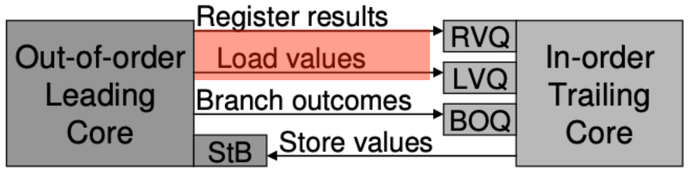
Two things on my mind.

- One pretty mature.
- One 2% baked.

First thing: "Leveraging 3D Technology for Improved Reliability" (MICRO 2007)

- 3D die stack an inorder checker on top of an OOO leading core
- Checker trusts all load values
  - D$, LV, and anything that carries load values protected by ECC
- Stores on leading core have to wait for checker



Trailing checker core:
- executes every instruction
- does not access data cache
- has perfect cache and br-pred
- can also employ a perfect register value predictor
- can support high ILP even with an in-order pipeline
- can execute at a lower frequency than the leading core

Summary of results by Madan et al. :
- In-order trailing core executes at a frequency that is only 45% of leading core's frequency
- No performance loss for the leading core
- Inter-core interconnects typically consume less than 2 W
- Redundant multi-threading can impose a power overhead of less than 10%

Fault model:
- a single error will be detected
- a single error can be recovered from if the trailing core's register file and datapaths are protected

Load values are also passed to the trail- ing core so it can avoid reading values from memory that may have been recently updated by other devices. Thus, the trailing core never accesses the L1 data cache. The lead- ing core is allowed to commit instructions before checking. The leading core commits stores to a store buffer (StB) in- stead of to memory. The trailing core commits instructions only after checking for errors. This ensures that the trail- ing core's state can be used for a recovery operation if an error occurs. The trailing core communicates its store val- ues to the leading core's StB and the StB commits stores to memory after checking.

Figure 1. Overview of checker core design proposed in [19].

rs such as resources not being freed. The following The checker core is a full-fledged core wit

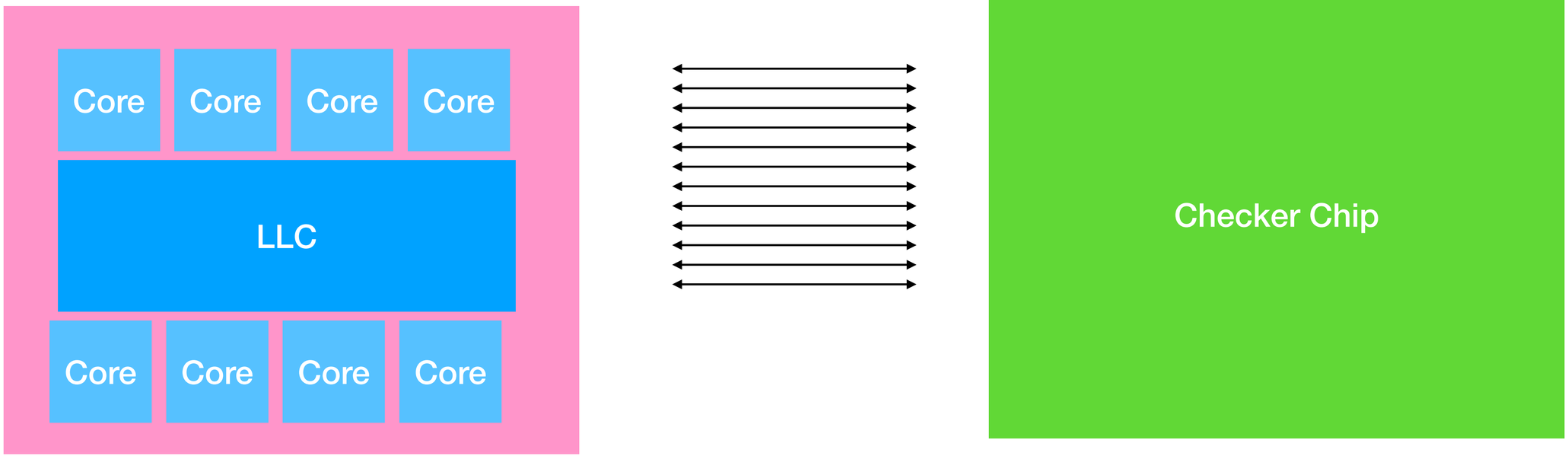
This work is thinking about reliability only

- While I find the "trust loads decision questionable for that use case..
- If we can get rid of that constraint, then we open up a lot of other use cases
  - o Security (fabrication attacks on leading core)
  - o Problems that arise in cache coherence, etc
  - o **Possibly more aggressive architectural decisions**
    - "This would be great but verification engineers will never go for it"
    - Cut voltage/frequency guard bands

      .....

High level gist of idea

3d die-stacked

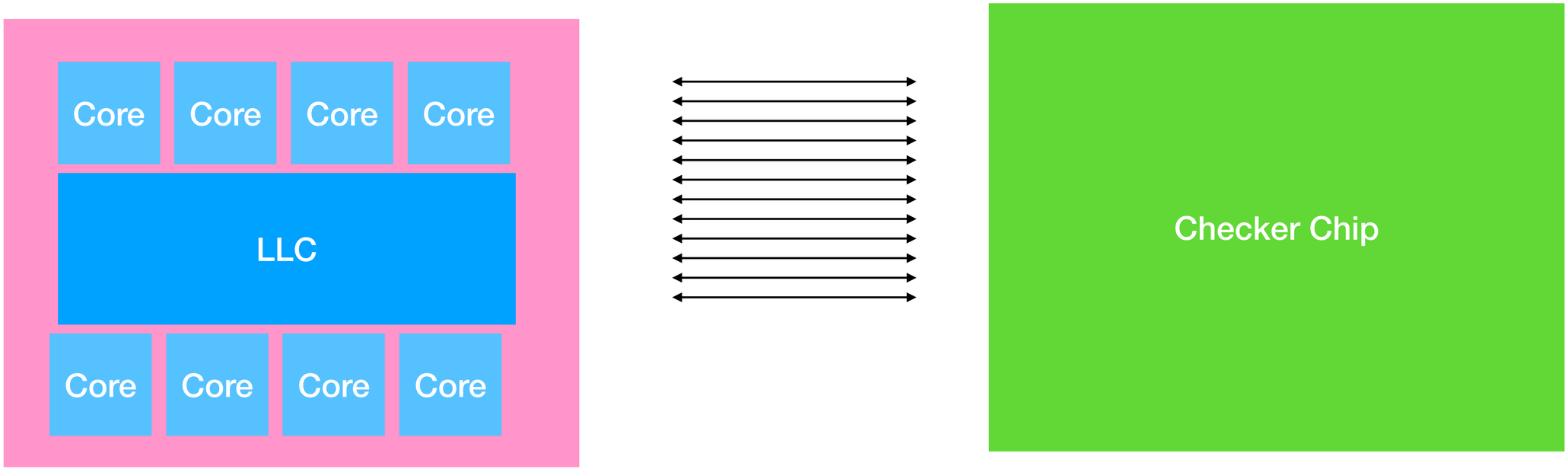


Very Aggressive OOO multi-core
- Focus on performance
- May include ultra-rare design bugs which could affect functional correctness
  (manifest rarely enough to not be performance problem)
- Fast clock, aggressive technology

Specially designed checker chip
- Intended to verify multi-core as well as single-core properties
  (e.g. cache coherence, memory consistency)
- Slow clock (lets say 1/4 of OOO)
- Incredibly high ILP
  - Values from OOO not trusted, but used as predictions
- Very unusual pipeline design

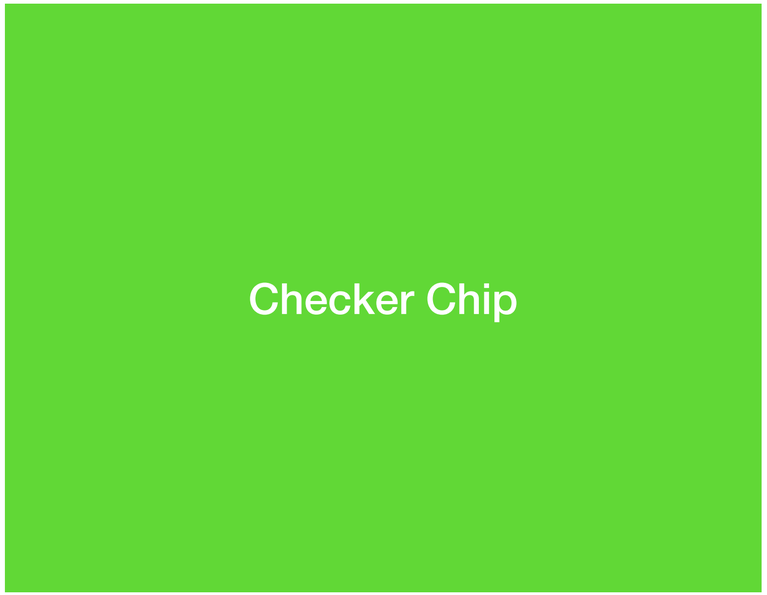
High level gist of idea

3d die-stacked



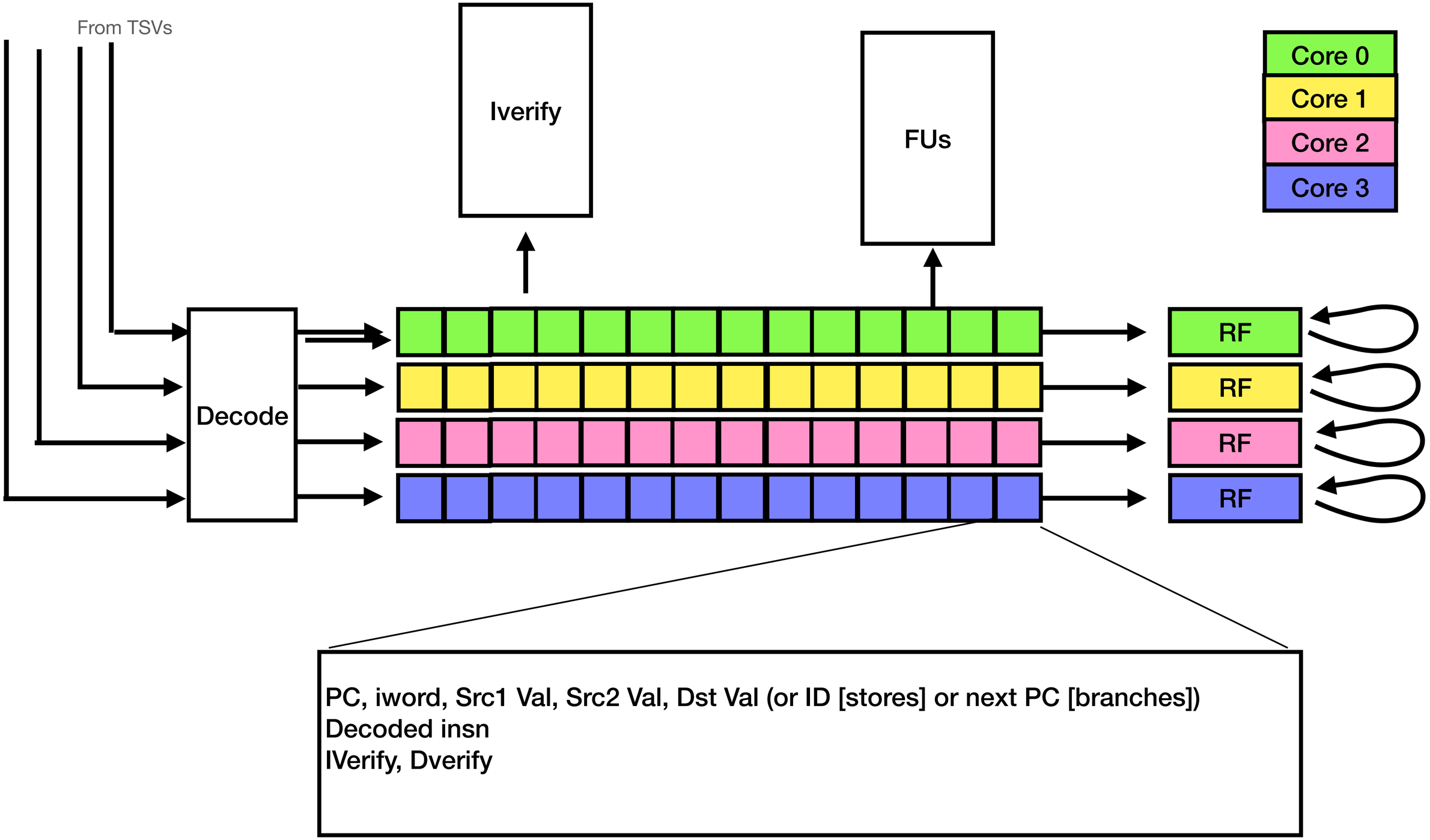
OOO Core sends 2 types of events
- Instruction commit
    - PC, input operands, output value
    - For loads: input operand, physical address, output value
    - Stores get an ID (storebuffer index)
- Store completion (write to D$)
    - I'll talk about this as x86, but adaptable to other consistency models
    - Just store buffer index is sufficient

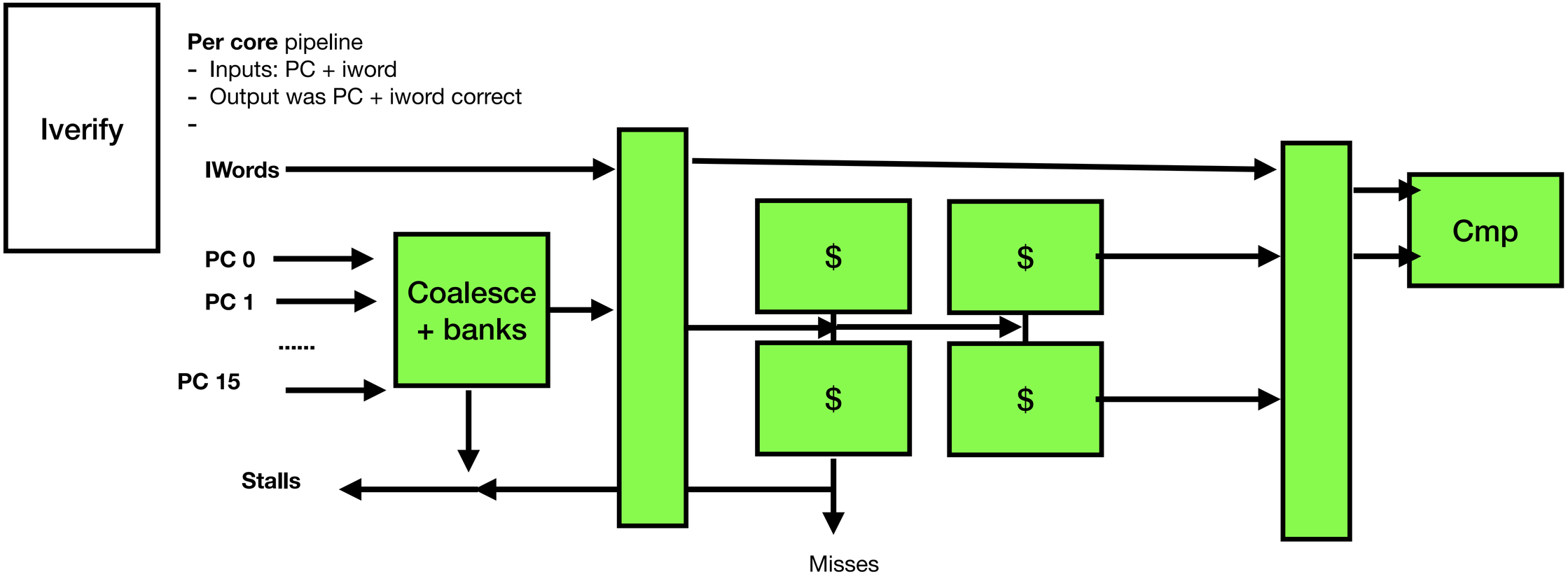
All cores have same latency from their commit to where the checker chip can timestamp them.
- Difficulty: either need to go 4x width across TSVs or have small region of checker chip that is at OOO clock to receive incoming
- 8 cores * 4 IPC * 256 bits/insn = 8192 TSV
    - I have no idea if this is reasonable, and can't find numbers easily?



What does this look like?

Design constraints (and non-constraints)
- Simple design: ensure correctness
- Can't trust anything from OOO
    - Can use as predictions (which should always be right)
- Latency does not matter much. Only matters at...
    - Errors (reset leading cores)
    - IO (has to be done via this chip at right time)
- Bandwidth matters: can't backlog to where OOO needs to stall
    - Can buffer against short bursts
- Needs to support an IPC of 8 (cores) * 4 (IPC of OOO Core) * 4 (clock ratio) = 128
    - Might be able to reduce that some: probably not 4 IPC on all 8 cores at once (~100?)
- Needs to verify coherence/consistency operations
    - Want to avoid any nasty coherence/consistency protocols


From TSVs
Iverify
FUs
Core 0
Core 1
Core 2
Core 3
Decode
RF
RF
RF
RF
PC, iword, Src1 Val, Src2 Val, Dst Val (or ID [stores] or next PC [branches])
Decoded insn
IVerify, Dverify



Frequently PC0, PC1, PC2 in same block. We coalesce those into one request.
May need to coalesce PCX = PCY (short loops)

Bank conflicts? Later PCs stall

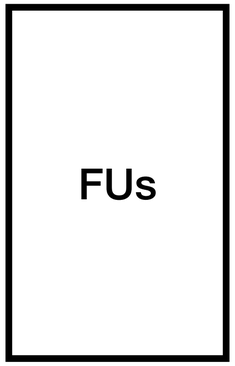
This logic can be across multiple cycles---no problem

Not pictured: verify PC1 = PC0 + 4 (unless branch instruction, in which case verify is nextPC)

Misses stall IVerify for that core

Can easily prefetch next addresses in queue under the miss

Could even verify other instructions under the miss
(just a matter of setting bits in queue---no ordering constraints/complexities

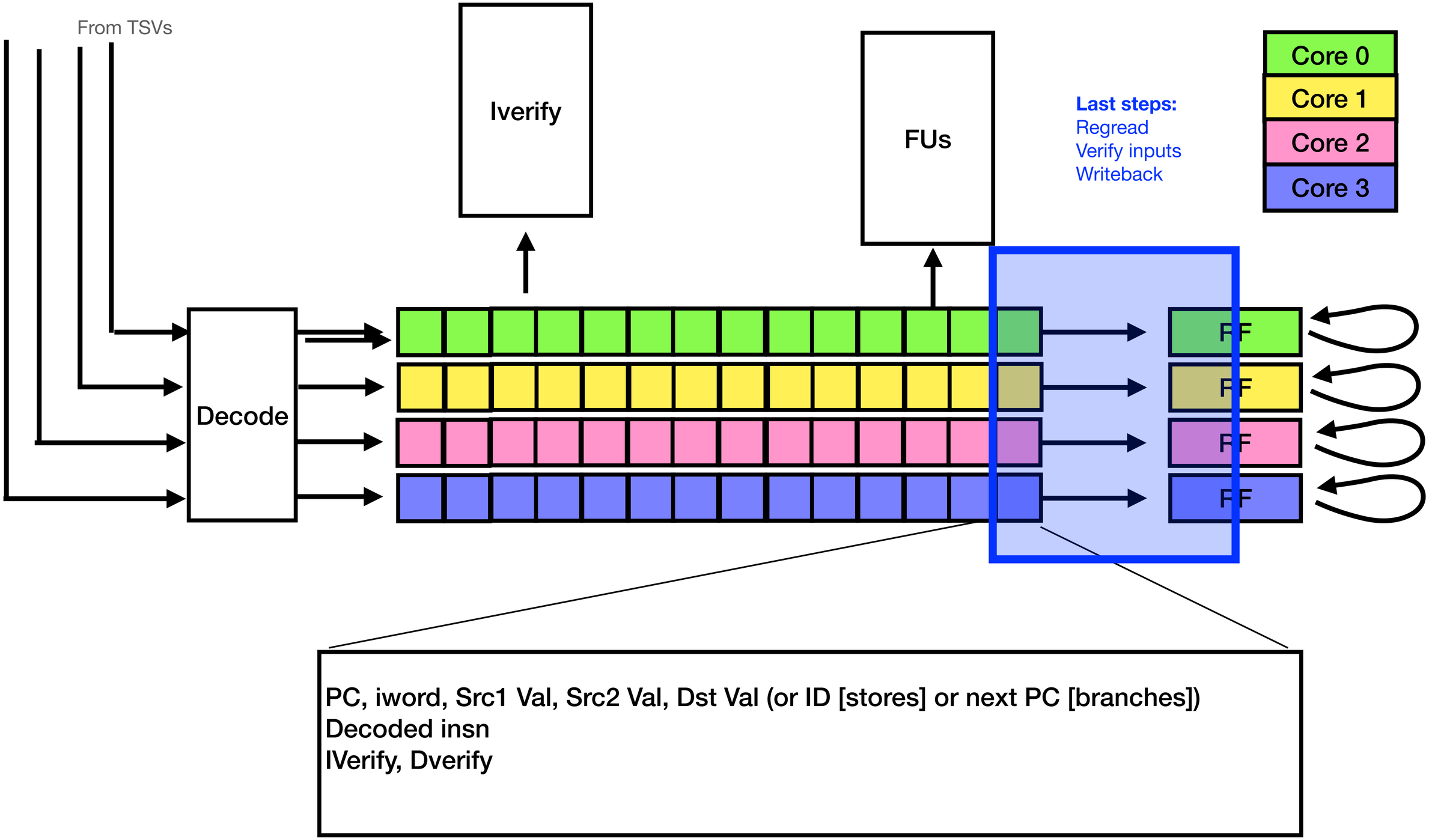


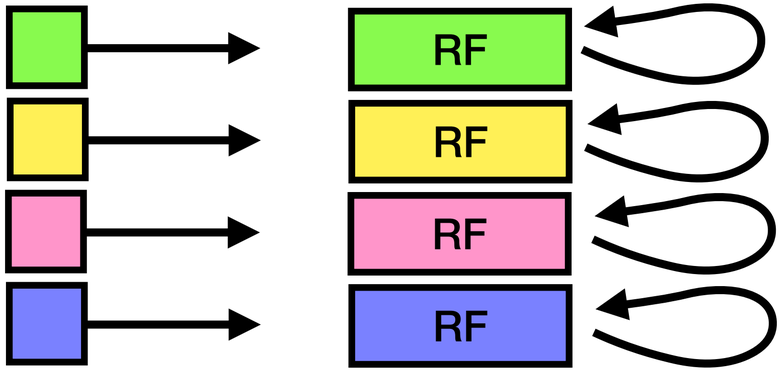
Execution pipeline

- Could be per core (simpler)
- Could be shared between cores (reduce number of complex FUs needed)
- Middle ground: per core Integer ALUs, shared FPUs
    - Per core queue for FP instructions, small "scheduler" to pick from queues based on available FP FUs

Can do dependent instructions in parallel

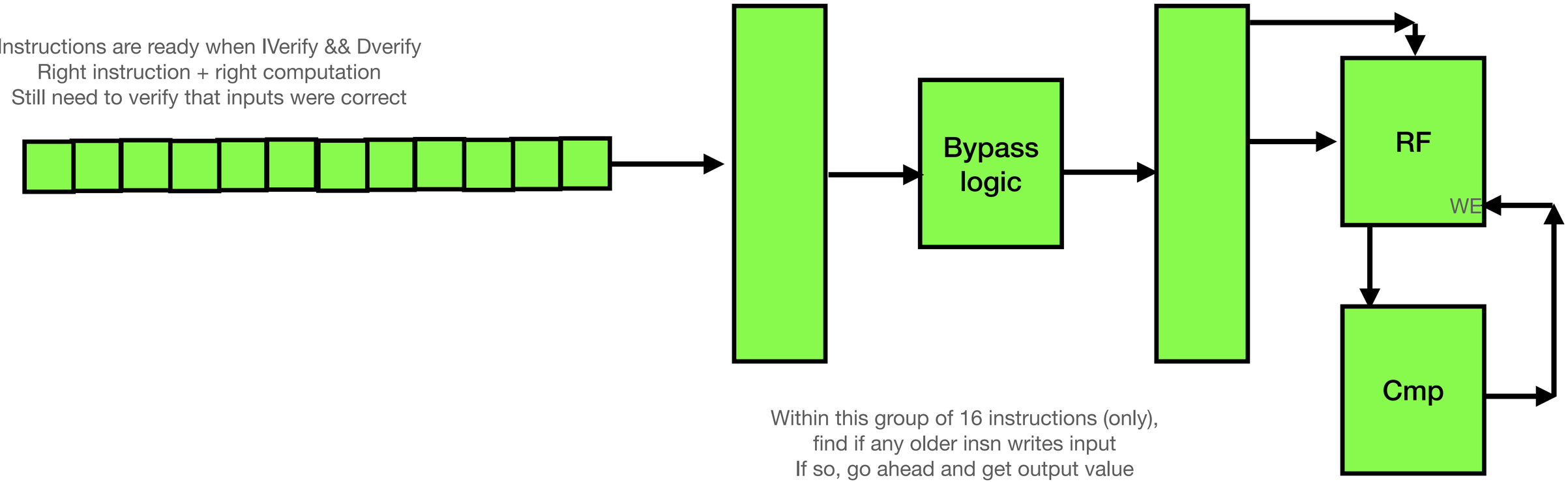
- Uses source values sent by OOO Core
- Compare result to output value sent by OOO Core
- Set DVerify bit in queue entry


From TSVs
Iverify
FUs
Last steps:
Regread
Verify inputs
Writeback
Core 0
Core 1
Core 2
Core 3
Decode
RF
RF
RF
RF
PC, iword, Src1 Val, Src2 Val, Dst Val (or ID [stores] or next PC [branches])
Decoded insn
IVerify, Dverify



Per core register files
But each needs to support 16 insn/cycle
- Even if we assume only ~10 reg writing instructions on average, this is a lot of write ports.
- We'll come back to this in a moment. Lets see how it works first

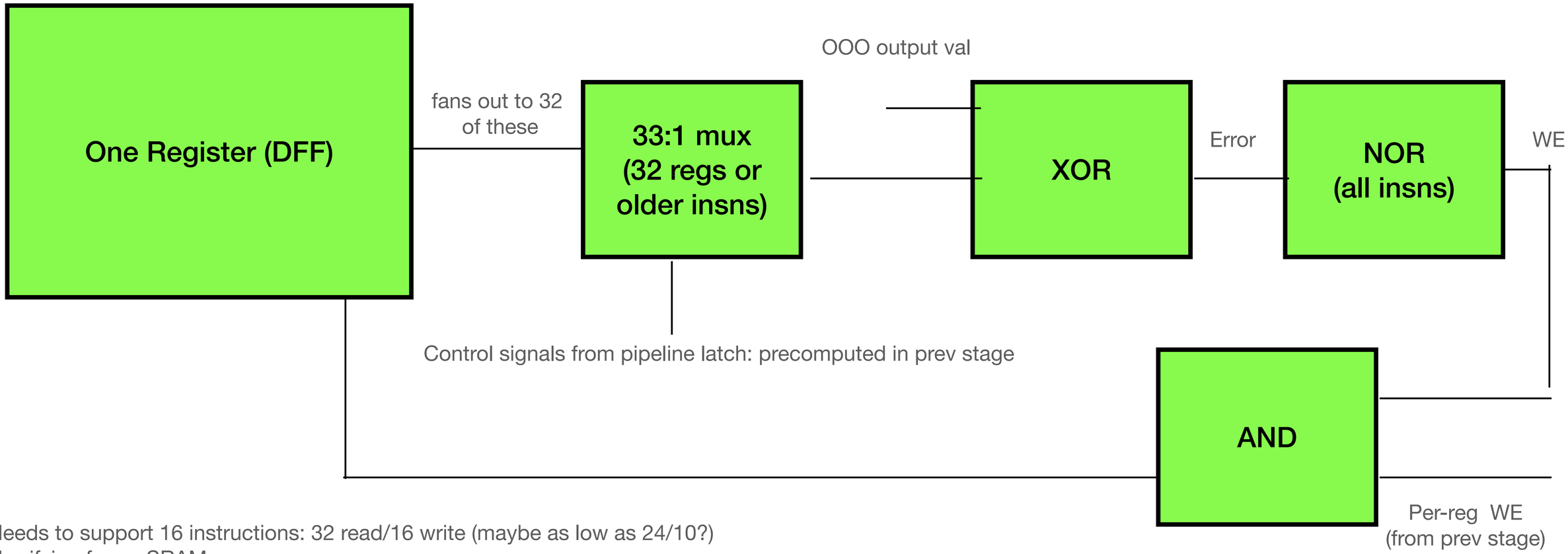
Instructions are ready when IVerify && Dverify
Right instruction + right computation
Still need to verify that inputs were correct



Within this group of 16 instructions (only),
find if any older insn writes input
If so, go ahead and get output value
(Each src has either value or reg to read)
OK if this takes 2 or 3 cycles

Read
Verify
Write
MUST(*) be single cycle

(*) If not, then we still have to do read/write in a single cycle (otherwise bypassing to next group = complexity blowup)
We then need another register file written a cycle later for error recovery (only used if error)

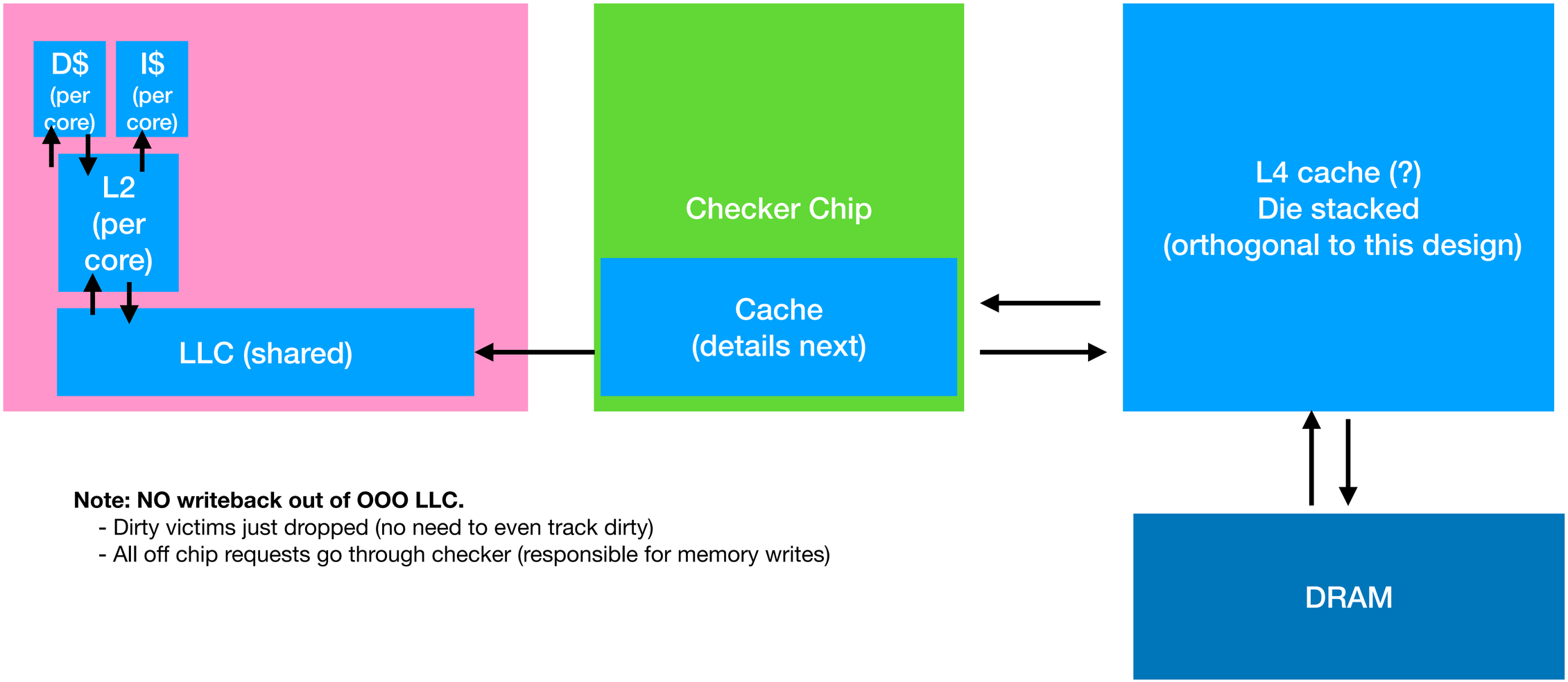


Needs to support 16 instructions: 32 read/16 write (maybe as low as 24/10?)
Horrifying for an SRAM
But we could just this out of DFFs.
- Fan out of 32 is probably bad... But we could make one replicate if needed
- Write mux control signals (16->1) and write enable computed in bypass logic stage

This is probably the part of this plan that will cause the most problems with logic design feasibility

Let's talk Loads and stores (these are the most interesting part)

-First the memory hierarchy



Note: NO writeback out of OOO LLC.

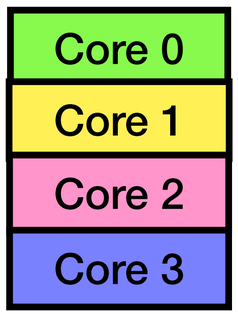
- Dirty victims just dropped (no need to even track dirty)
- All off chip requests go through checker (responsible for memory writes)

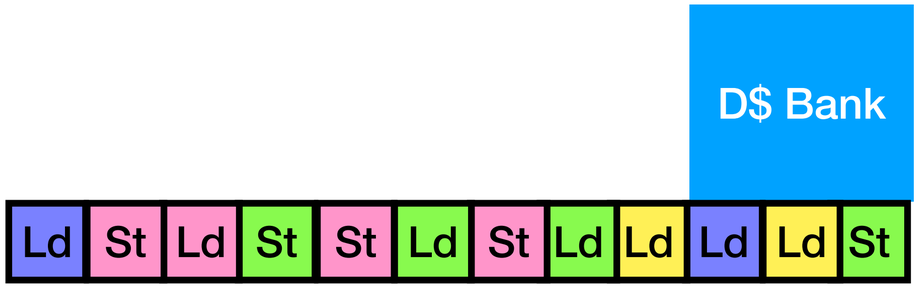
Let's talk Loads and stores (these are the most interesting part)

- Let us guess we need to support roughly 48 loads/cycle (8 * 4 * 1.5)
- And about 16 stores/cycle (8 * 4 * 0.5)
= 64 memory ops/cycle

All physical addresses known apriori
-> Build **128 banks**, divided by physical address
-> Lets say 32KB/bank (each bank looks like a normal D$, except 7 bits for bank ID)
-> Note the "L1" is 4MB. No L2 in this design.

Each bank has a queue of operations for that bank.
- Loads + store completions to that addresses in that bank
- Combines operations from all cores in the order they were received

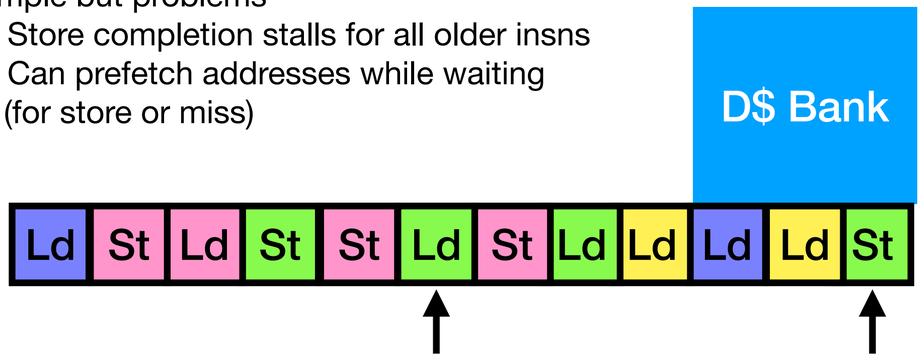




1/cycle: store completion OR load verification

A few design details add a bit of complexity.

Simple but problems
- Store completion stalls for all older insns
- Can prefetch addresses while waiting (for store or miss)

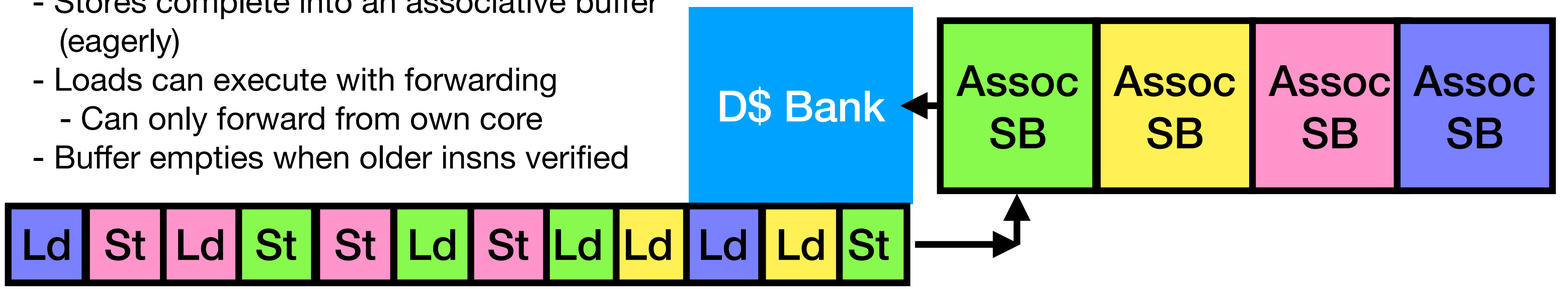


Store commit -> load verify loop. Likely with significant latency

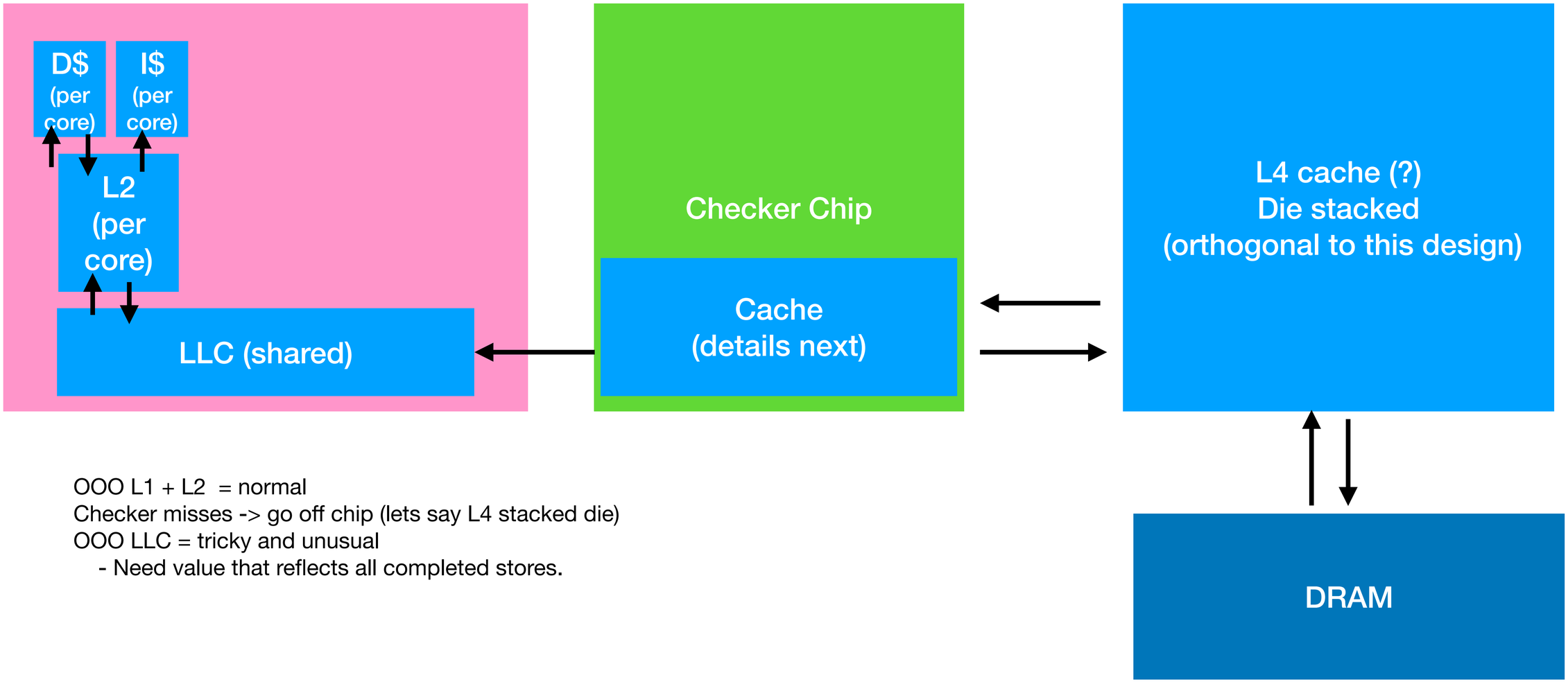
Queues -> 128 banks -> Queues

Loads can get verified while older stores uncommitted
Recovery done by clearing SB entries
Actually slightly more complex to handle OOO LLC Misses

Choice 2
- Stores complete into an associative buffer (eagerly)
- Loads can execute with forwarding
  - Can only forward from own core
- Buffer empties when older insns verified



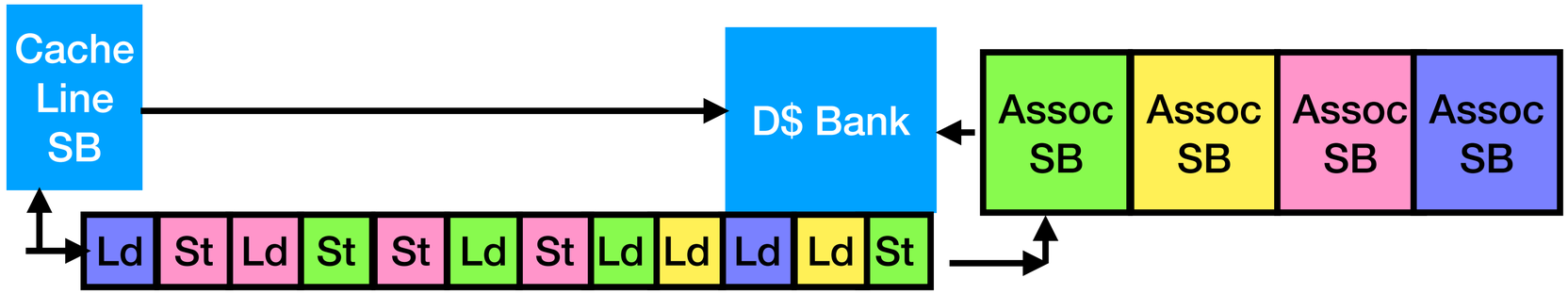
How do we handle cache misses?



OOO L1 + L2 = normal
Checker misses -> go off chip (lets say L4 stacked die)
OOO LLC = tricky and unusual
- Need value that reflects all completed stores.

For handling OOO LLC Misses

For forwarding during verification

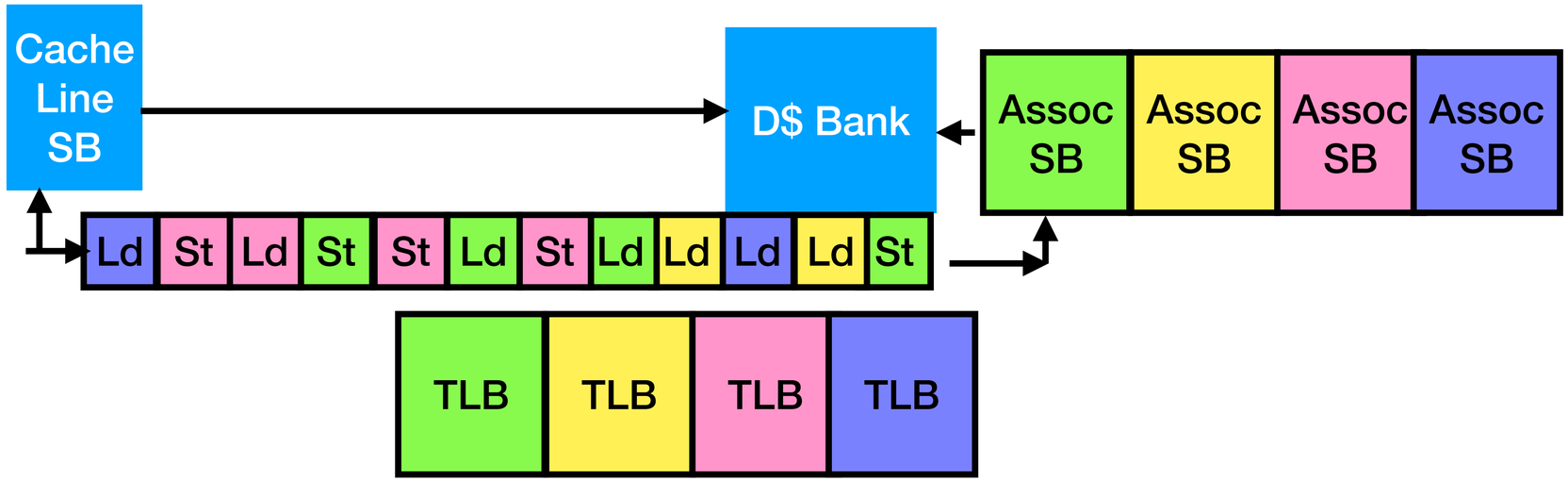


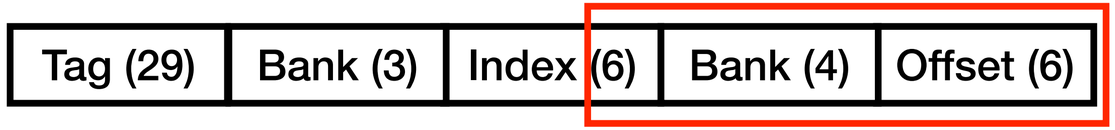
Second Store buffer organized at cache line granularity
  Valid bits per byte
  Stores write this SB as soon as they enter their bank's pipeline

OOO LLC misses:
  Go to Checker $ first
    - If miss, Checker $ should fill (assume it will need it)
    - If hit (or after fill), "fixup" block with store buffer values as needed
  Send corrected value up to OOO Core
    - Maybe a race here: what if stores complete right before or right after SB lookup?
    - May be impossible if LLC inclusive and cores require hit to complete
    - If possible, but super rare:
      - Ok (but not great) to get it wrong: Checker still right
    - If possible but common (e.g., dbz on PPC)
      - Cores need to send recent completions to where LLC fill path comes in.
        These have a short duration of need.

For handling OOO LLC Misses

For forwarding during verification

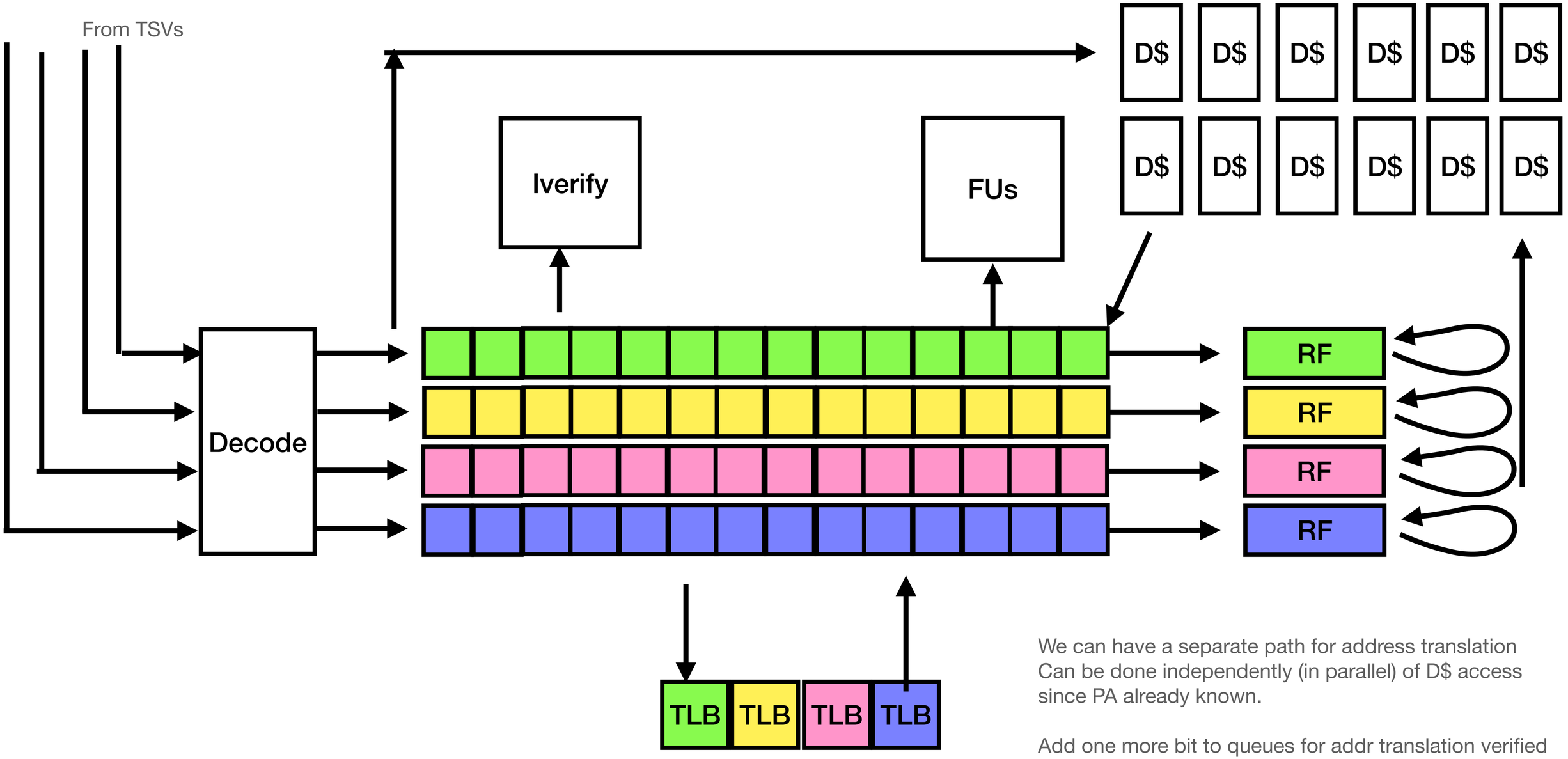




One option: small TLB in each bank
- This is probably wasteful: offset bits contain bank number
- 256 entry TLB? 2 entires per bank = not much flexibility

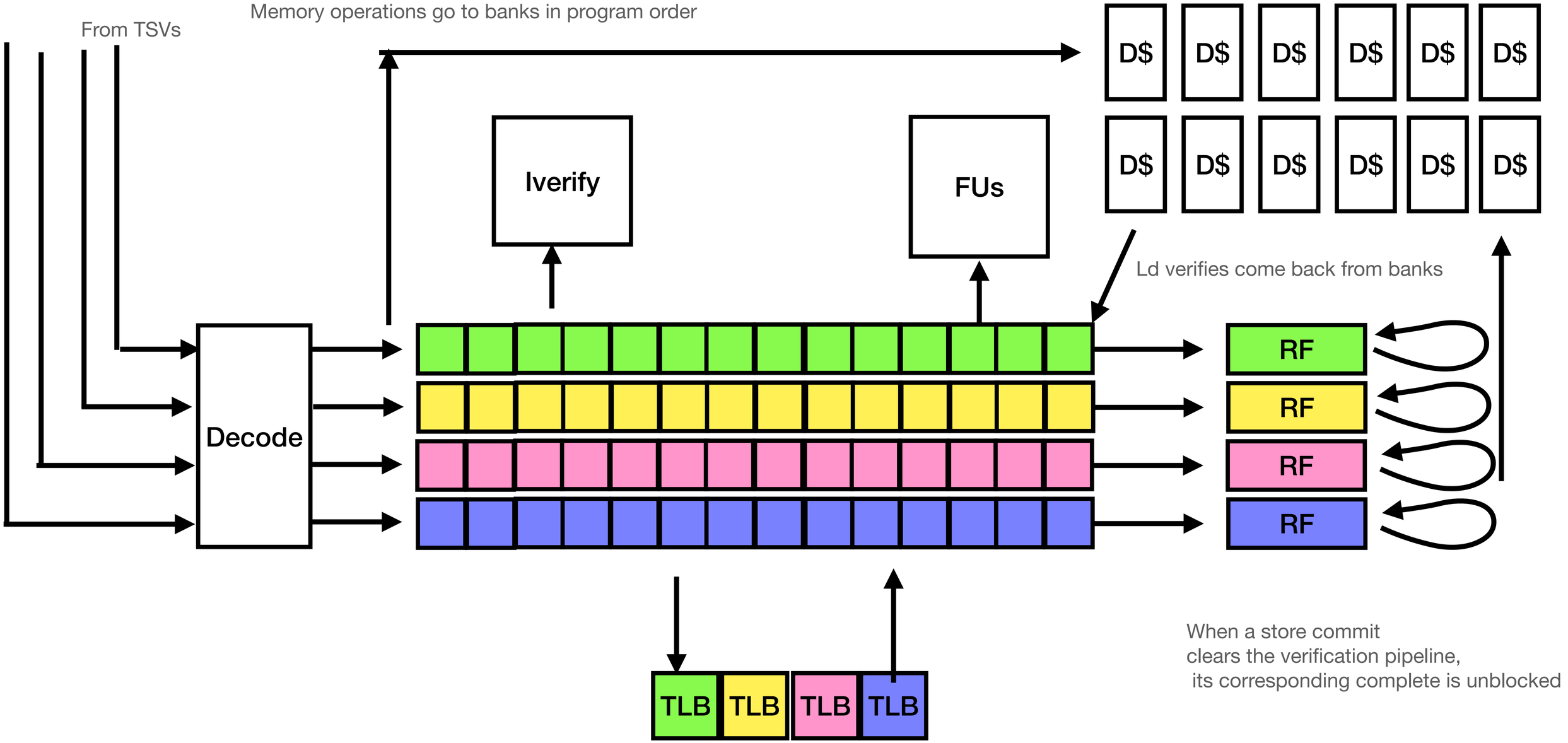
16 banks will contain same translation

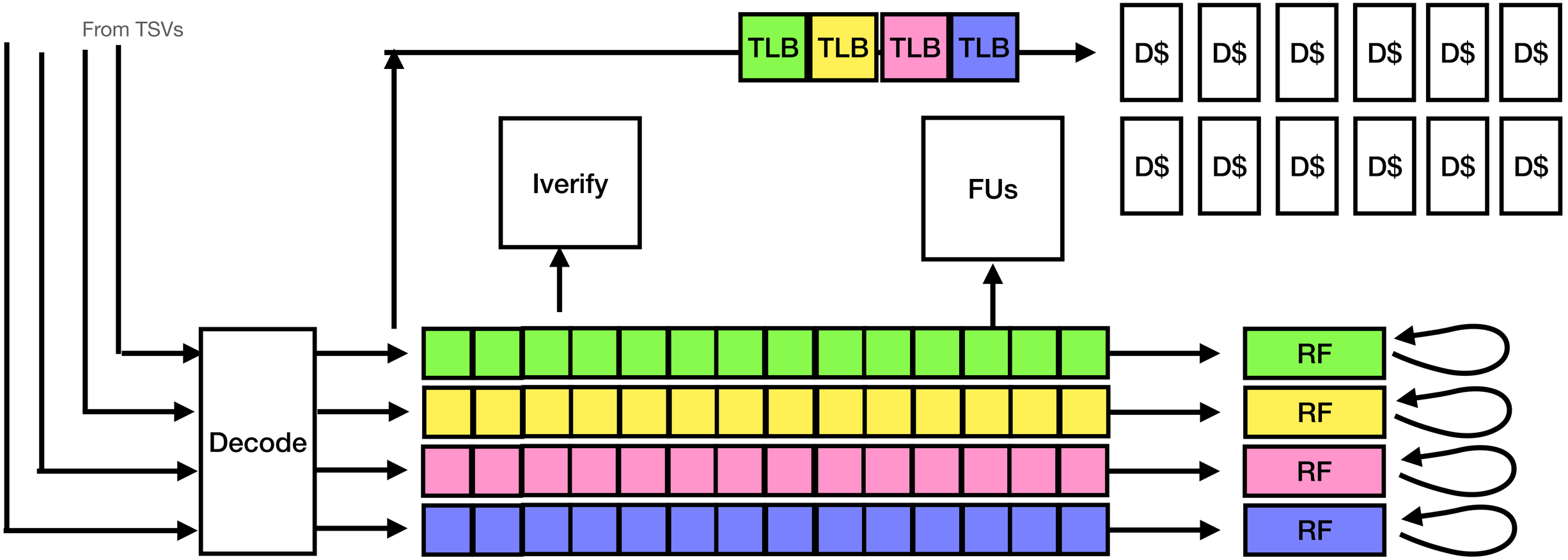
I realized after I sent it that this is wrong: banking is by Physical Address and translation needs to be done by Virtual Address.



We can have a separate path for address translation
Can be done independently (in parallel) of D$ access since PA already known.

Add one more bit to queues for addr translation verified


From TSVs
Memory operations go to banks in program order
D$
D$
D$
D$
D$
D$
D$
D$
D$
D$
D$
D$
Iverify
FUs
Ld verifies come back from banks
Decode
RF
RF
RF
RF
TLB
TLB
TLB
TLB
When a store commit
clears the verification pipeline,
its corresponding complete is unblocked



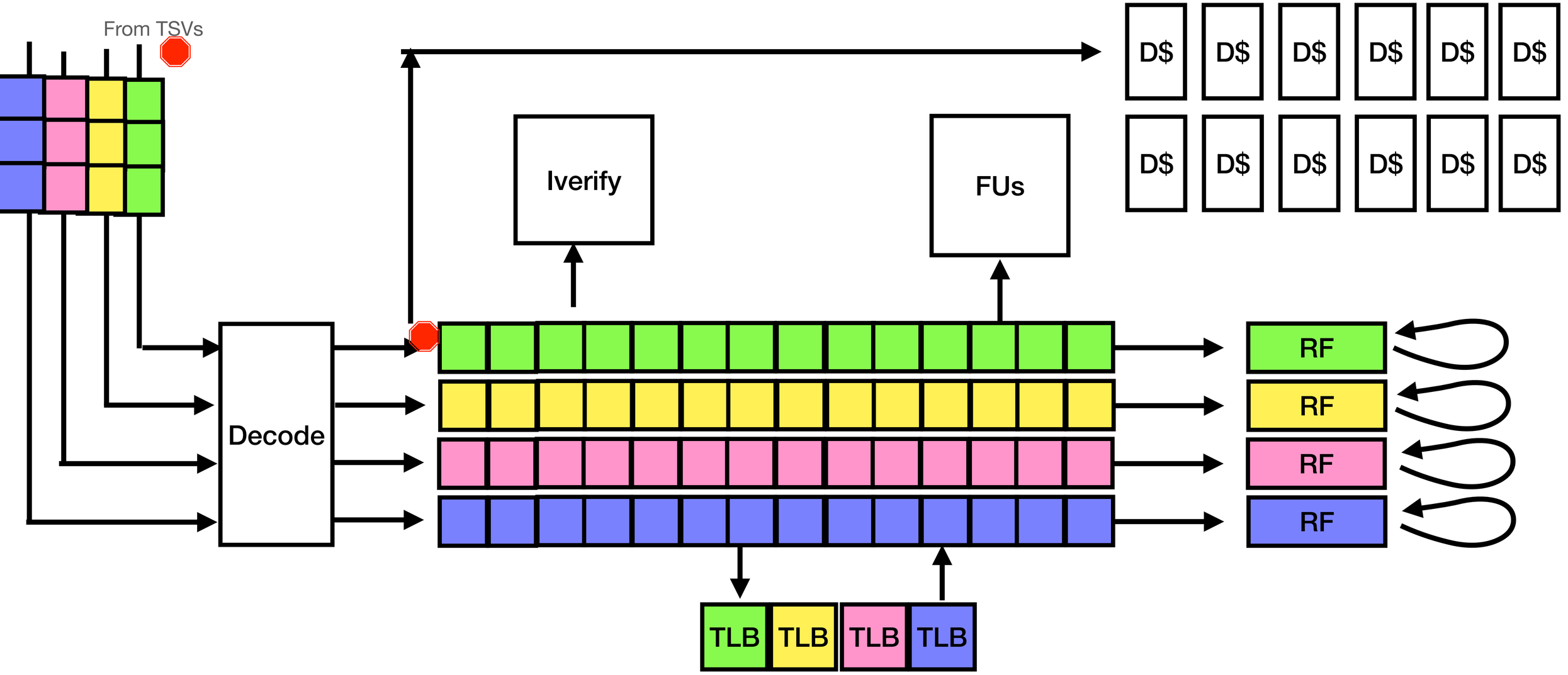
Lets make an assumption that for loads it takes us
- 4 cycles to get to our correct bank,
- ~~2 cycles for address translation,~~
- 3 cycles for D$ access + verify,
- 3 cycles to get back

Total time on a hit: ~~12~~ 10 cycles => We want to provision our main queues above for about **256** instructions/core

=> Iverify and FU pipelines can easily take ~~13~~ 10 cycles with no problem

=> We can spend a few cycles getting to things, maybe use slower (simple adders) logic than normal, a few cycles getting back

May want more to handled misses
- Need experiments
- Misses mean non-recent data in OOO LLC.
- Latency should be not so bad with Die stacked L4



Flow control done with tokens = point where stalls can happen

- Some small queueing at the front token loop to OOO core.
- Main queue entry point can just have standard "pipeline stall" logic back through decode to TSV entry queues
- ~~Each D$ bank group has a set of tokens for the buffering at the TLBs (2 + 2 + 2 cycle token loop, 16 entry queue per core is sufficient in TLBs)~~
- Each D$ bank has a set of tokens, stalls for them are from the TLB output (2 + 3 + 2, 64 entry queue shared by all cores per bank?)
- Each D$ bank has a set of tokens (~10 cycles round trip on hits) -> 32 entry per core per bank (thats a lot of queueing)

  Ouch 128 token sets?

What does OOO Core recovery look like on an error?

- I'm going to assume that the Checker core sends a "stop" signal on a dedicated TSV.
  - The OOO core then stops, and we can "reverse the direction" of the TSVs it uses (and repurpose for recover)
- The register file is certainly messed up....
  - We actually have the RF read bandwidth to read the whole RF at once (16 instructions * 2 src operands)
  - We have the TSV bandwidth for the whole RF at once
  - So we can probably recover this on the order of 5-10 (Checker Core) cycles: (latency for OOO to stop sending + data movement across Checker)

- If specific addresses in the D$ were messed up, we could clock them across the TSVs pretty quickly and do some special writes on the OOO Core
  - Load had right inputs but wrong output? Easy to guess one address that is wrong
  - May be hard to guess in general: add messes up -> shift -> ... -> store
  - Could invalidate entire D$ (and even L2) of offending core

- If wrong values have exited the D$ to the LLC and other cores, they may be wrong too.
  - Could invalidate every cache on the OOO Cores: big hammer

Deadlock or Livelock: if no instruction completes from a core within X cycles, Checker could do next instruction + signal error (restart from state after insn)

- Requires some slightly different datapaths in the checker than we have discussed

Proposed recovery strategy: escalating interventions

- Per core counter for cycle number (Checker TSC) since last error, and number of "recent" errors
- On error:
  - If time since last error < threshold: increment recent errors
  - Else set recent errors to 1
  - Severity = f(time since last error, number of recent errors)
  - Set last error time to now
  - If severity is low, just recover offending core's register file
  - If severity is medium, recover offending core's register file + clear L2
  - If severity is high, recover all cores' register files + invalidate all OOO core caches