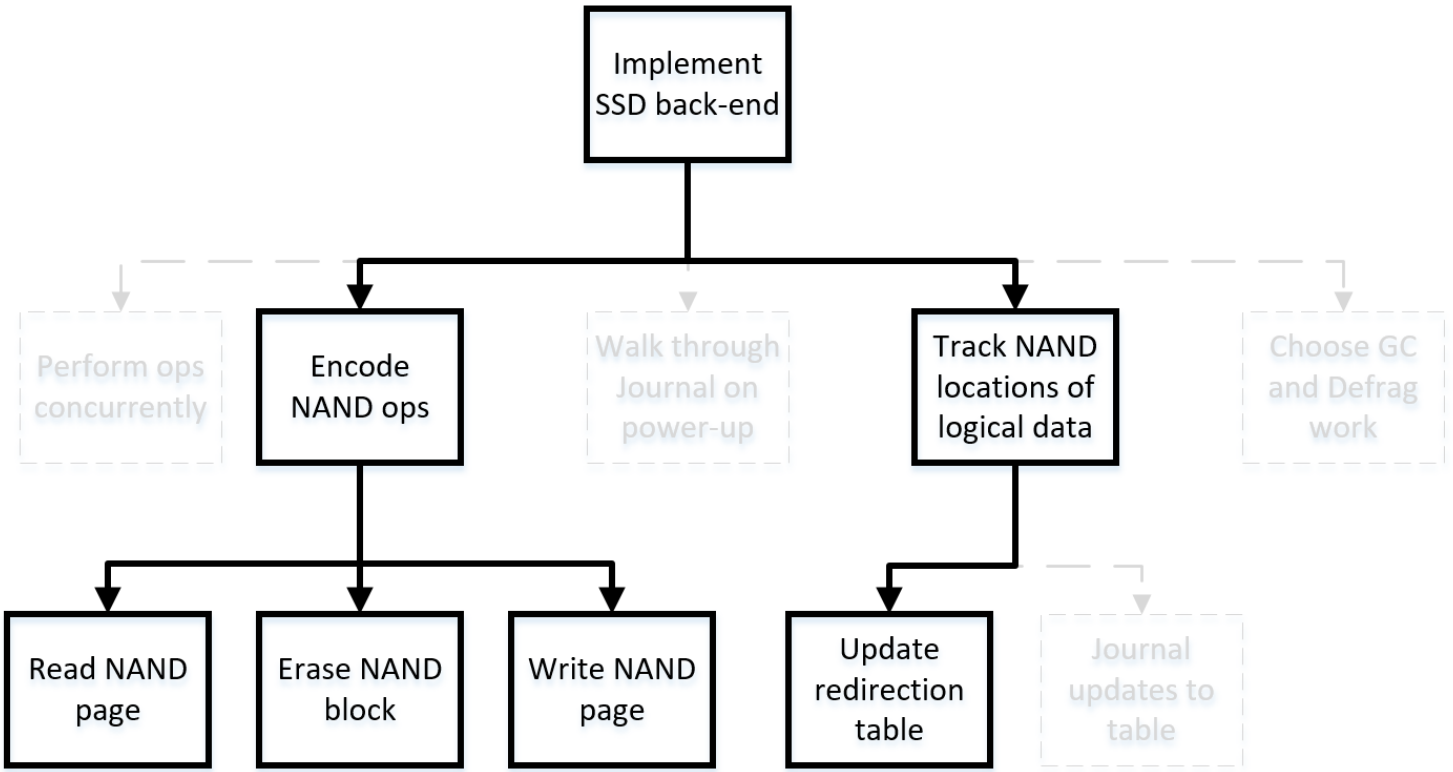
Example of a possible progression (1/4)



Two people might start with some vague ideas (gray, dashed), and some ideas that are getting *real* (black, solid):

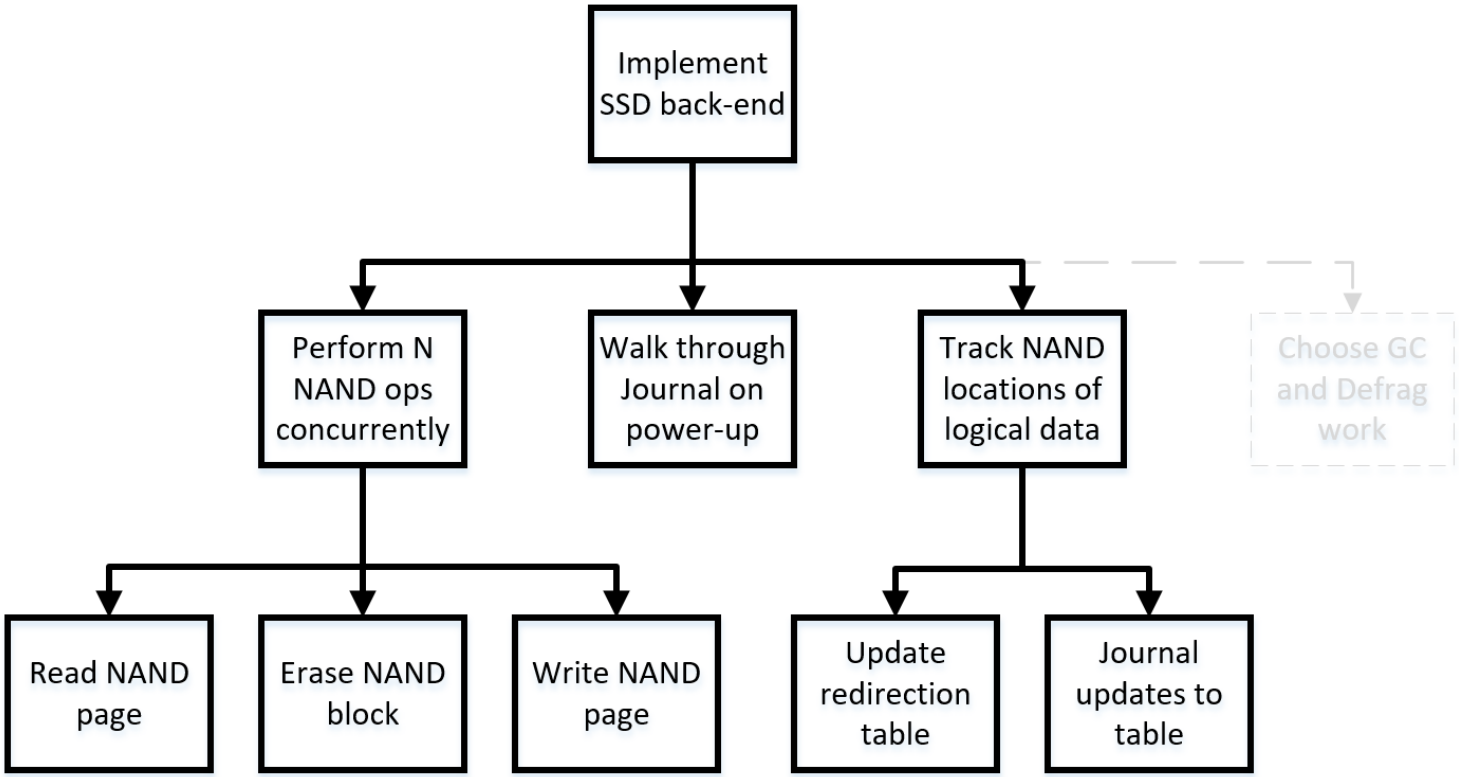
1. Need to map logical data addresses to changing physical ones
2. Start figuring out how to access Flash, without data payloads

Example of a possible progression (2/4)



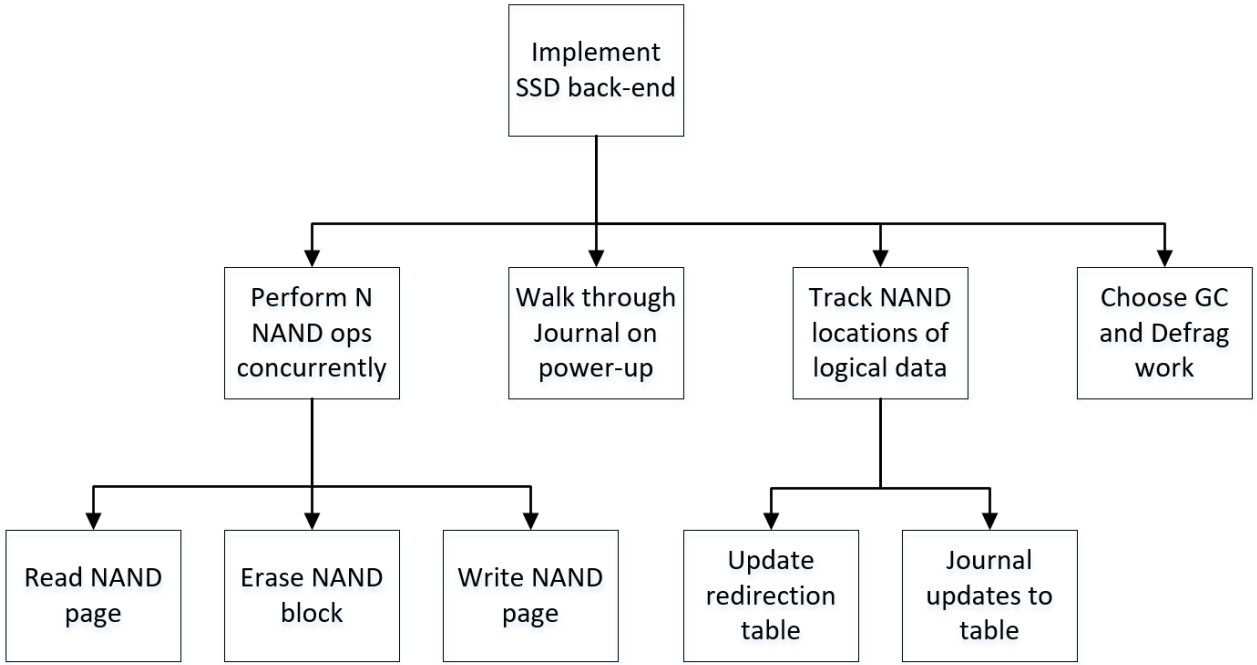
1. Mapping *use* leads to updates *to* the map
2. Generalization of one op leads to ideas of how to do many, concurrently

Example of a possible progression (3/4)



1. Mapping updates require journaling for power-loss protection
2. Concurrency in hardware for all required ops

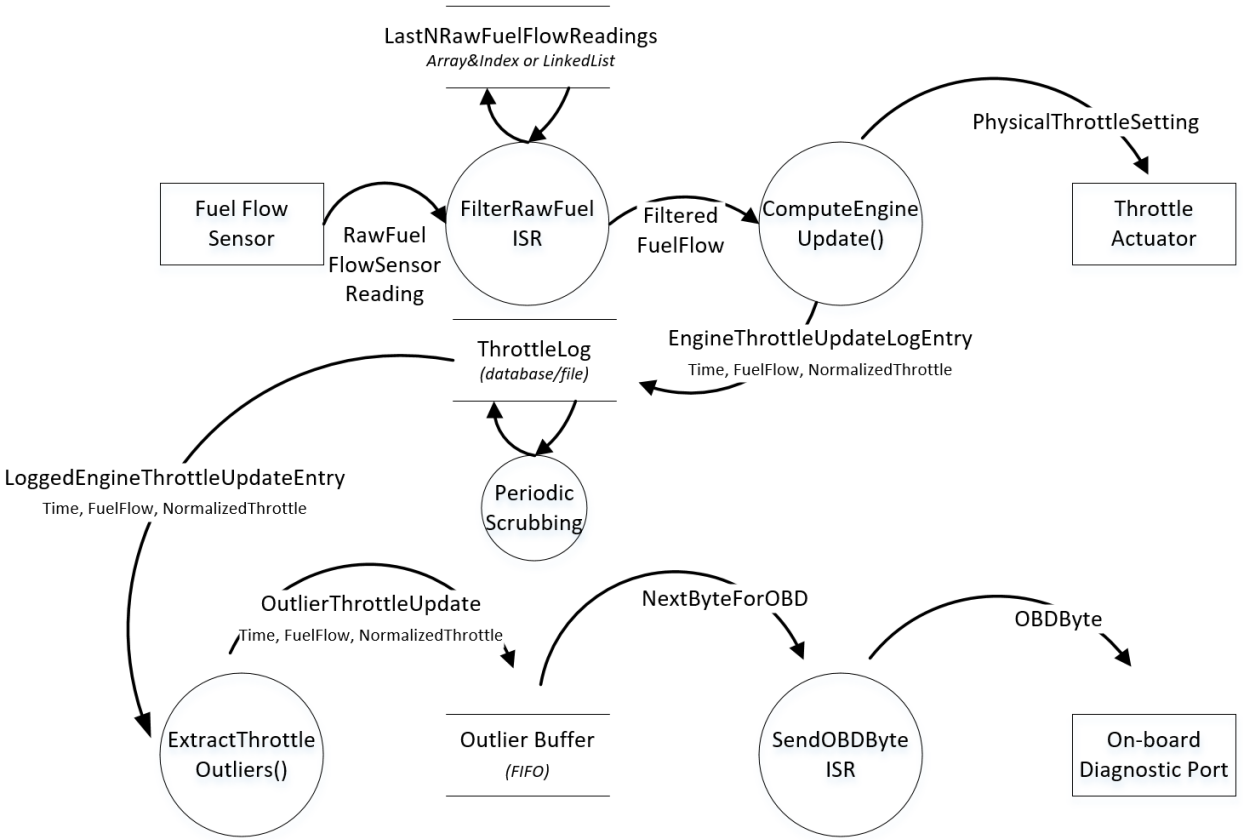
Example of a possible progression (4/4)



Finally, the whole design is done. Some (especially lower-level) solutions may already be coded from the process of building understanding. *Assume* that most of that should be re-designed/re-written, or get used to disappointment.

Data Flow Diagram

. . . the next diagram with very broad appeal to us as designers of software systems. It shows data (stores, or transient) and the transformations of data, as it exists between inputs and the outputs.



Identify Unit Test Boundaries

The question arises: Where should we cut that picture and implement testing of parts without the whole?

Read Now:

- Pragmatists.com: Test doubles, fakes, and stubs

Optional:

- MartinFowler.com: Mocks aren't stubs

Targeting Appropriate Unit Testing

- Why
- What
  - For this class
  - More generally
- Example
  - Scheduling

Why Unit Test?

By testing pieces before the whole solution is ready, you move the detection of design or implementation bugs earlier in your work, and are testing smaller (simpler) pieces. ("Shift Left" is a reference to a schedule)

Is this benefit free?

Why Unit Test?

By testing pieces before the whole solution is ready, you move the detection of design or implementation bugs earlier in your work, and are testing smaller (simpler) pieces. ("Shift Left" is a reference to a schedule)

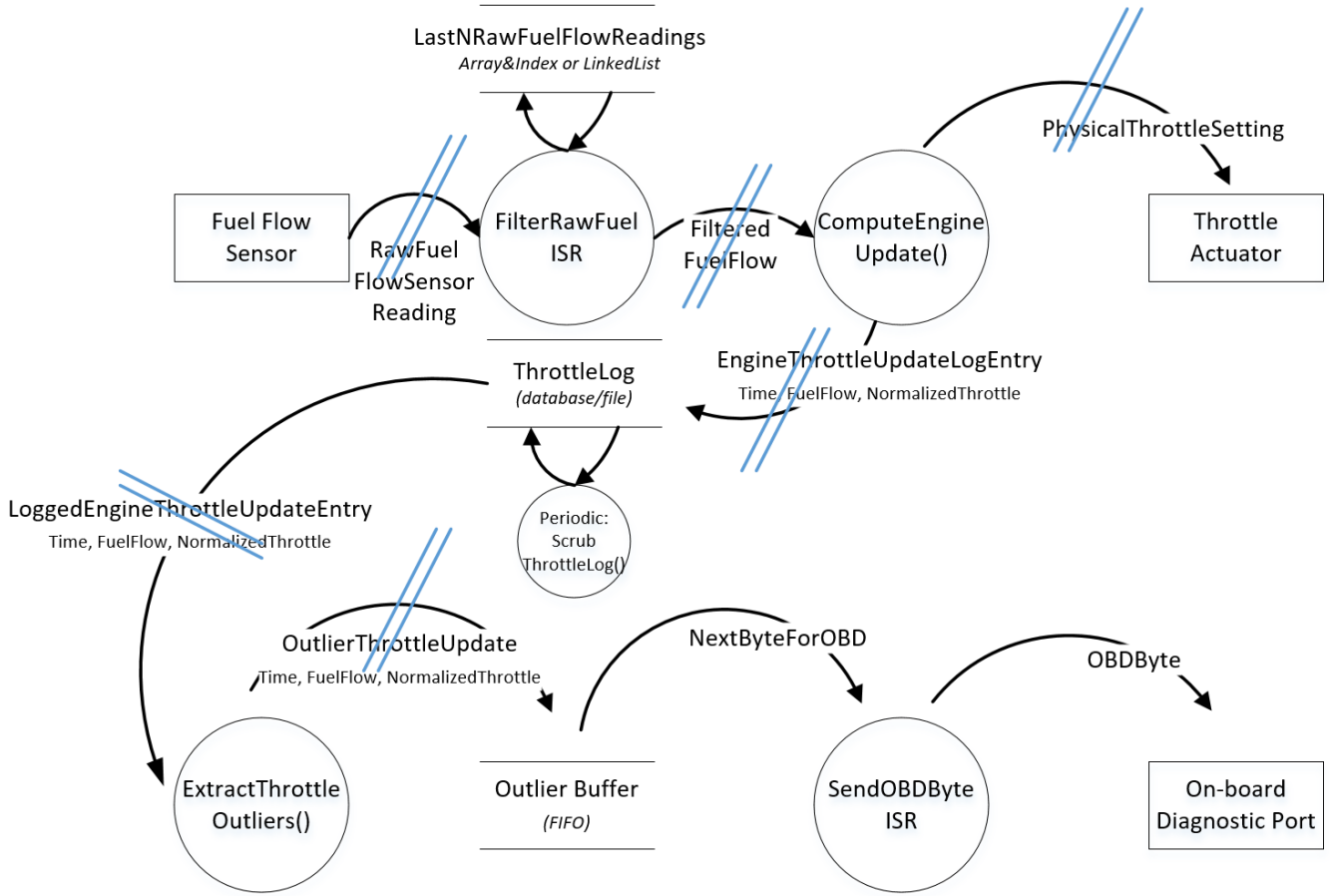
Is this benefit free?

Of course not. Effort has to go into buiding a UT framework, mocks/fakes, building the test case implementations, and figuring out exactly what the UT outputs should be. The gamble (with **excellent** odds) is that debug using UTs is less *difficult* debug than system-wide debug you'd otherwise need more of later, so there is a net benefit.

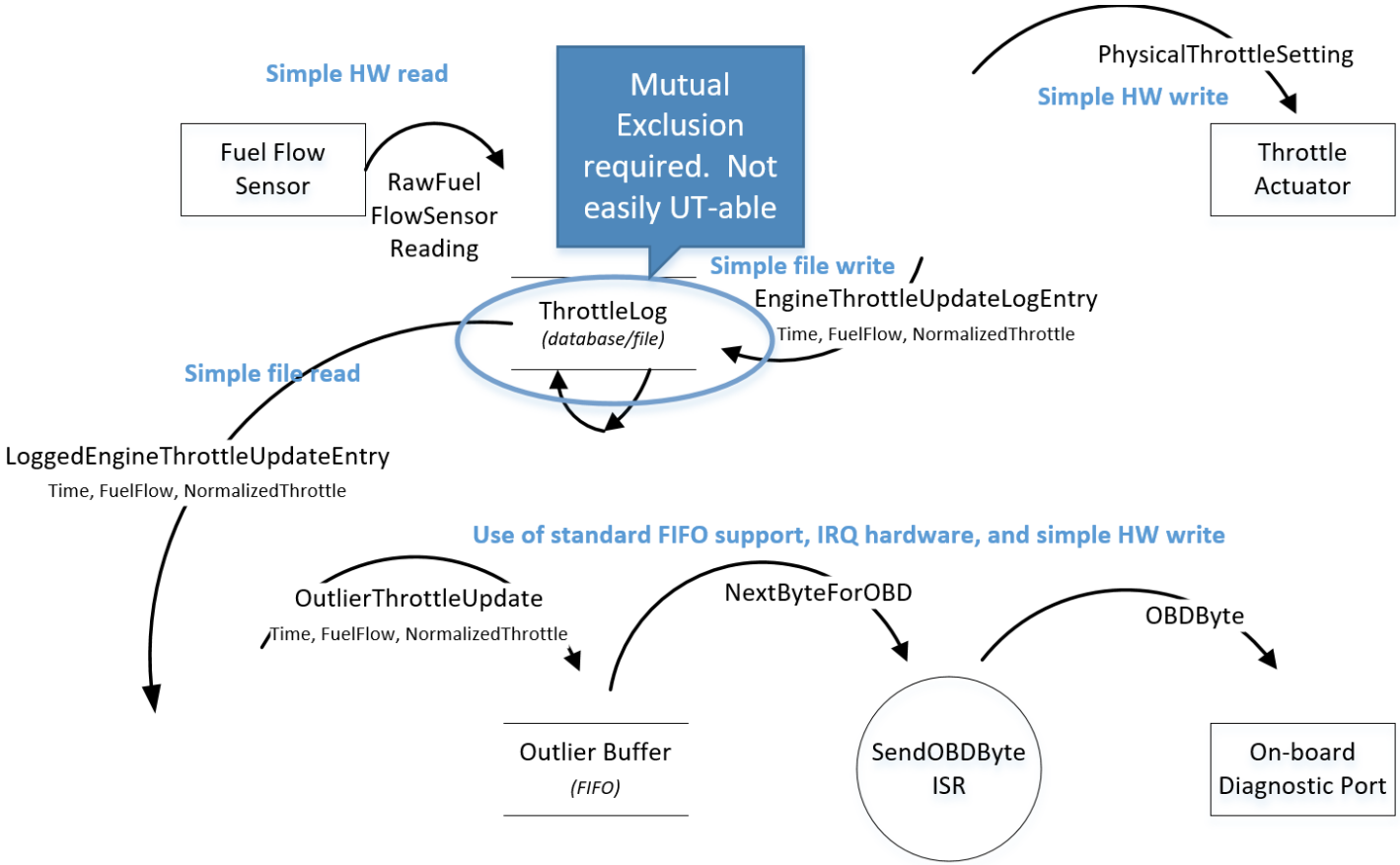
What should we Unit Test?

- "The stuff between parameters"
- In this class–Algorithms
  - Scheduling choices–Labs 4,5
  - A RT algorithm in your project, where it is separable from the hardware
- In general
  - It still depends, on maturity of HW (design/implementation), level of code reuse/maturity, tolerated risk parameters.
    - HW not yet made: HW Sim can support UT, or mocks/fakes can be built to allow much faster testing (with less precision).
    - HW still immature: sometimes UT (in HW sim) can allow code to be quickly verified on an updated HW design (Verif/Val)
    - If code is significantly reused generation-to-generation in products, a large amount of change will still go into new generation SW, which causes risk to "known working" SW.
  - Higher-risk, easier-to-UT code would be easier to justify than harder-to-UT, or very low risk (RTOS mechanisms, IRQ HW)
  - Hugely useful as part of speedy regression testing.

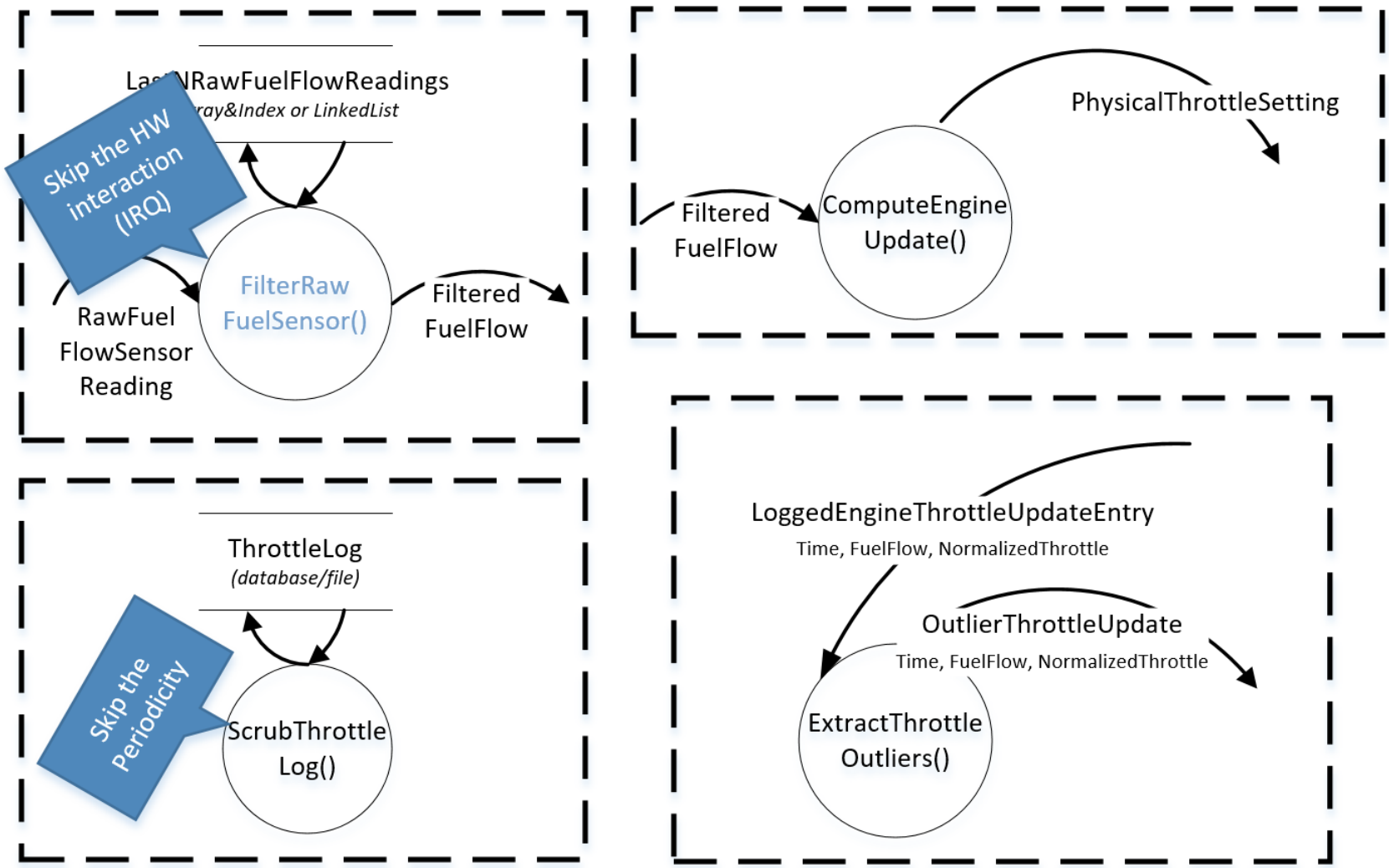
Data Flow Diagram, with Cuts



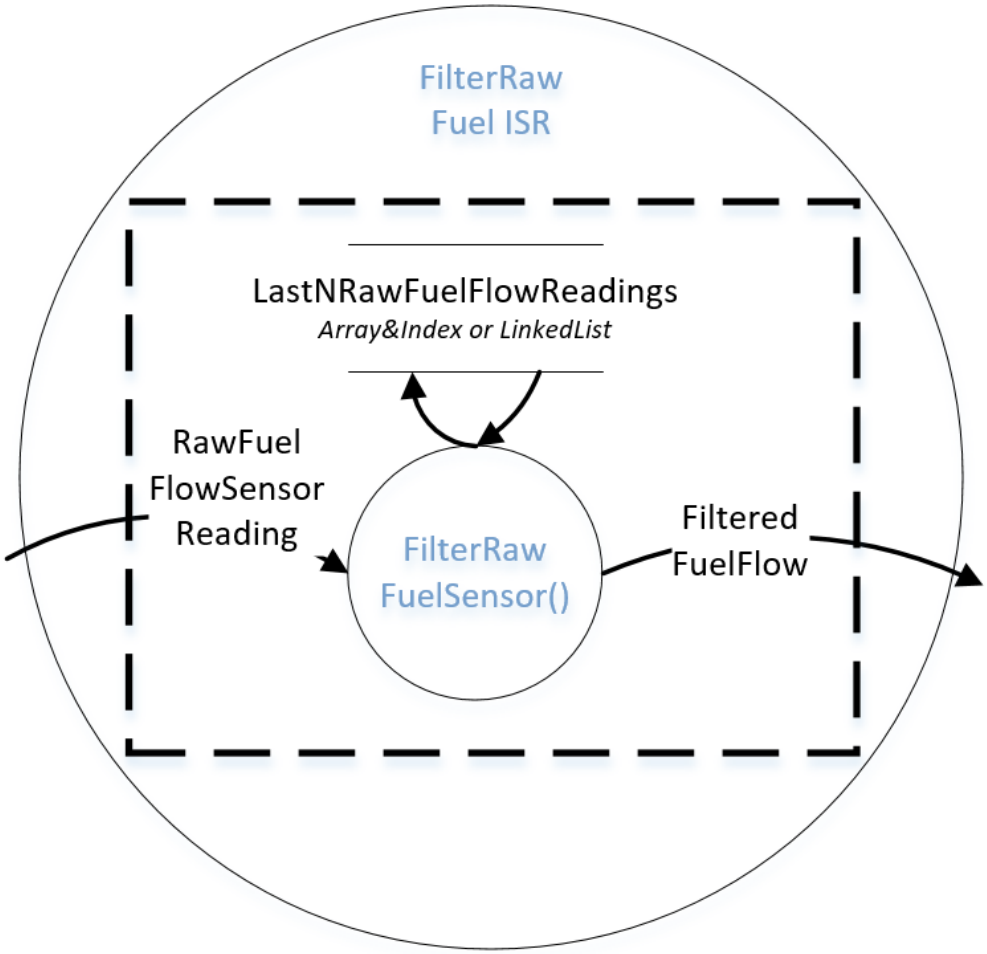
Judgement: What NOT to Unit Test?



Judgement: What TO Unit Test?



Unit Testing the Guts



Unit Testing in this Course

Prior to Lab 4 (you will write unit tests to verify your own scheduling implementations):

- Your TA will introduce you to the unit testing framework that you'll use to:
  - Initialize a test
  - Run the test
  - Verify results of the test

Starting from that, for Labs 4 & 5 you will build multiple tests in a suite, and possibly multiple suites, that check behavior beyond the ends as well as the middle of healthy parameter ranges, etc.

In your project, you'll need to plan your own unit tests, and execute them as part of your proof-of-demonstrability, along with functional tests. You must submit all of your test plans and results BEFORE the actual demo.

Onward to State Diagramming

Imagine that dual-copies are used to protect some really important data.

- Usually:
  - Write(A), Write(B) to save update.
  - Read (A) to read back up at power-up.

Let's use the 7-step programming method to derive some appropriate behavioral responses.

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B)

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails$\Rightarrow$ Write(B)

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails$\Rightarrow$ Write(B)
- Read(A) fails, Read(B) fails

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails$\Rightarrow$ Write(B)
- Read(A) fails, Read(B) fails$\Rightarrow$ wave down a passing car

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails$\Rightarrow$ Write(B)
- Read(A) fails, Read(B) fails$\Rightarrow$ wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails$\Rightarrow$ Write(B)
- Read(A) fails, Read(B) fails$\Rightarrow$ wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

- Read(A) and Read(B) succeed, but S(A) $>$ S(B)

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails$\Rightarrow$ Write(B)
- Read(A) fails, Read(B) fails$\Rightarrow$ wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

- Read(A) and Read(B) succeed, but S(A) $>$ S(B)$\Rightarrow$ Write(B)

Duplicated Boot-up Data Recovery

Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds $\Rightarrow$ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails$\Rightarrow$ Write(B)
- Read(A) fails, Read(B) fails$\Rightarrow$ wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

- Read(A) and Read(B) succeed, but S(A) $>$ S(B)$\Rightarrow$ Write(B)
- Can we only Read(A) on power-up to save time (as long as it's read-able)?

Duplicated Boot-up Data Recovery

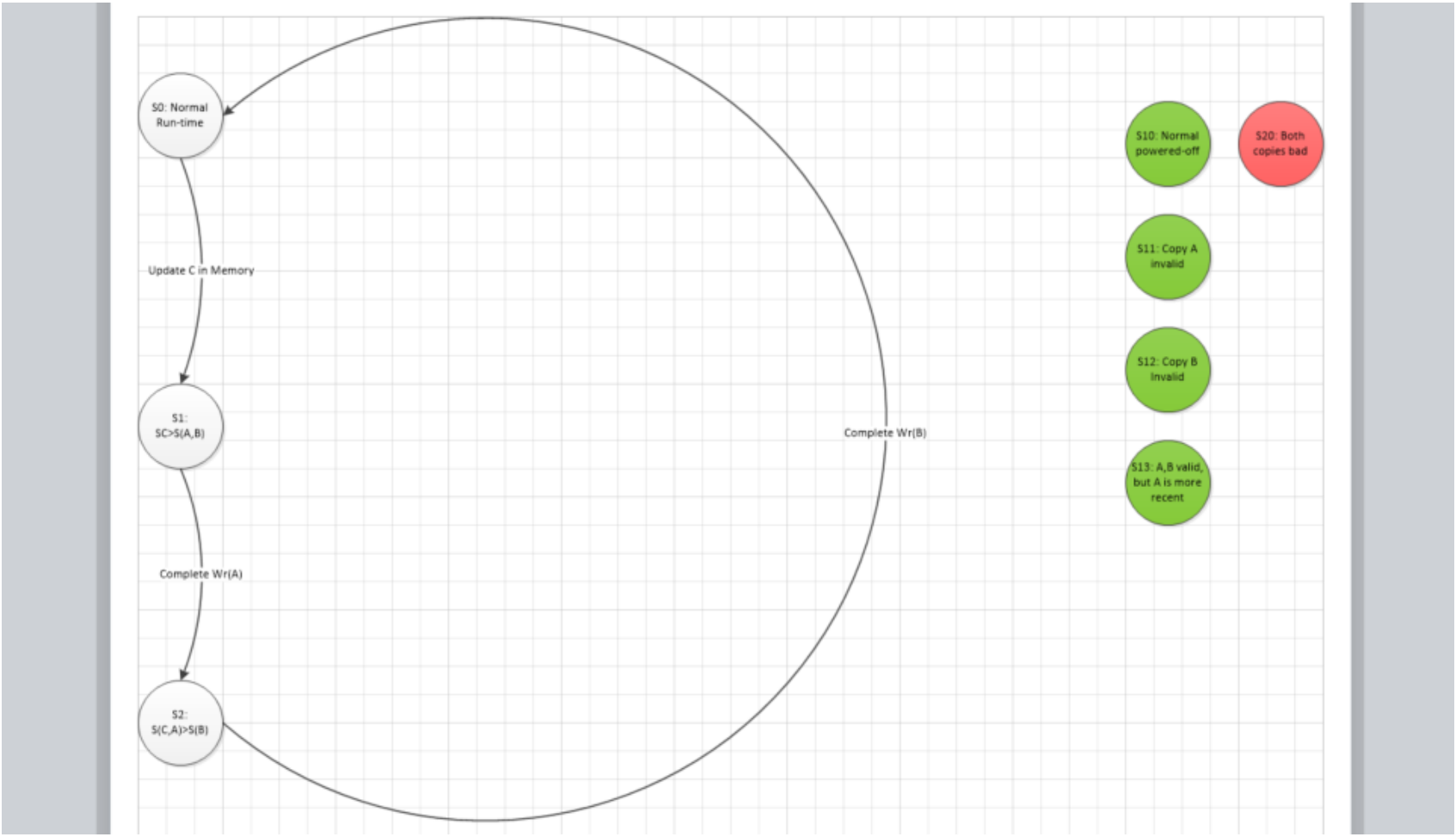
Using the 7-step programming method, derive some appropriate behavior:

- Read(A) fails, Read(B) succeeds ⇒ Write(A)
- Read(A), Read(B) : No writes needed
- Read(A) succeeds, Read(B) fails⇒ Write(B)
- Read(A) fails, Read(B) fails⇒ wave down a passing car

We're going to turn that into a complete state diagram for the non-volatile states of the duplicated data, using sequence numbers to identify older/newer copies, denoted "S" in the diagrams.

- Read(A) and Read(B) succeed, but S(A) > S(B)⇒ Write(B)
- Can we only Read(A) on power-up to save time (as long as it's read-able)?⇒ Look at the prior bullet.

Simple flow when update is made, and no problems arise

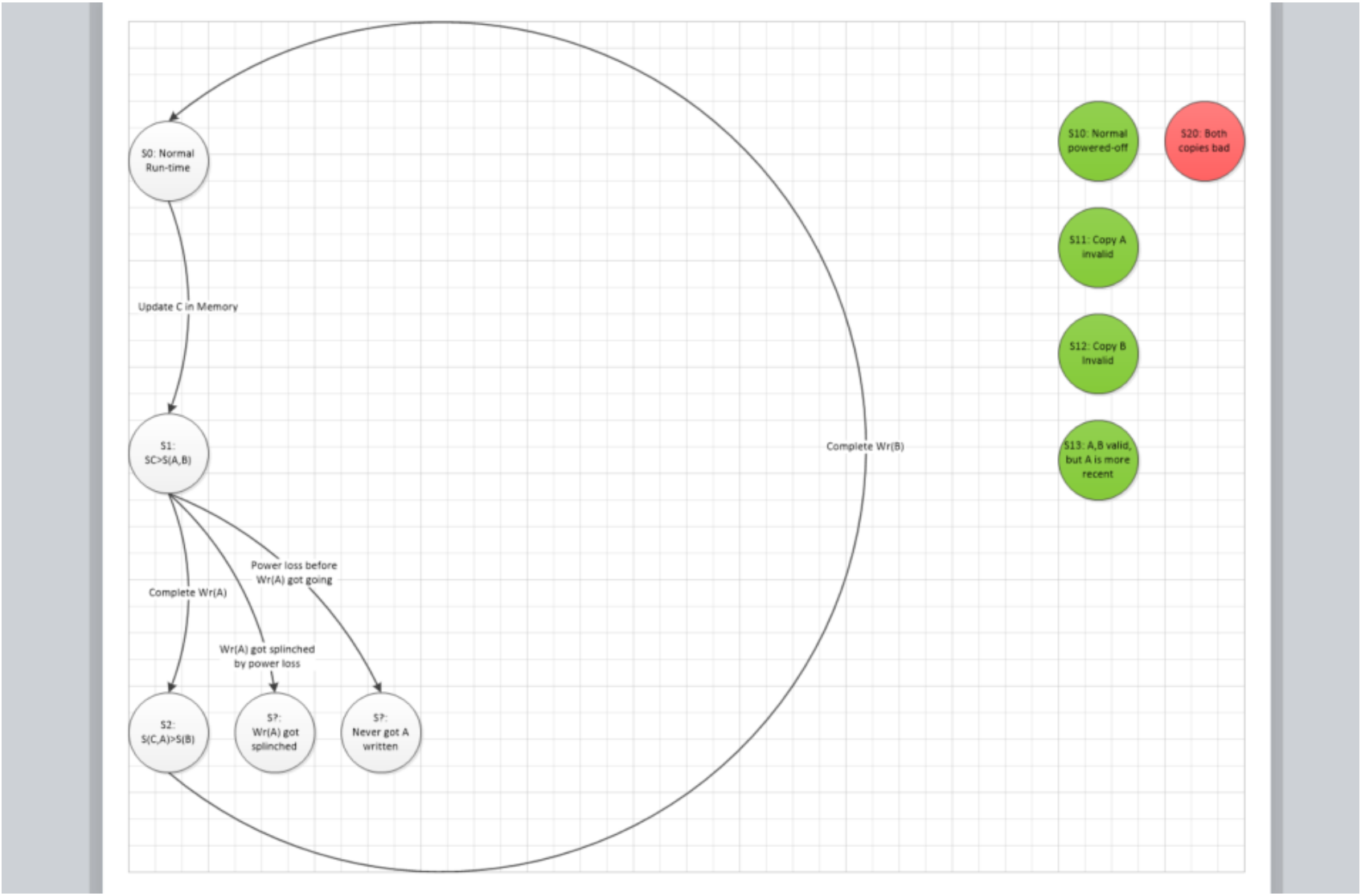


Who has Read-or-Watched Harry Potter?

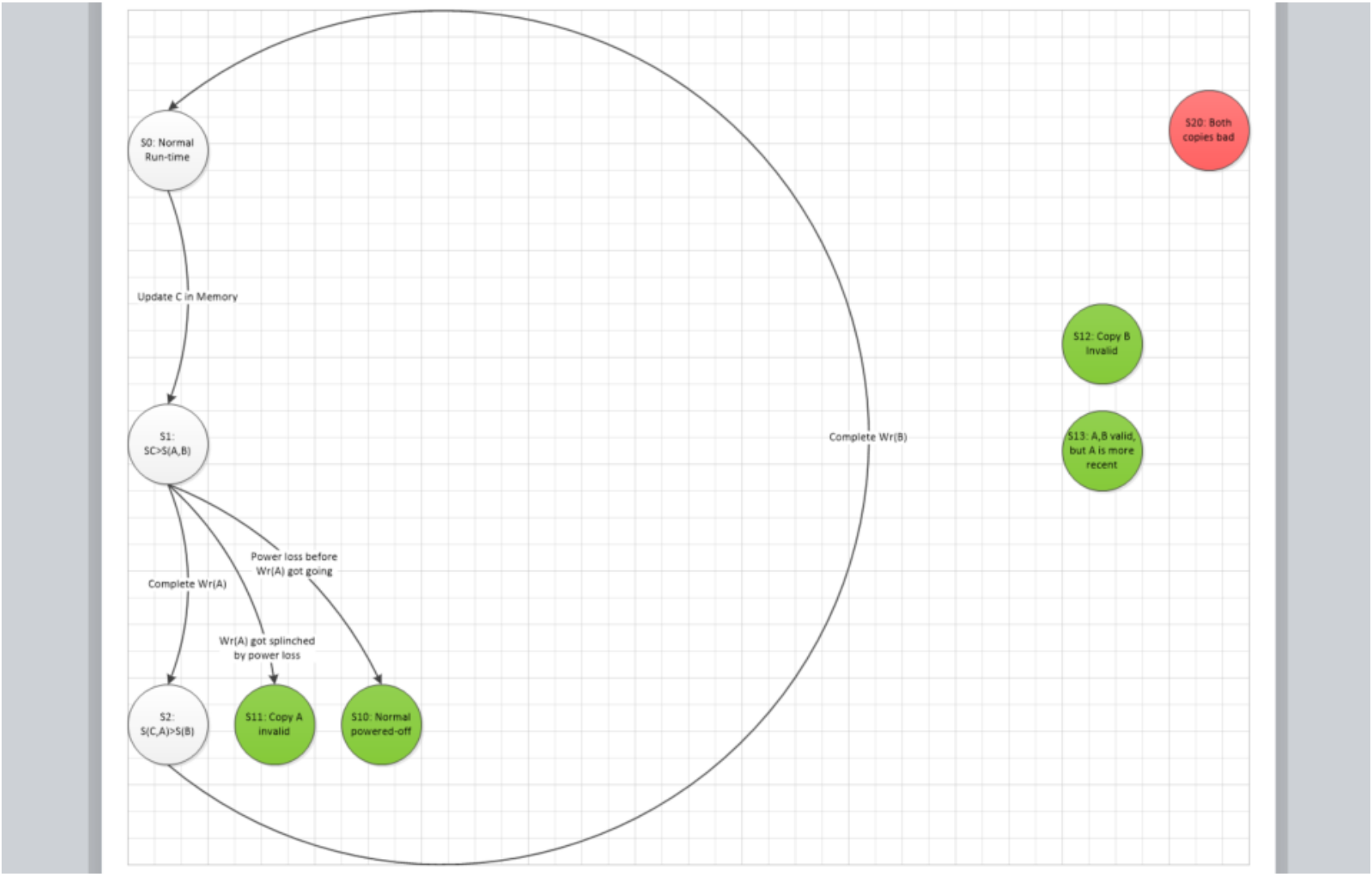
If a Write is interrupted by an unanticipated power-loss, the data is "splinched" – that is, the earlier part of it is from a data update, but the rest is what was there from a prior write.

In systems with Error Detection Coding, this will be detected as "bad data", rather than "Oop, I've been splinched!" Let's assume that we have EDC.

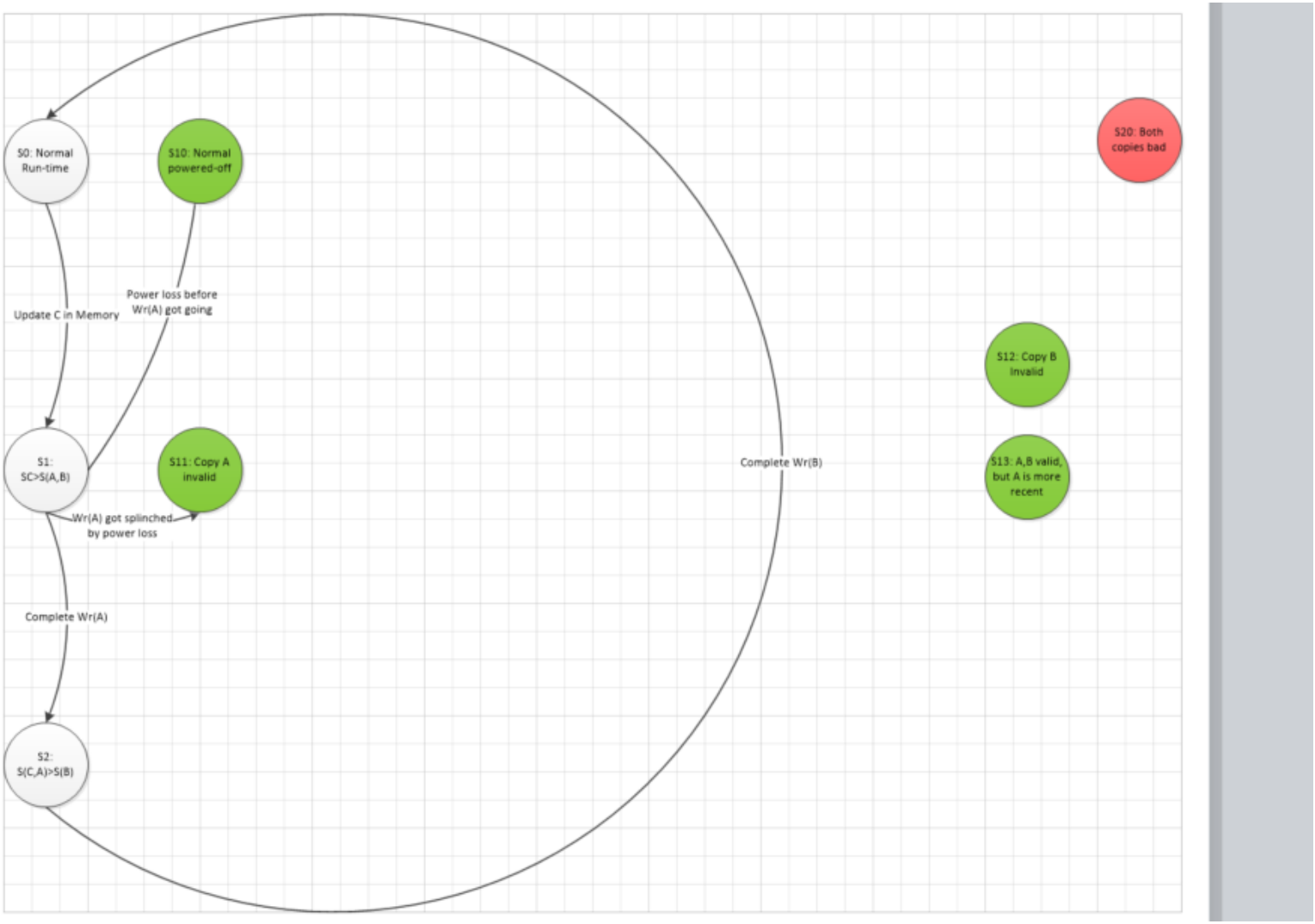
Add the 3 Wr(A) outcomes (success, splinch, not started)



Note equivalence to the NV states from the right

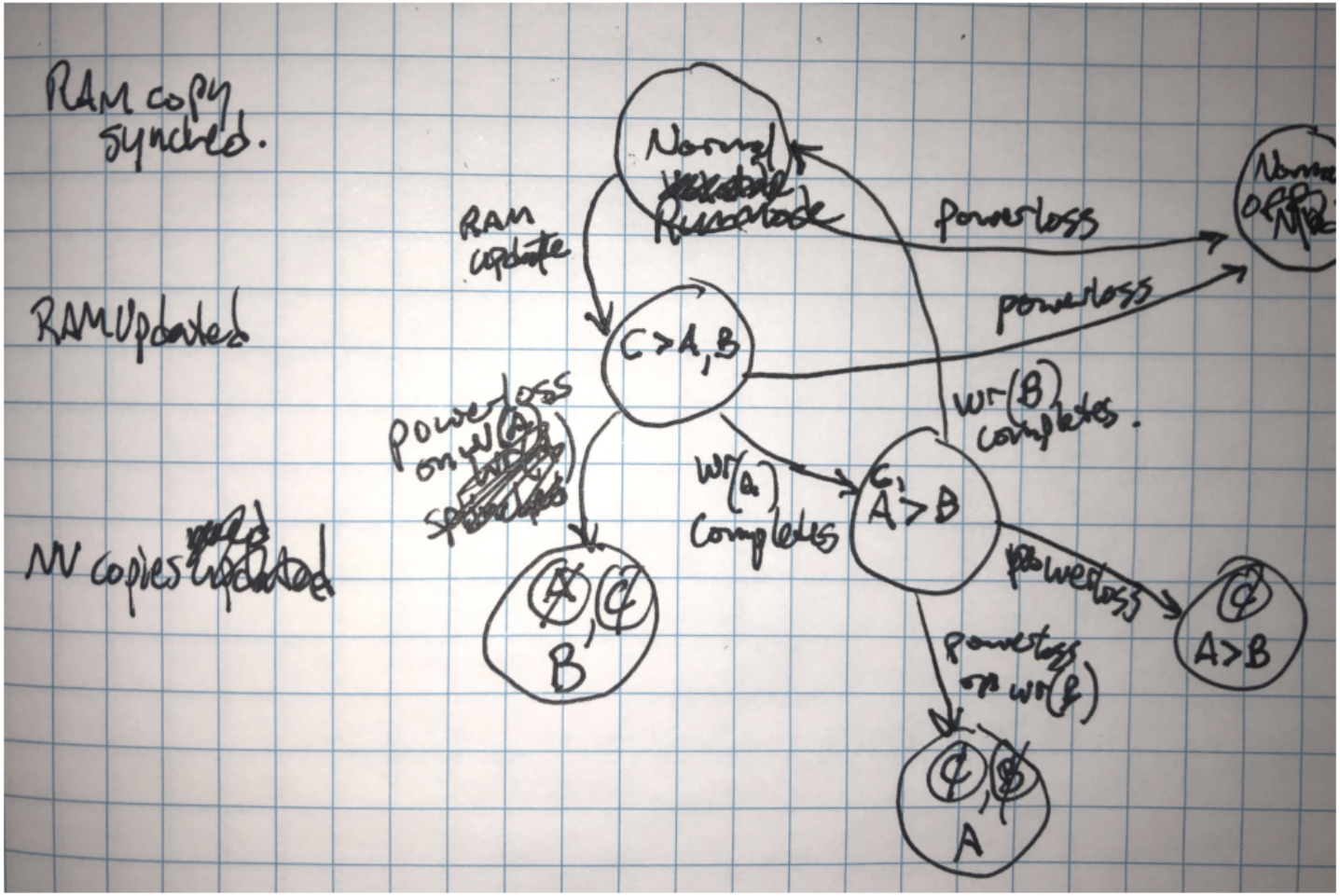


Now scoot the states upward in the diagram



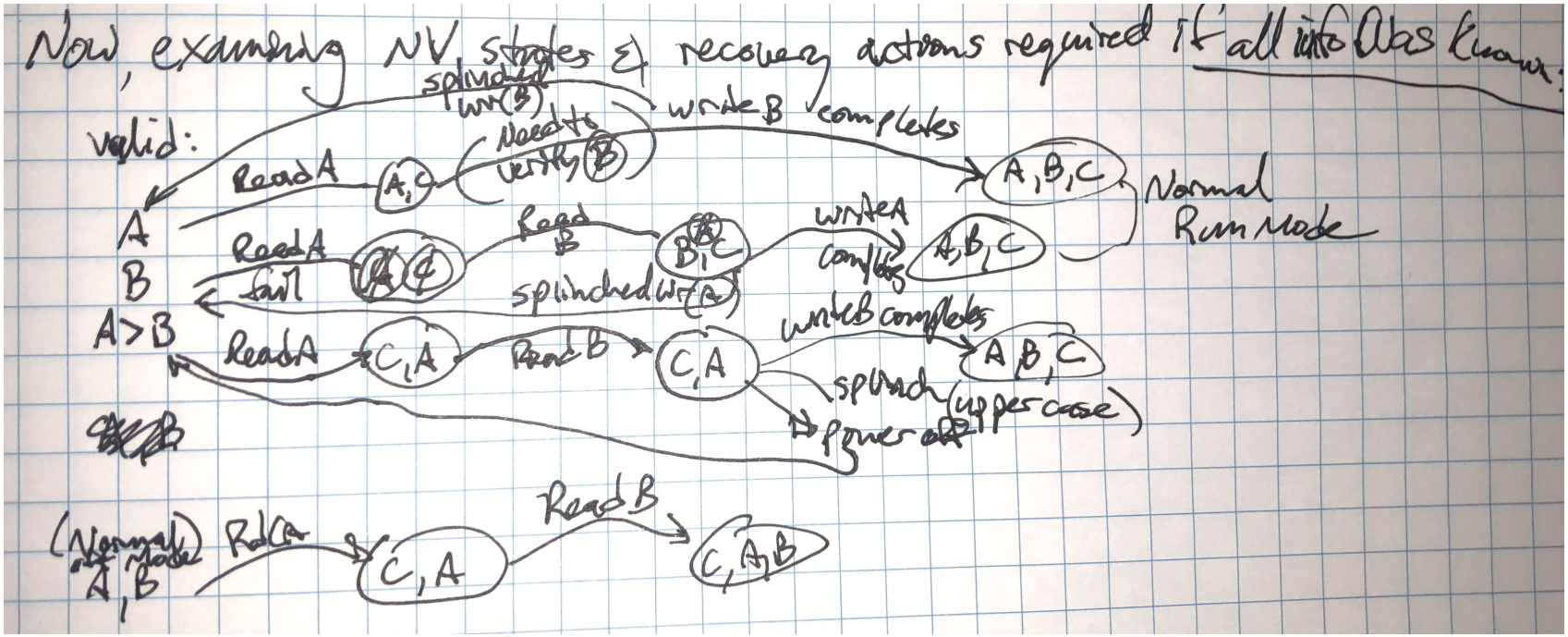
Now add the 3 outcomes for Wr(B) as well.

Level-setting: Reality



These things never just pop out in finished form. Even though I've solved this problem previously, I had a couple of false-starts with segmentation of the state space, and in breakdown of events.

Level-setting: Reality, cont.



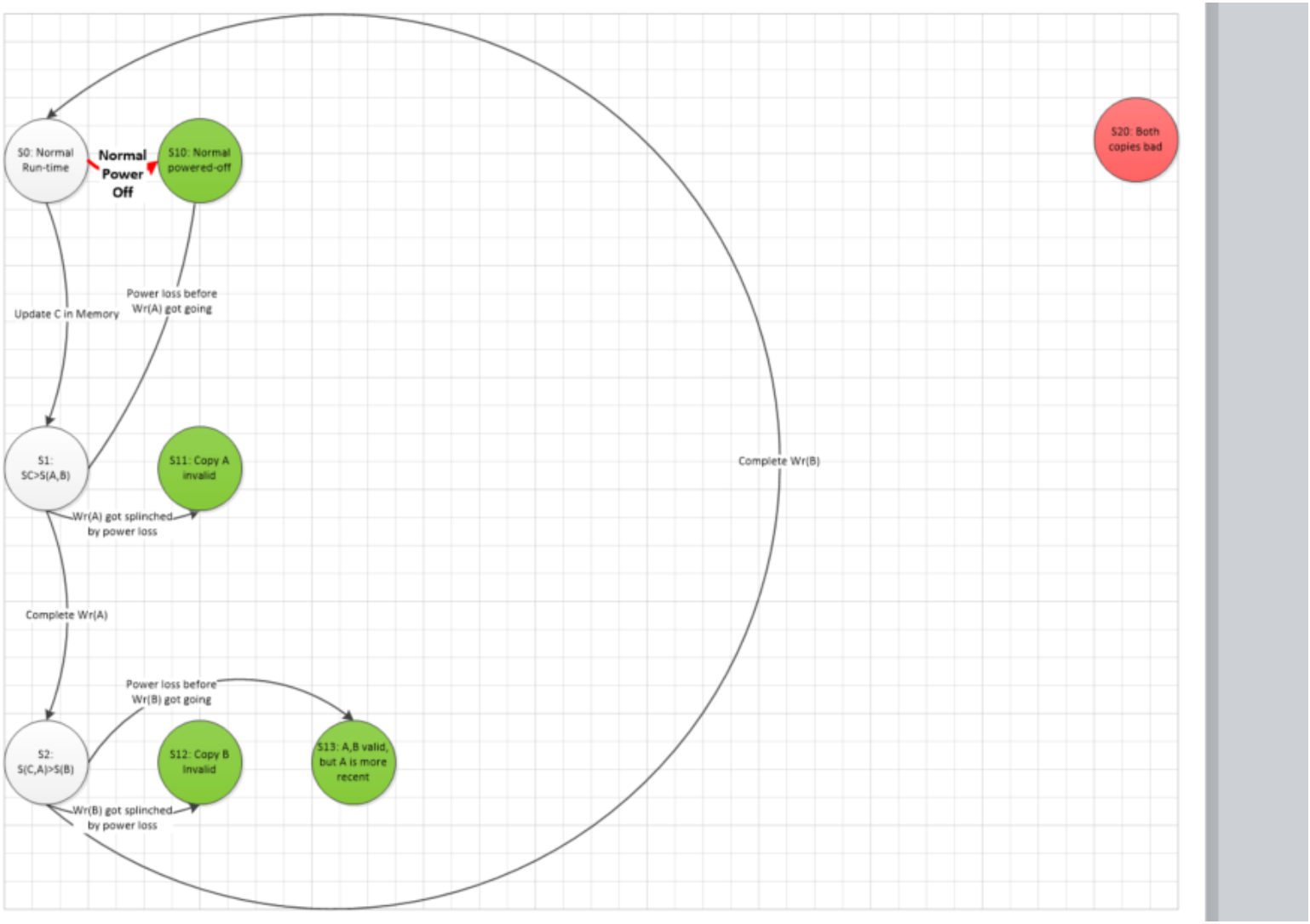
back to it. . .

We completed the state diagram for

- being already powered-up,
- no remanence issues (data decay)
- through all possible combinations of updates and a single power loss
- OR DID WE?

Can you spot the missing combination/case?

All “Run-time” cases, with Remanence



Next Step: Determine which NV case

- Read Both A, B
  - Note Success, SequenceNum of each read
- Use the (a) new/good one
- Write the old or defective one

In the manner we defined A as "first-to-update", the old one must be B and the new one is A.

Concurrent vs. Sequential Reads

Depending on media, A and B could be accessible at the same time, or only sequentially.

- One HDD: sequential
- Multiple NAND chips: possibly concurrent

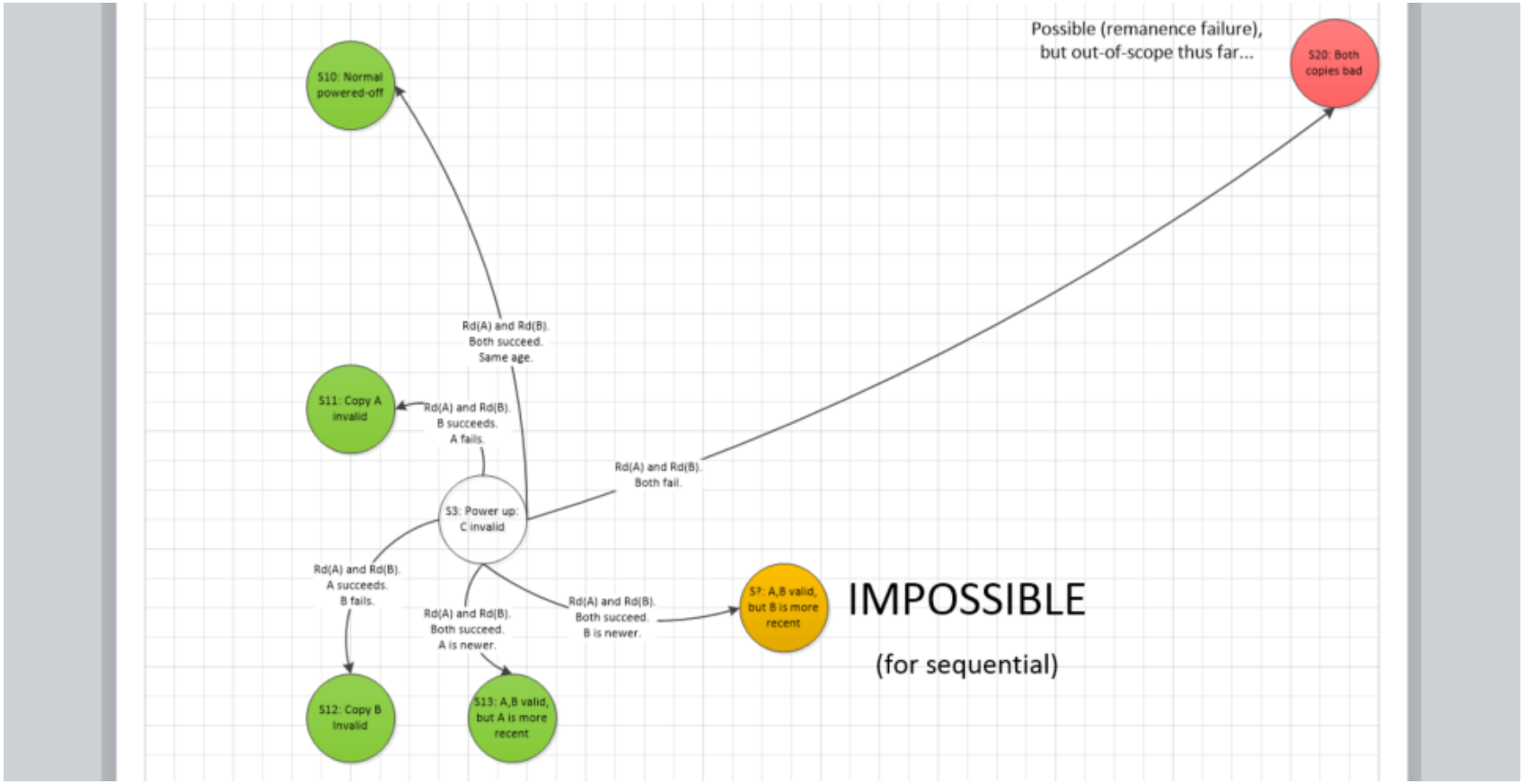
Support Concurrency?

- Is there a simple generalization we can make on the SD to allow concurrency?
  - Not simple, unless we change A/B to be first/last for writing, and older/newer on reading
- Is true concurrency safe, as the system is defined?
  - *What if concurrent writes are both splinched?*

Continuing with Sequential Assumption

In this SD, we can't know which NV state the data is in when we begin the power-on-recovery. So the first step is to simply go from "unknown NV state" to unambiguous state.

Power-On Discrimination



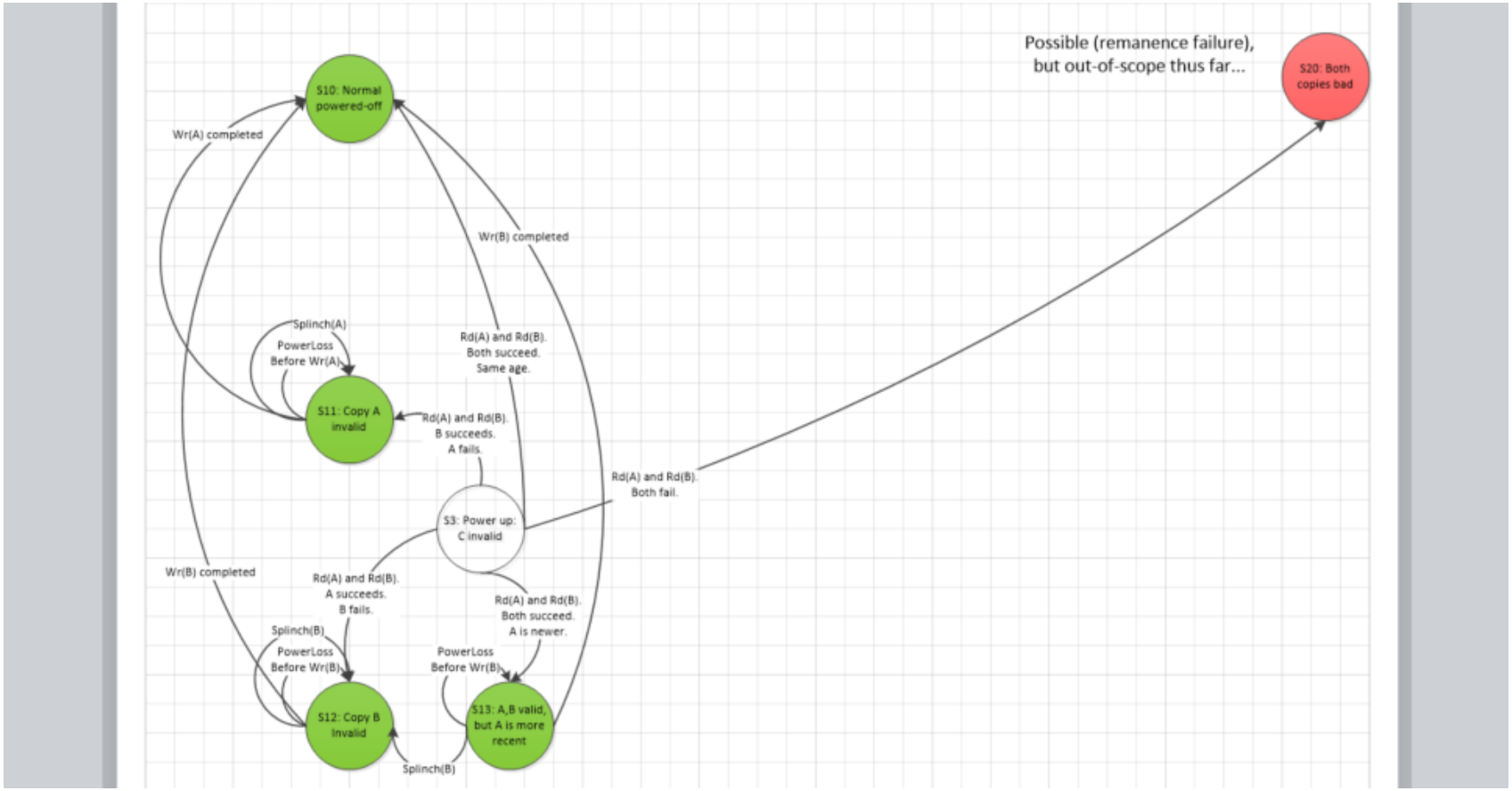
We'll circle back on the non-remanent issues in a bit.

Back to the Bigger Picture

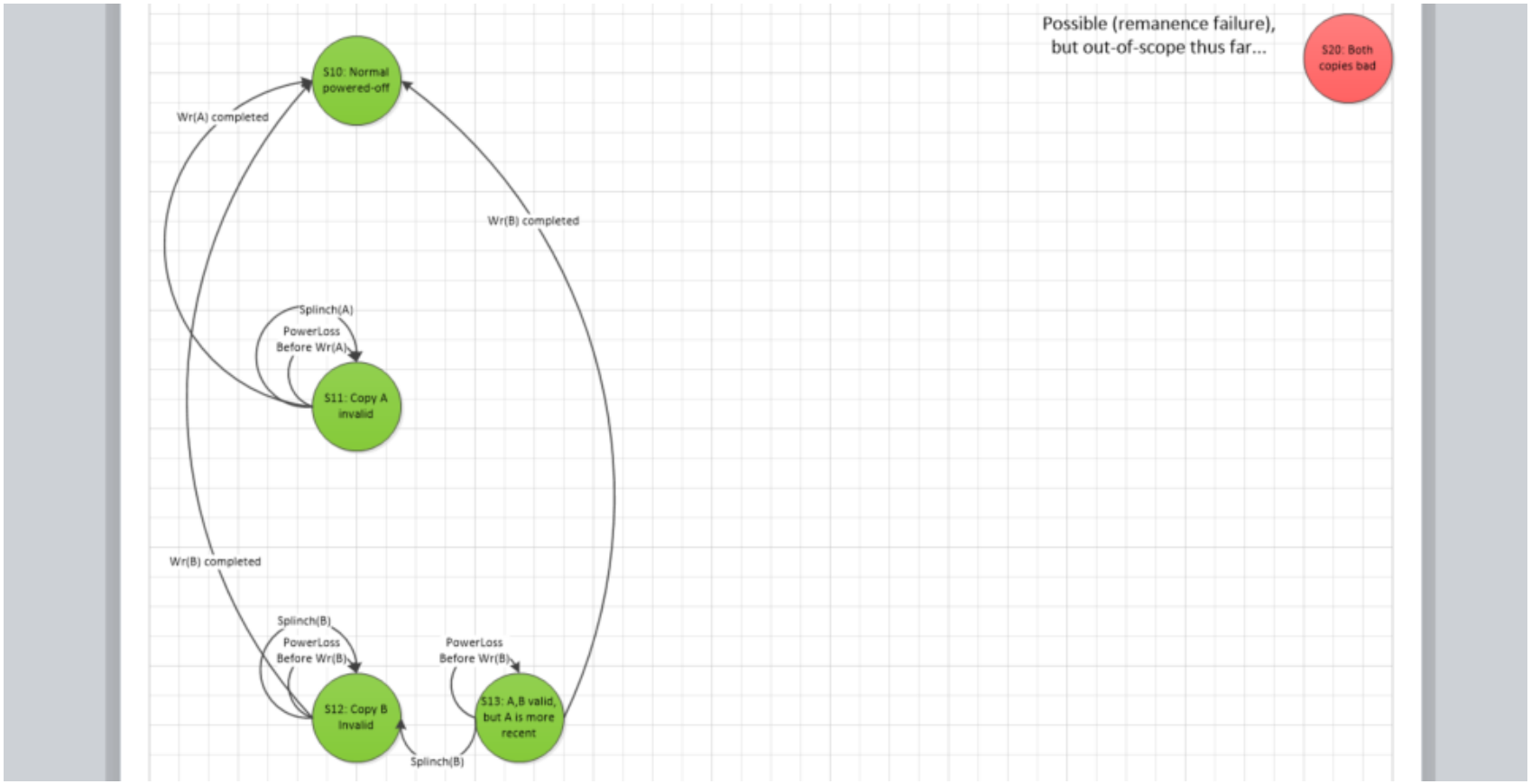
The state diagram that represents the NV states (which our original SD exceeded, by showing the normal volatile updates) doesn't need to show the Power-On Discrimination.

It does need to show the events that lead to different NV states.

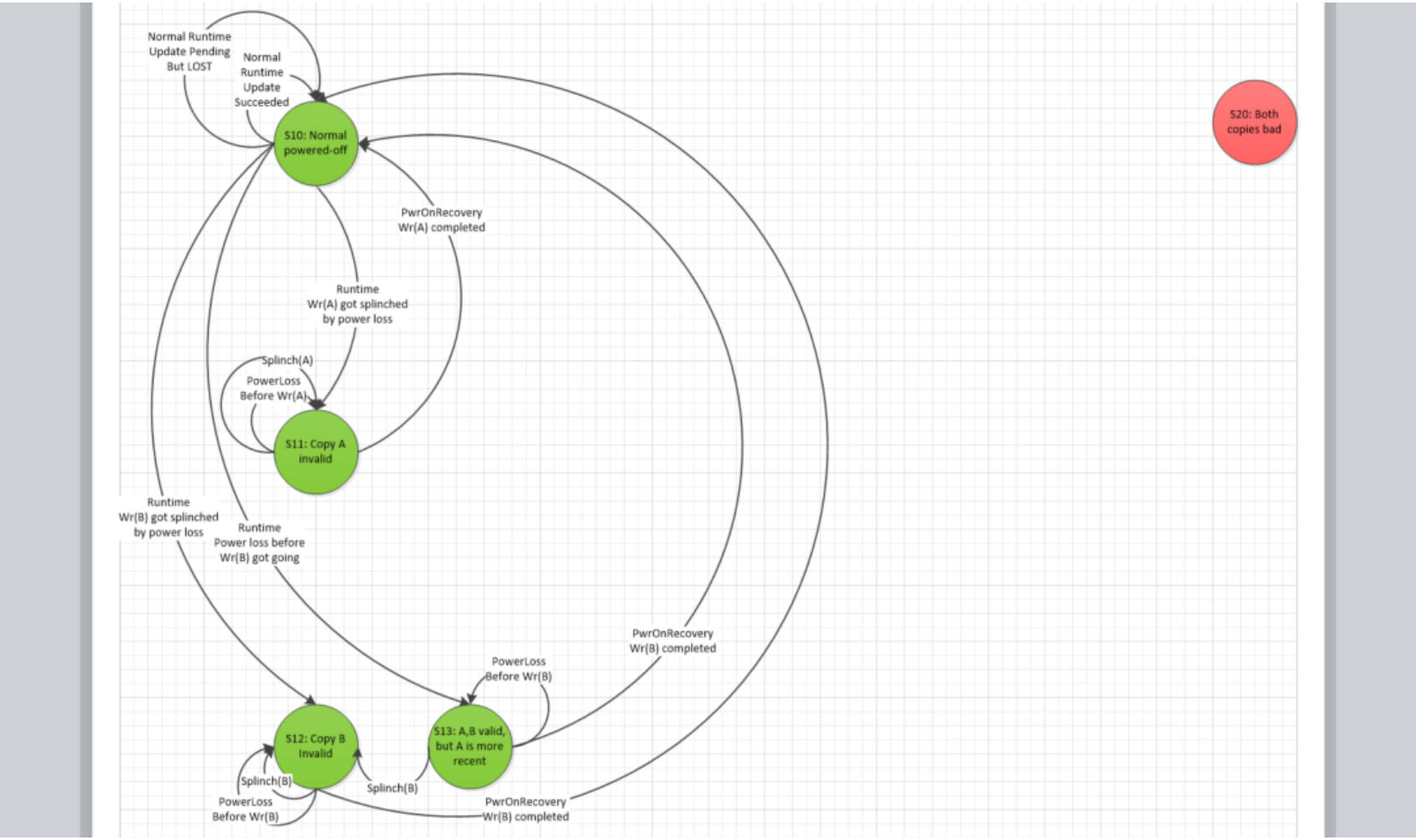
Overly-Busy Discrimination and Recovery



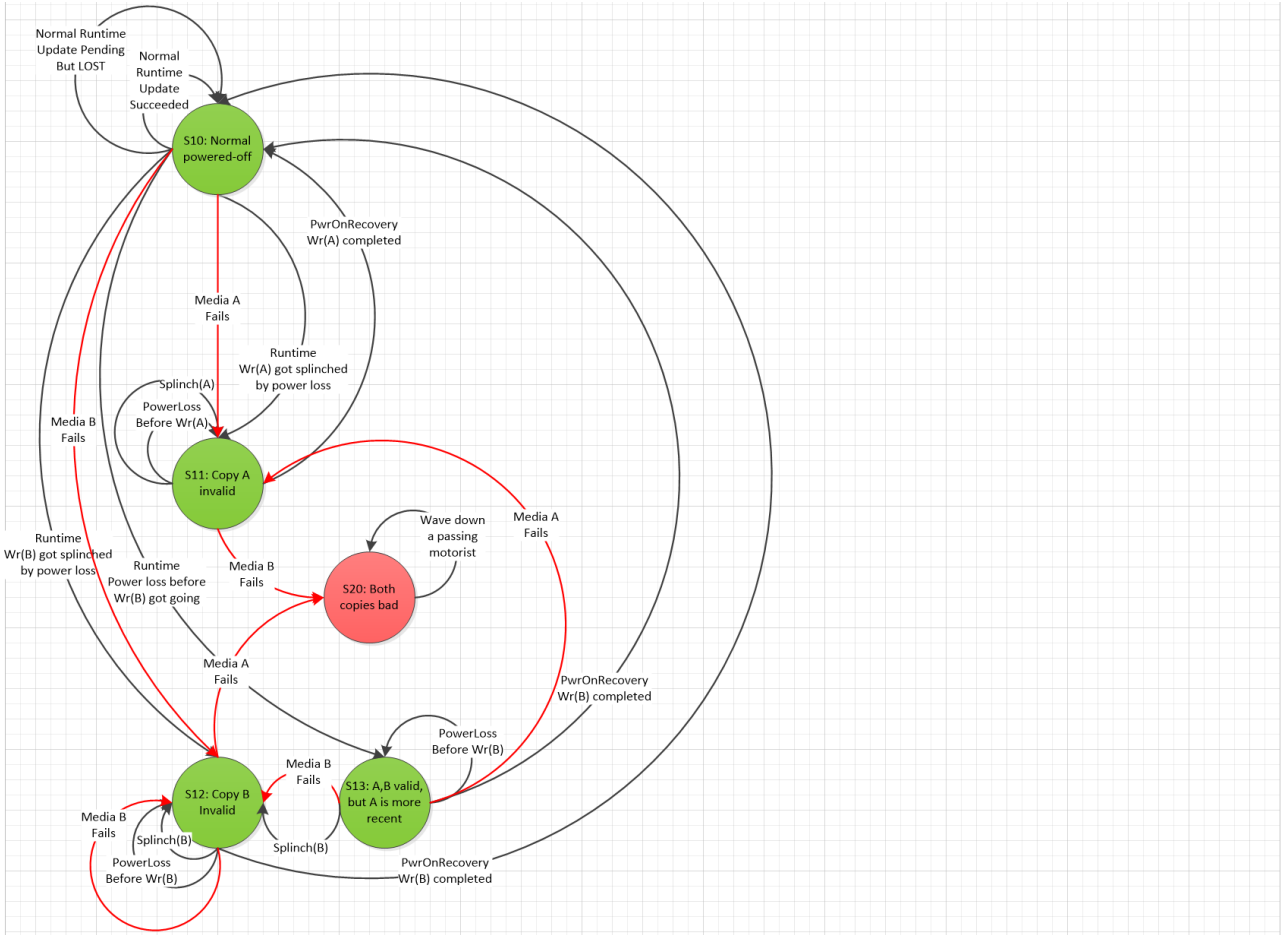
Power-On Recovery (NV State changes, stable media)



RunTime and Recovery (NV Only, Media Perfect)



Now, Add Single Media Failures



Unit testing of this system

Read:

- https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html

and reflect on how you could design unit tests from this state diagram.

There are parts of unit testing here that could be written to use Dummy/TestDouble, Stub, Spy, Mock, and Fake constructs. But Mocks and Fakes take a lot more effort and are more complicated, so we more likely would build several test cases from the state diagram, and then unit test the responsible functions with Dummies, Stubs, and Spies to ensure that the simple internal decisions are implemented correctly in code.

When you do the Scheduling labs in this course, you'll need to generate multiple specific tests to ensure that your scheduling logic is operating as expected.