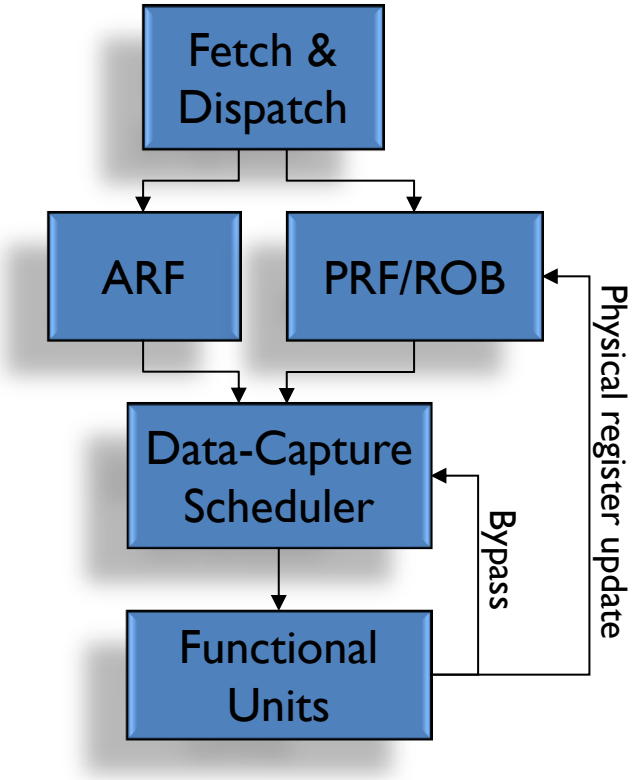
CSE 502:
Computer Architecture

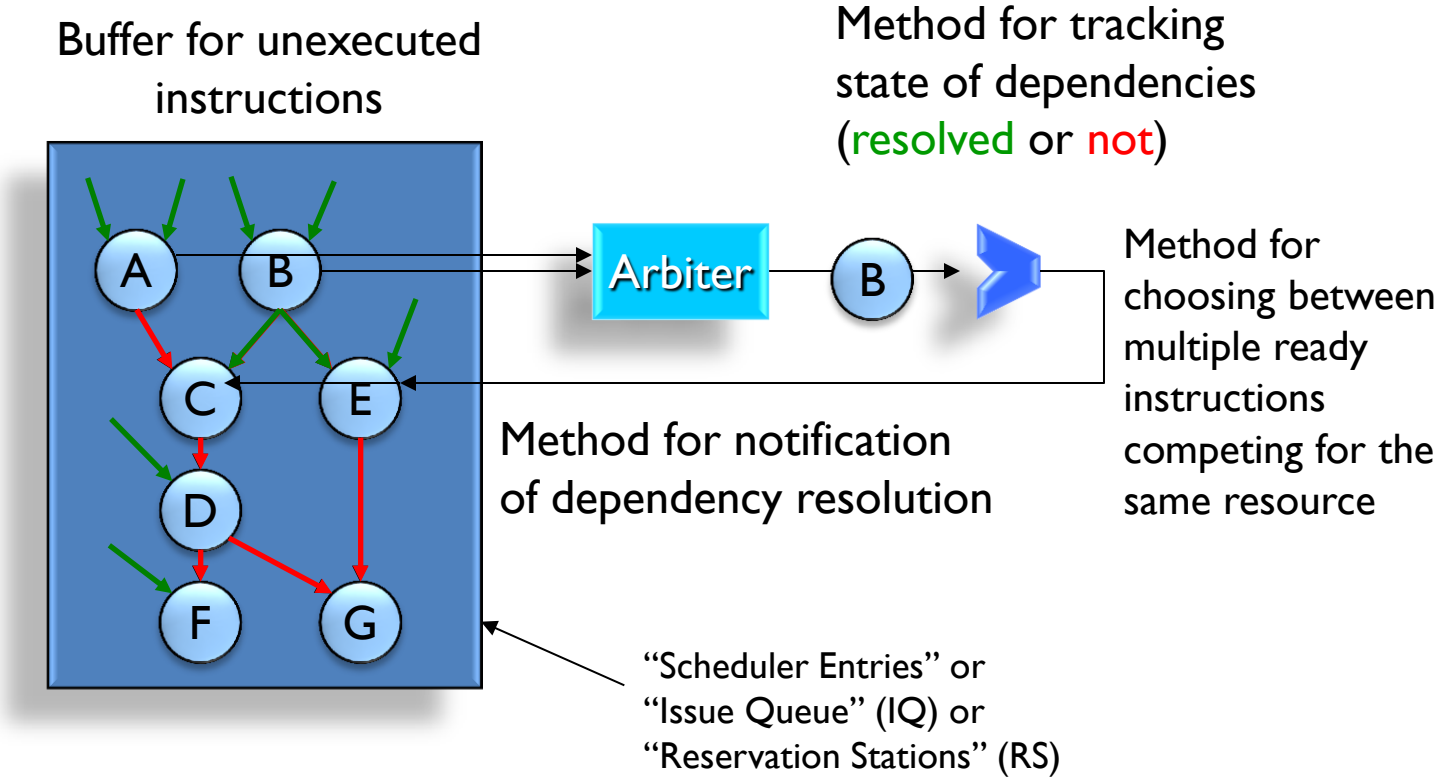
Out-of-Order Schedulers

Data-Capture Scheduler

- Dispatch: read available operands from ARF/ROB, ***store*** in scheduler
- Commit: Missing operands filled in from bypass
- Issue: When ready, operands sent directly from scheduler to functional units



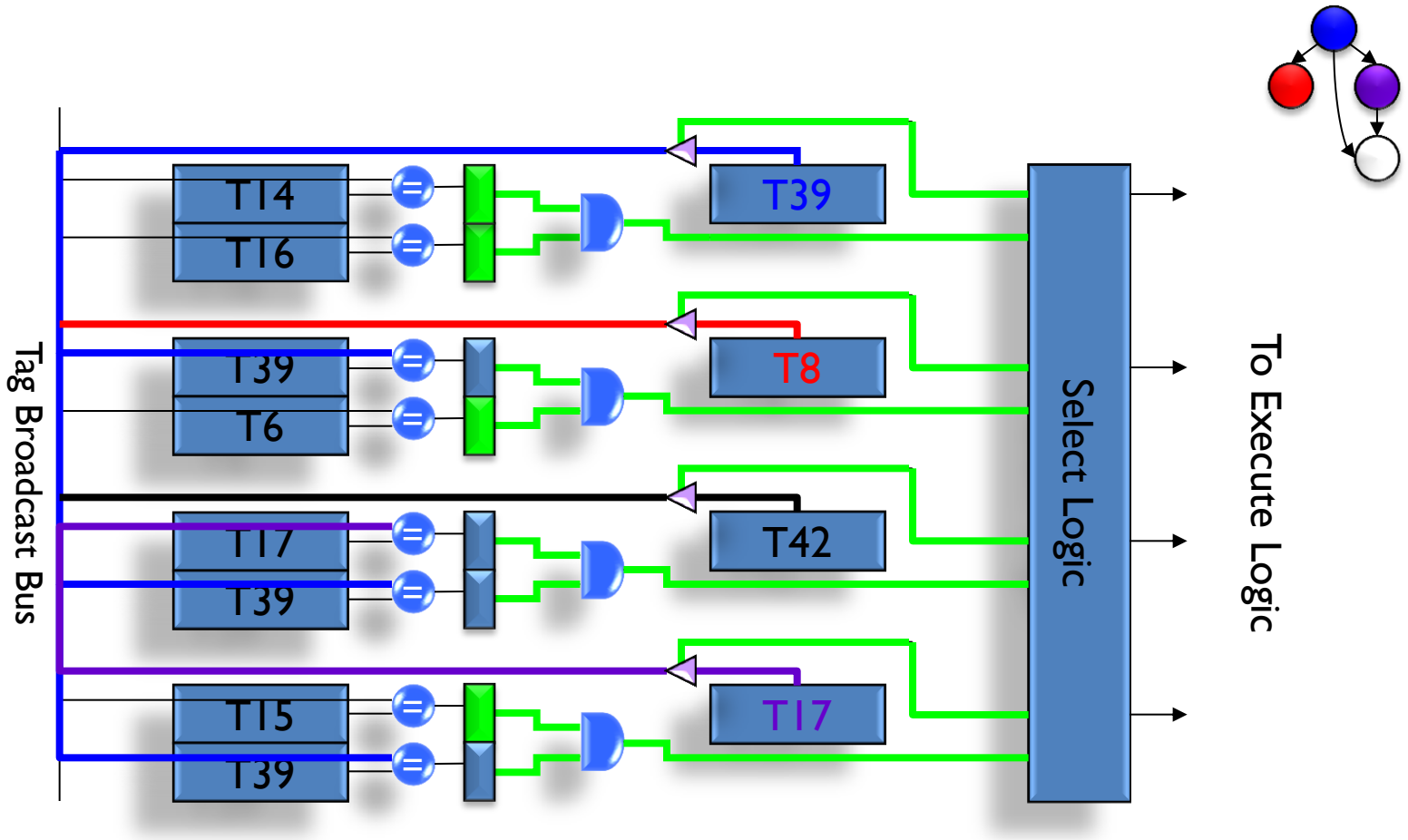
Components of a Scheduler



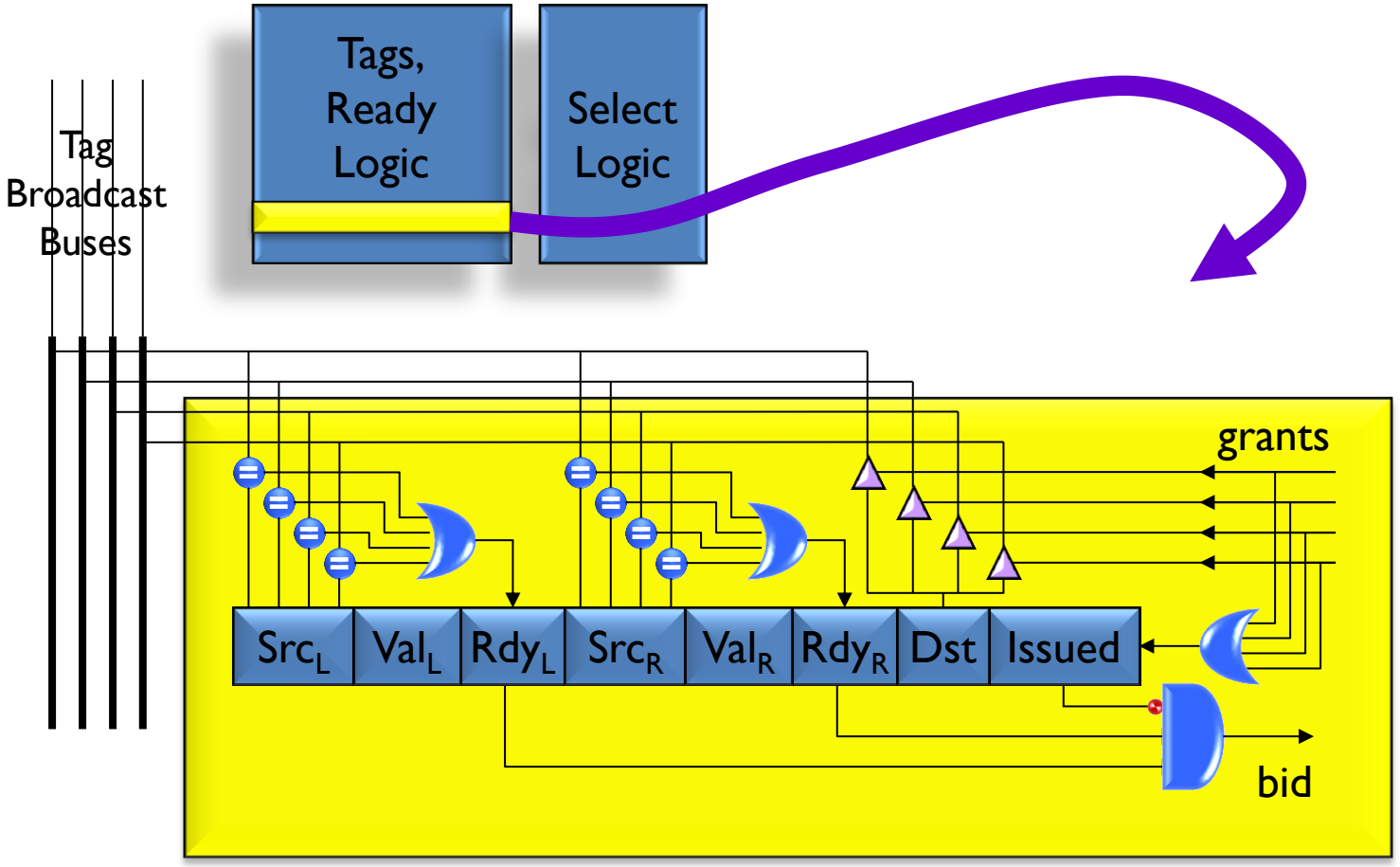
Scheduling Loop or *Wakeup-Select Loop*

- Wake-Up Part:
  - Executing insn notifies dependents
  - Waiting insns. check if all deps are satisfied
    - If yes, “wake up” instutrction
- Select Part:
  - Choose which instructions get to execute
    - More than one insn. can be ready
    - Number of functional units and memory ports are limited

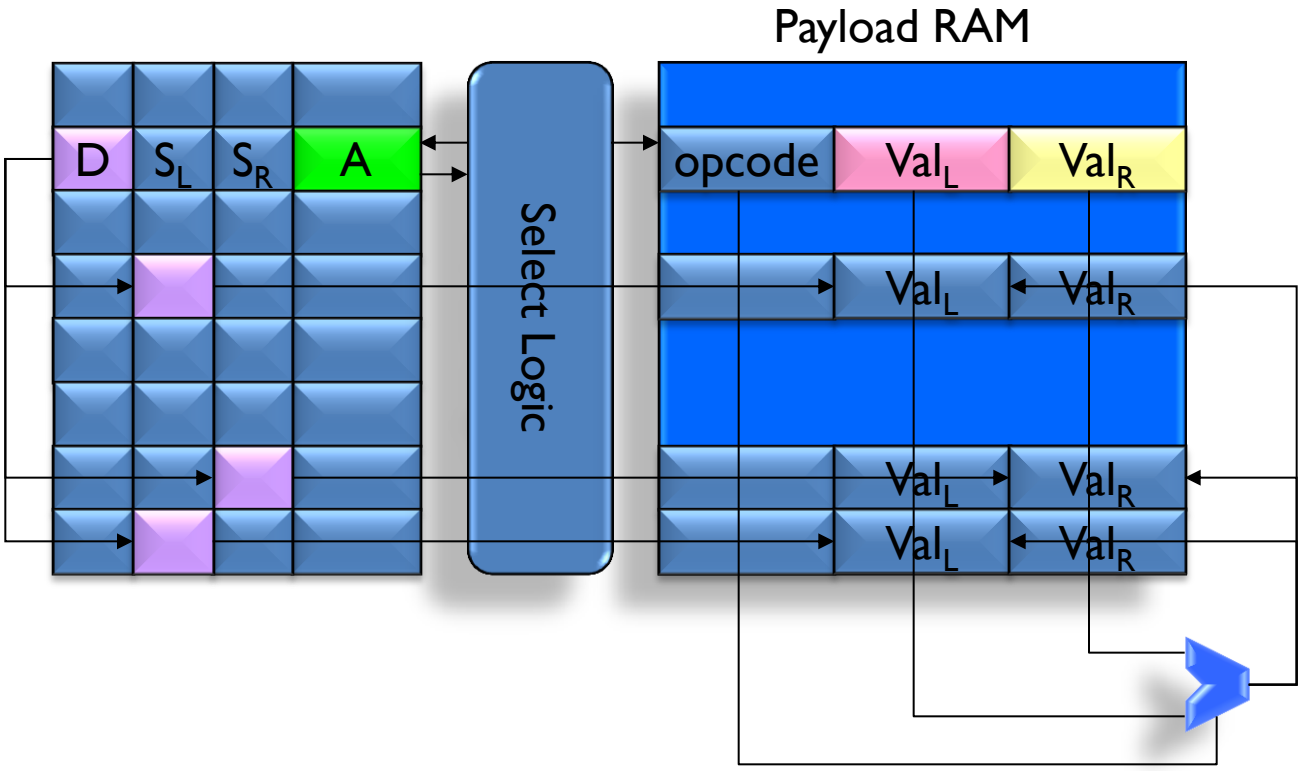
Scalar Scheduler (Issue Width = 1)



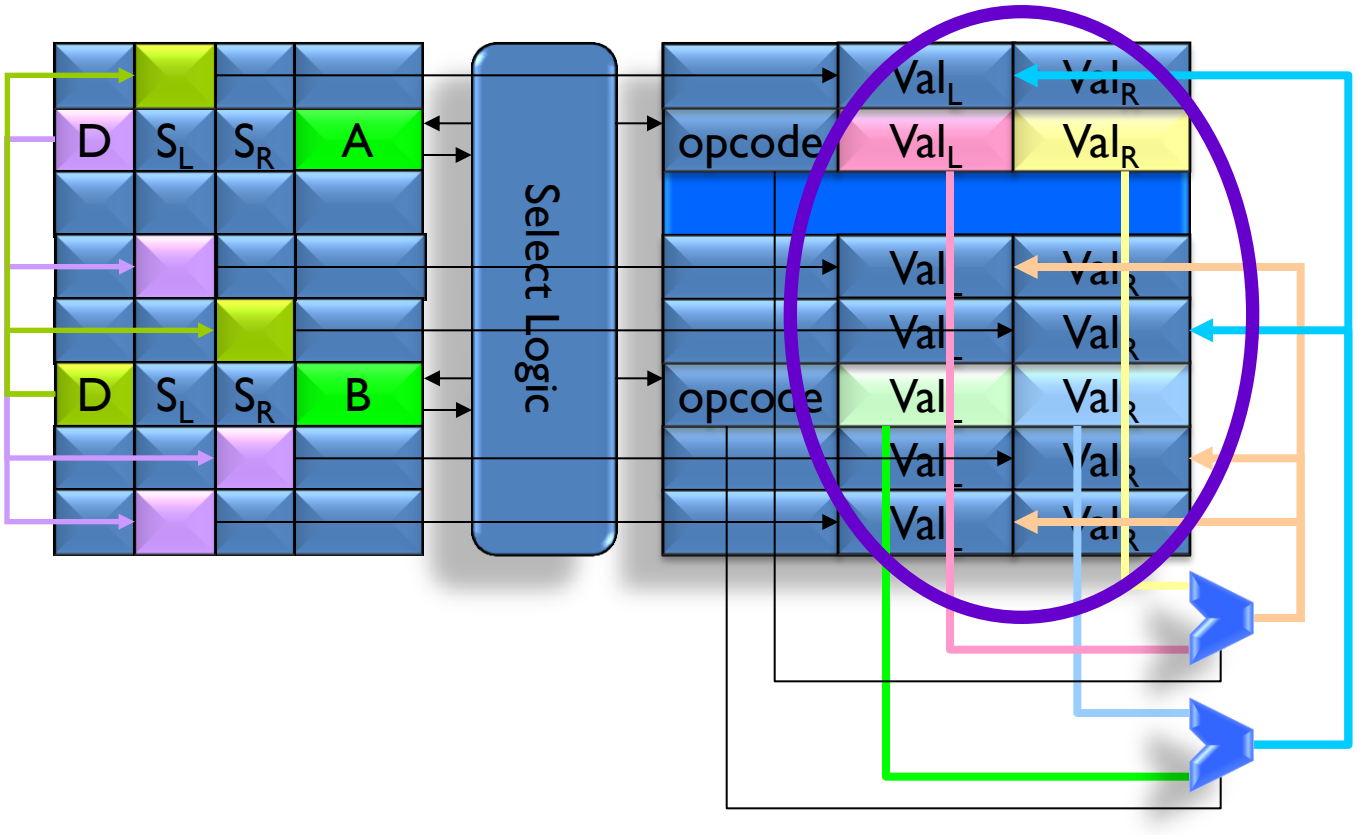
Superscalar Scheduler (detail of one entry)



Interaction with Execution



Again, But Superscalar

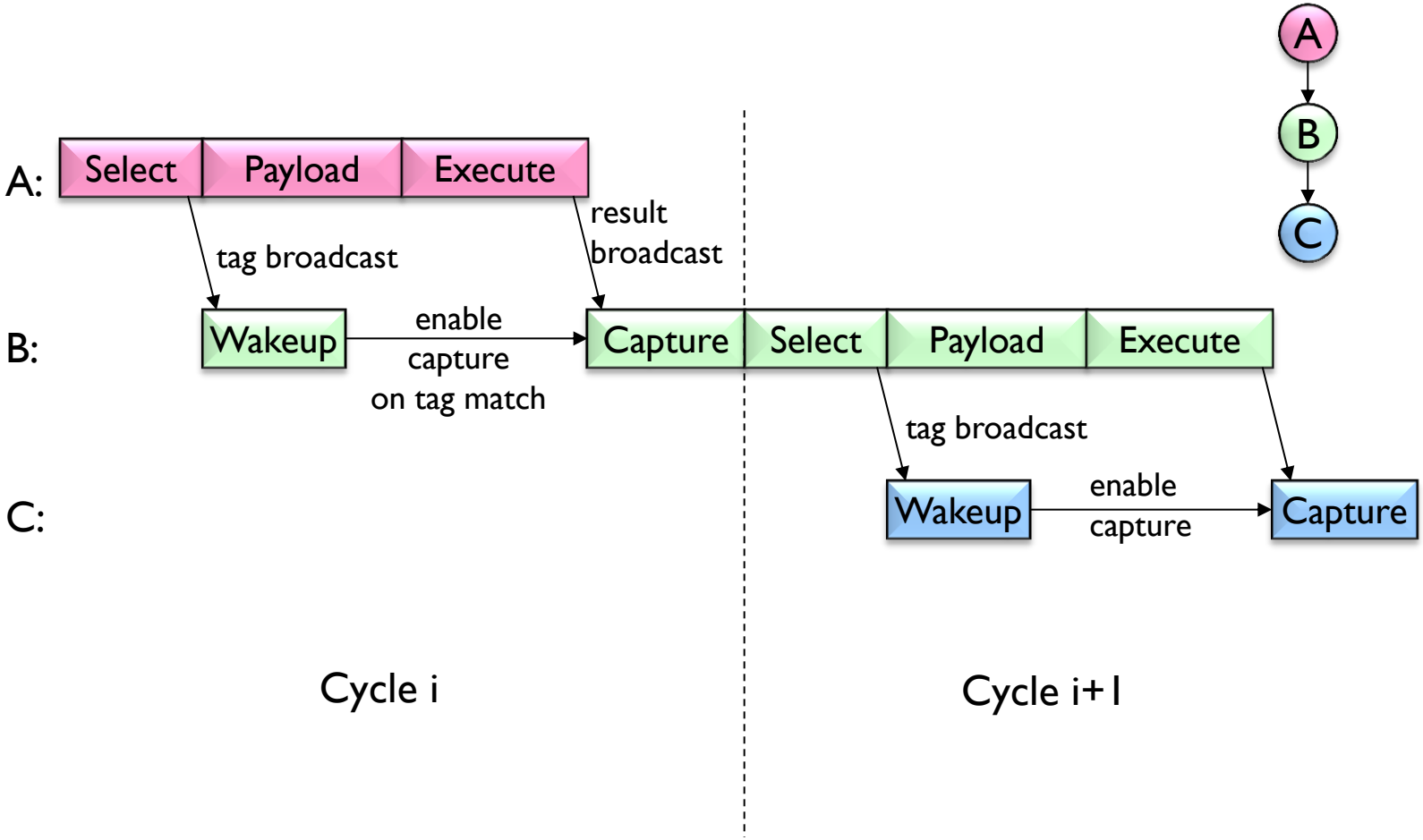


Scheduler ***captures*** values

Issue Width

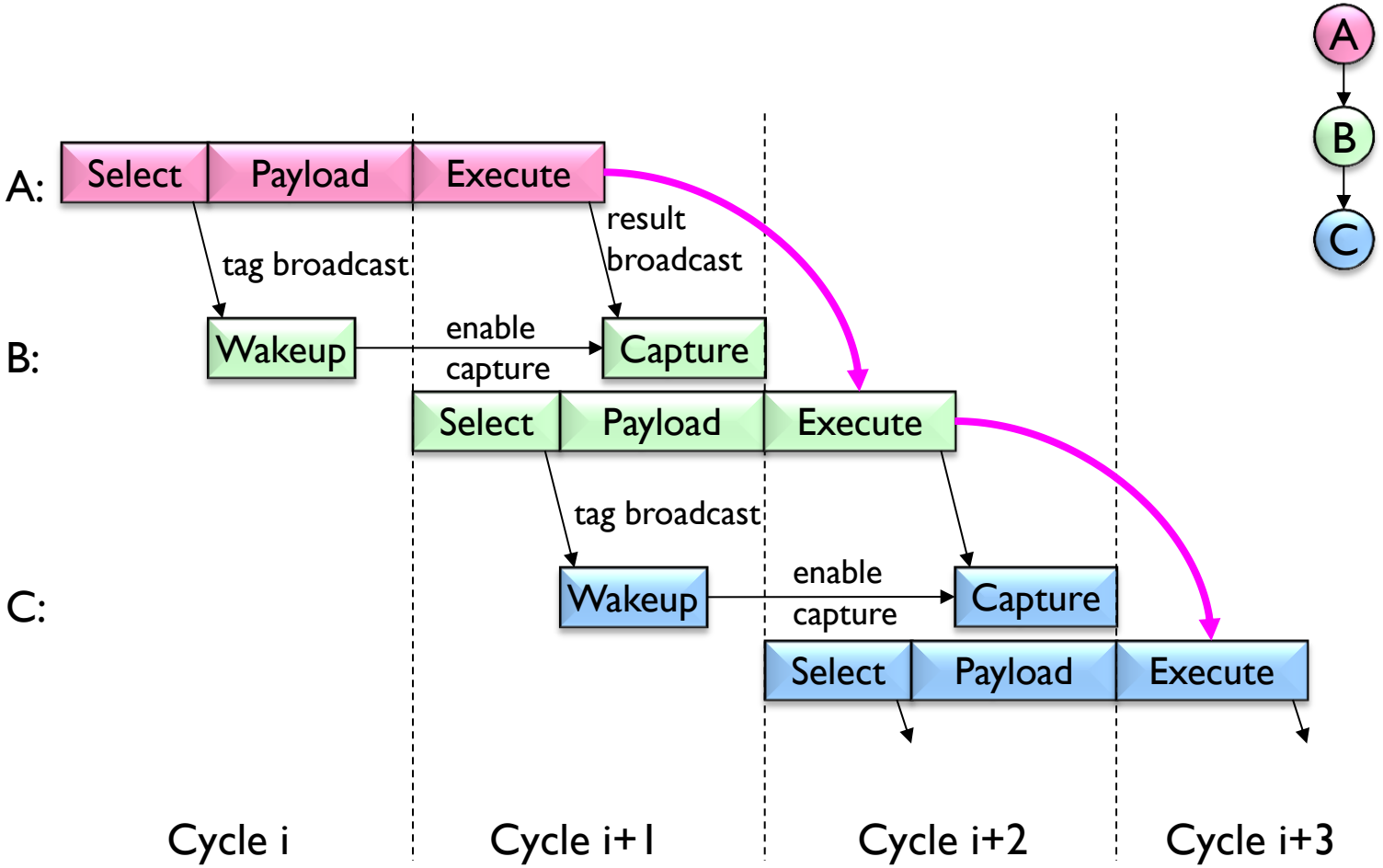
- Max insns. selected each cycle is *issue width*
  - Previous slides showed different issue widths
    - four, one, and two
- Hardware requirements:
  - Naively, issue width of N requires N tag broadcast buses
  - Can "specialize" some of the issue slots
    - E.g., a slot that only executes branches (no outputs)

Simple Scheduler Pipeline



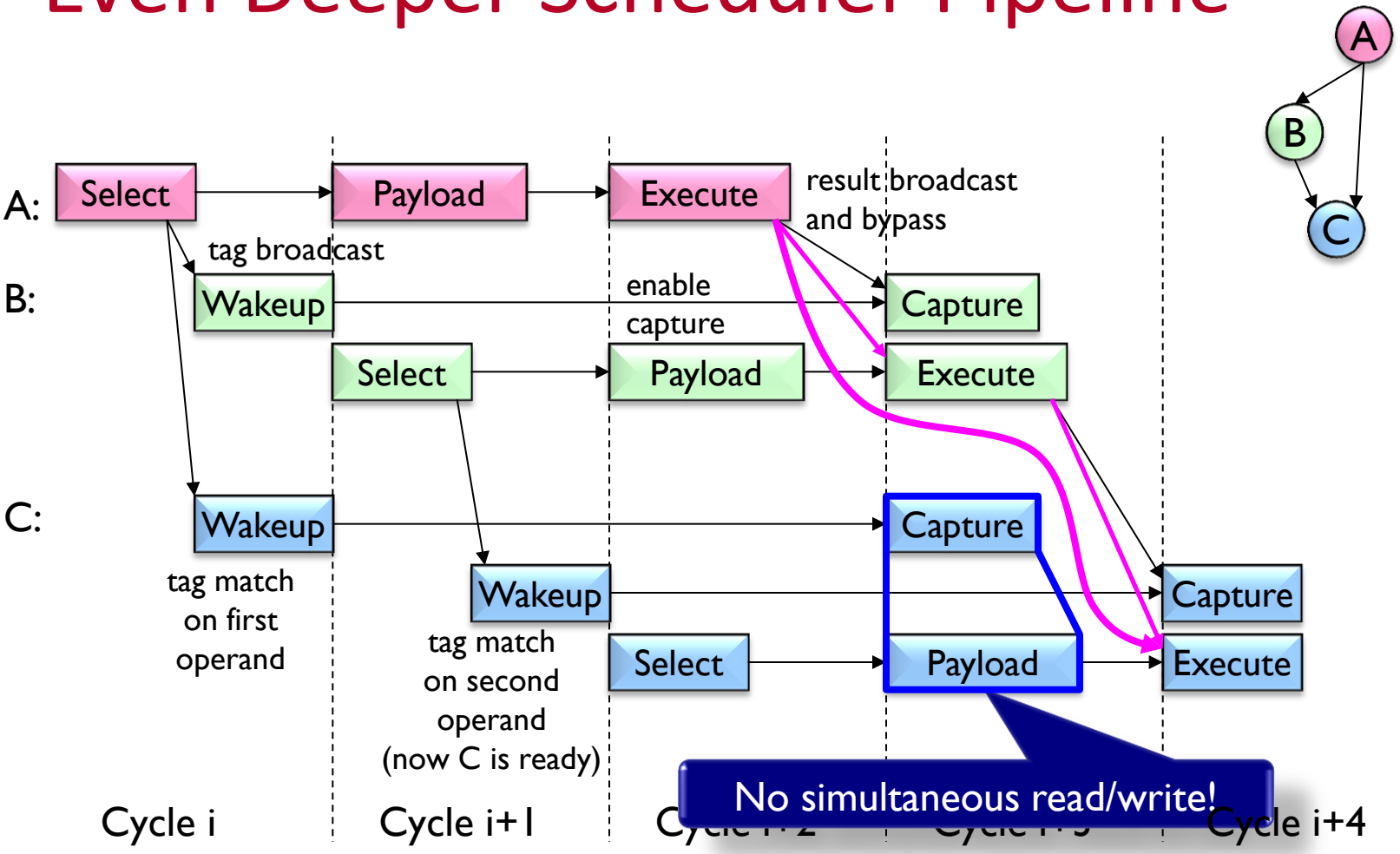
Very long clock cycle

Deeper Scheduler Pipeline



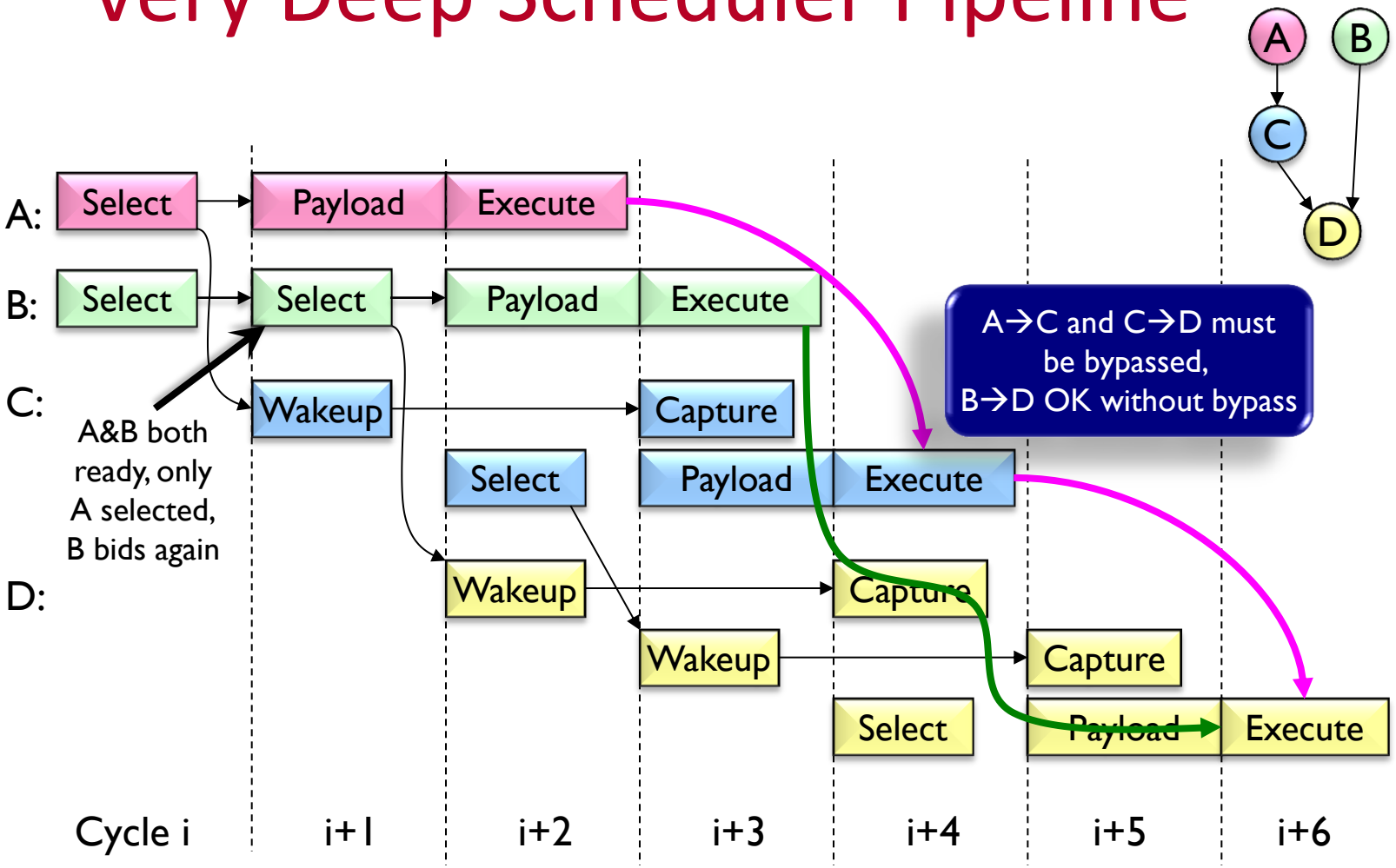
Faster, but Capture & Payload on same cycle

Even Deeper Scheduler Pipeline



Need ***second*** level of bypassing

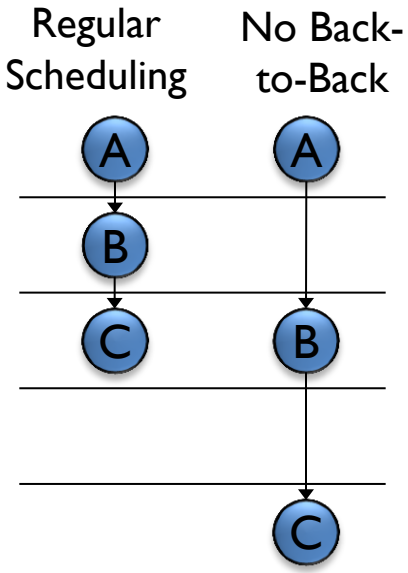
Very Deep Scheduler Pipeline



Dependent instructions can't execute back-to-back

Pipelineing Critical Loops

- Wakeup-Select Loop hard to pipeline
  - No back-to-back execute
  - Worst-case IPC is ½
- Usually not worst-case
  - Last example had IPC $^{2}/_{3}$

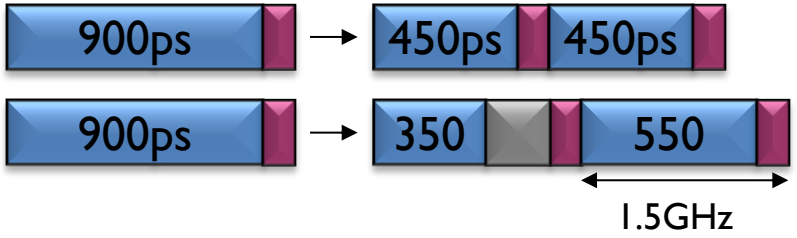


Studies indicate 10-15% IPC penalty

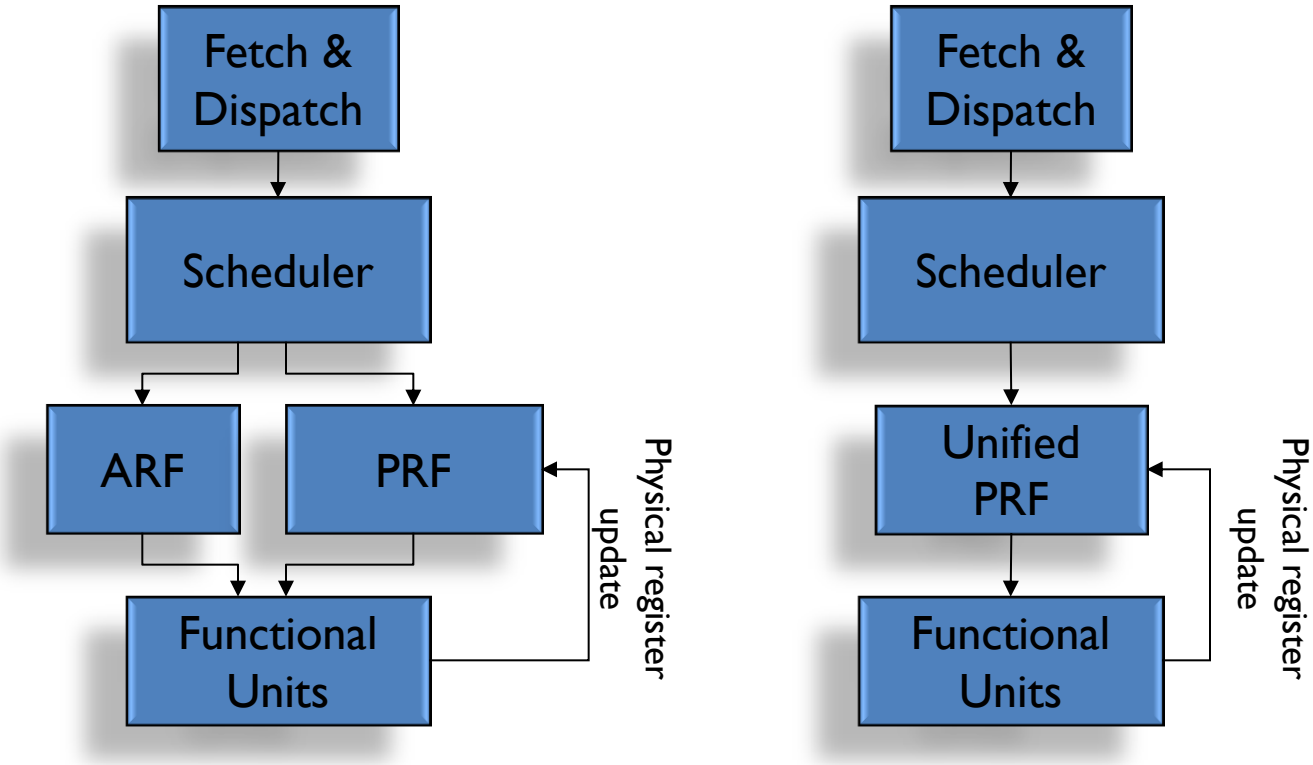
IPC vs. Frequency

- 10-15% IPC not bad if frequency can double

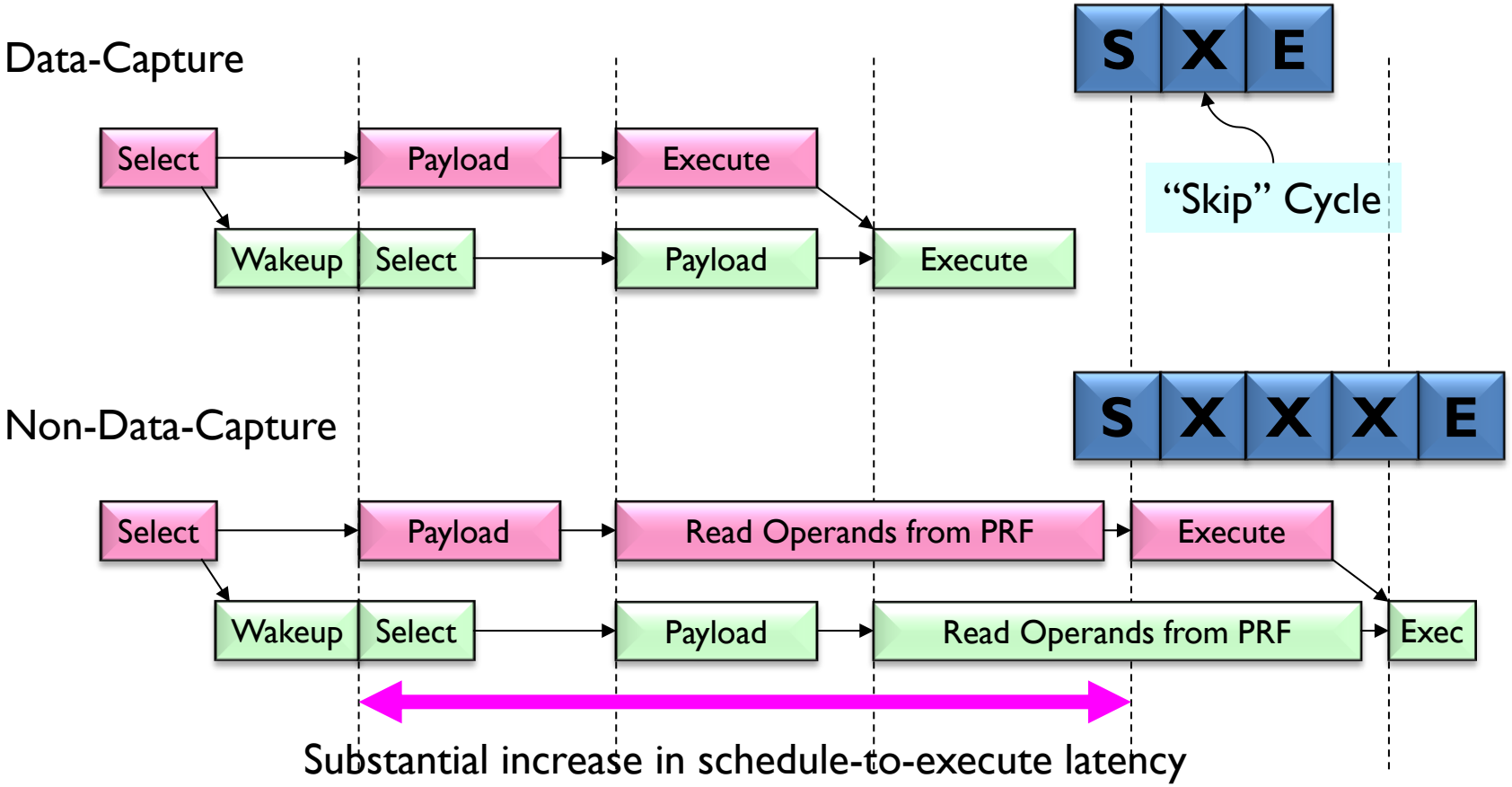
- Frequency doesn't double
  - Latch/pipeline overhead
  - Stage imbalance



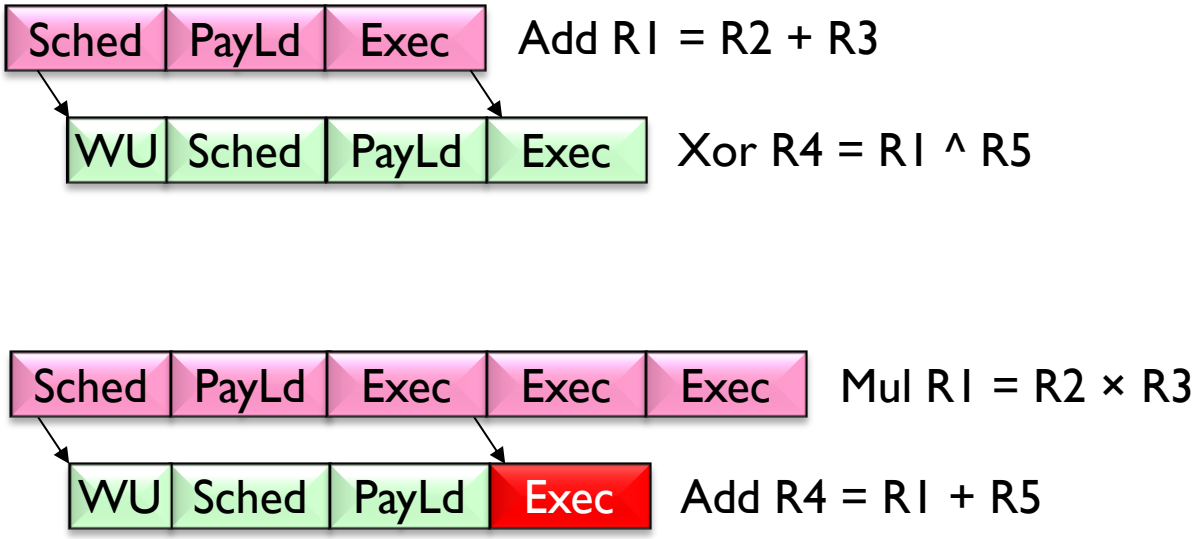
Non-Data-Capture Scheduler



Pipeline Timing

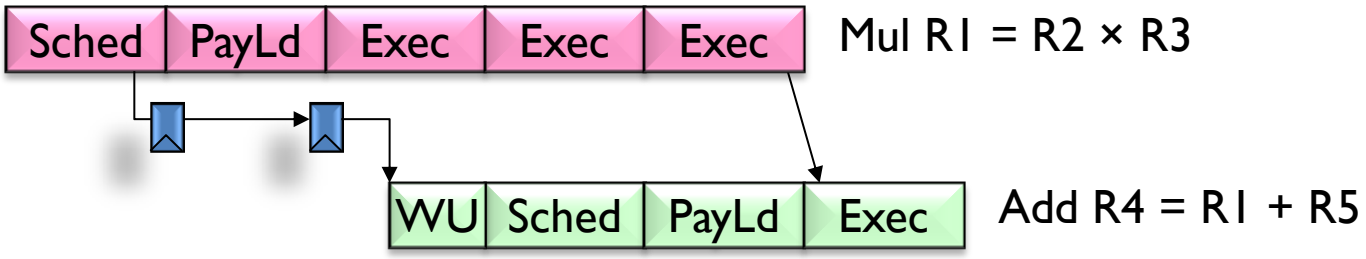


Handling Multi-Cycle Instructions



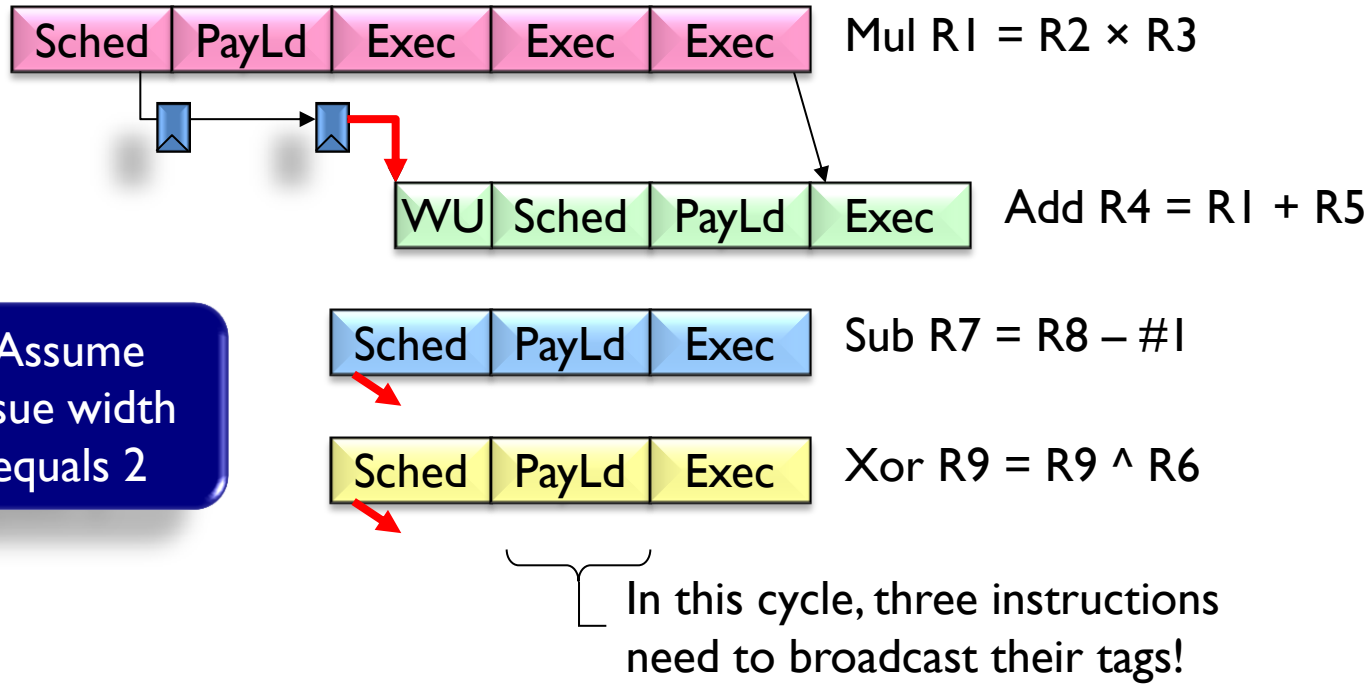
Instructions can't execute ***too early***

Delayed Tag Broadcast (1/3)



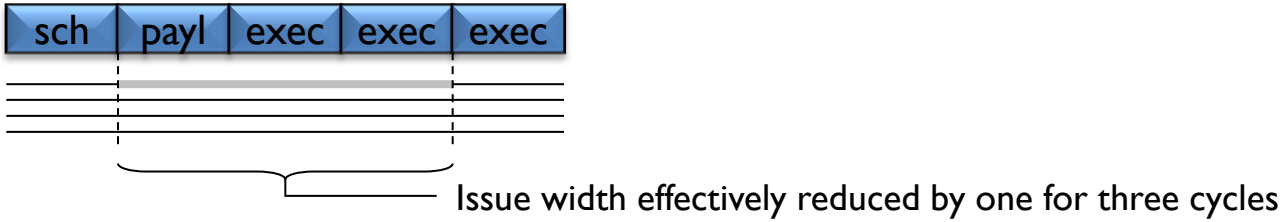
- Must make sure broadcast bus available in future
- Bypass and data-capture get more complex

Delayed Tag Broadcast (2/3)



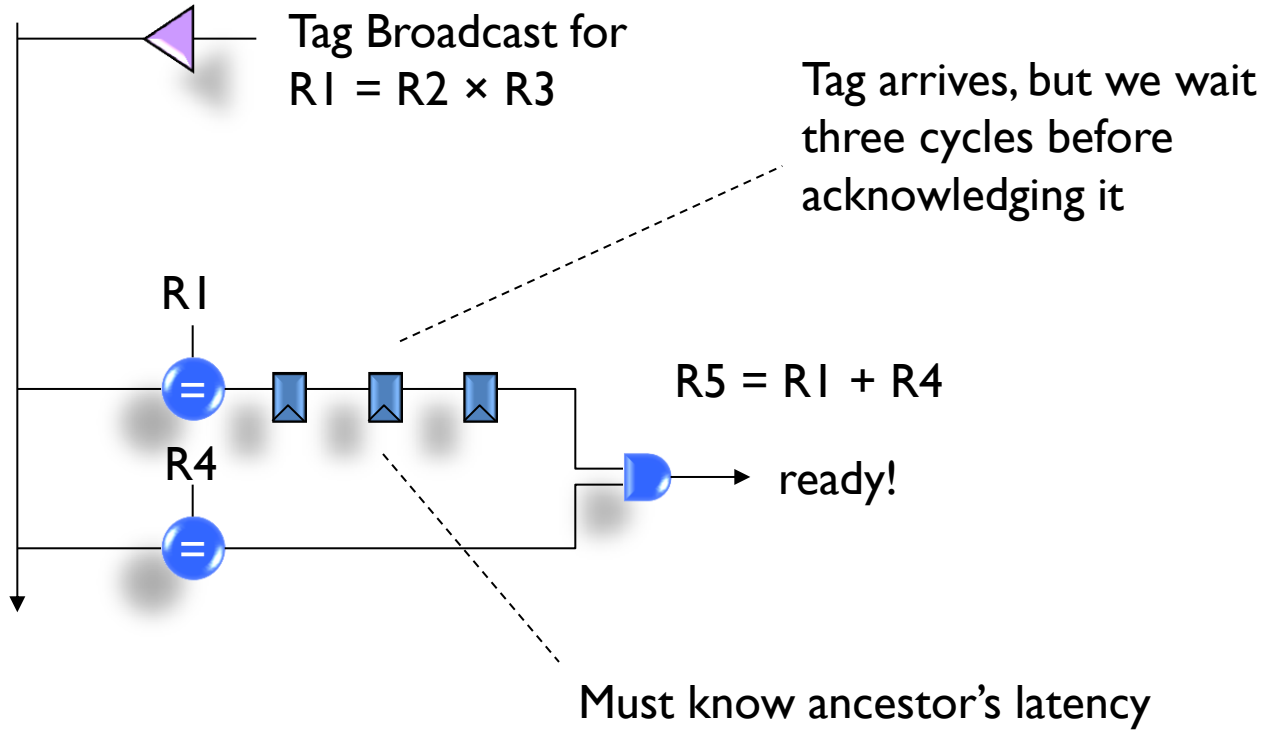
Delayed Tag Broadcast (3/3)

- Possible solutions
  1. One select for issuing, another select for tag broadcast
     - Messes up timing of data-capture
  2. Pre-reserve the bus
     - Complicated select logic, track future cycles in addition to current
  3. Hold the issue slot from initial launch until tag broadcast



Delayed Wakeup

- Push the delay to the consumer



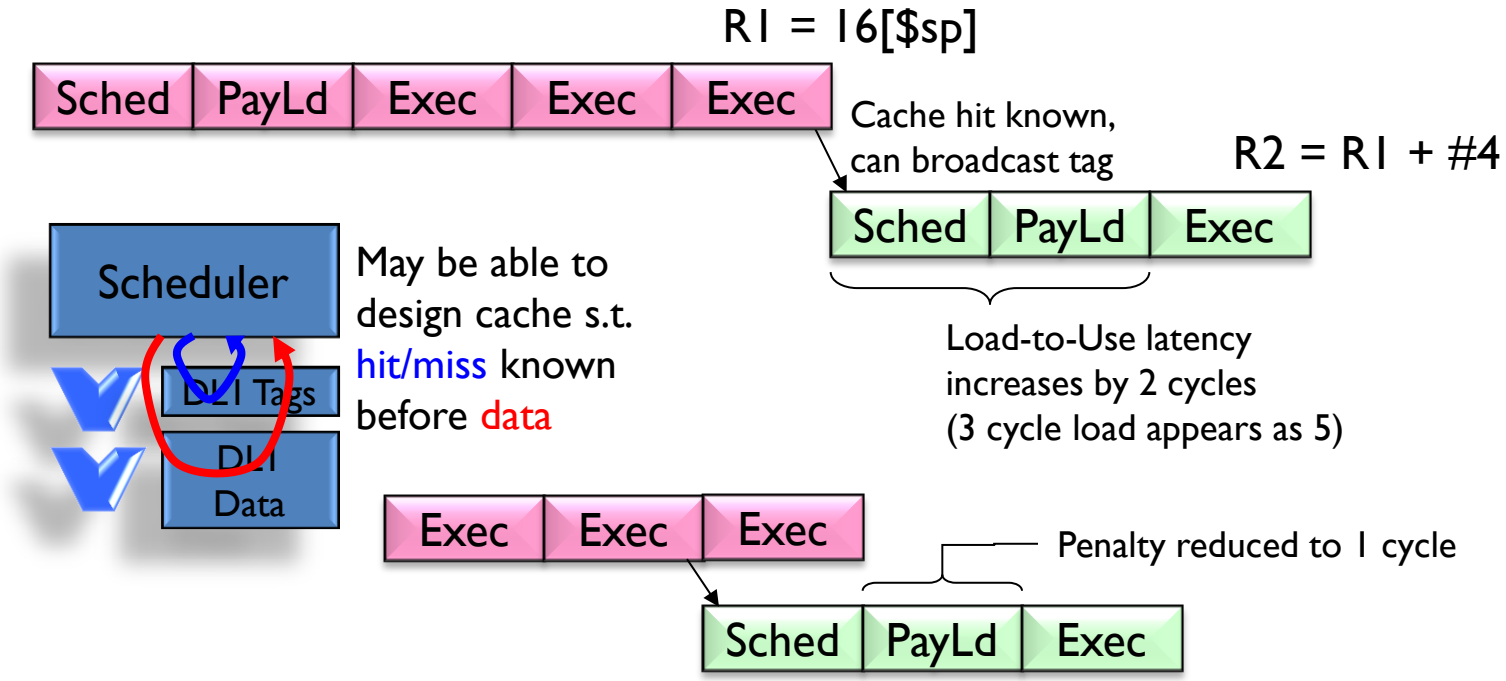
Non-Deterministic Latencies

- Previous approaches assume all latencies are known
- Real situations have unknown latency
  - Load instructions
    - Latency ∈ {L1_lat, L2_lat, L3_lat, DRAM_lat}
    - DRAM_lat is not a constant either, queuing delays
  - Architecture specific cases
    - PowerPC 603 has "*early out*" for multiplication
    - Intel Core 2's has early out divider also

- Makes delayed broadcast hard
- Kills delayed wakeup

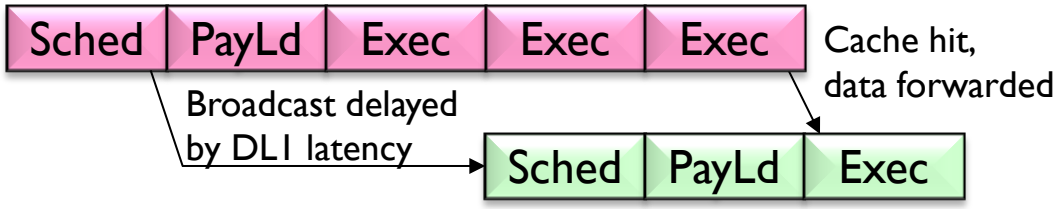
The Wait-and-See Approach

- Complexity only in the case of variable-latency ops
  - Most insns. have known latency
- Wait to learn if load hits or misses in the cache



Load-Hit Speculation

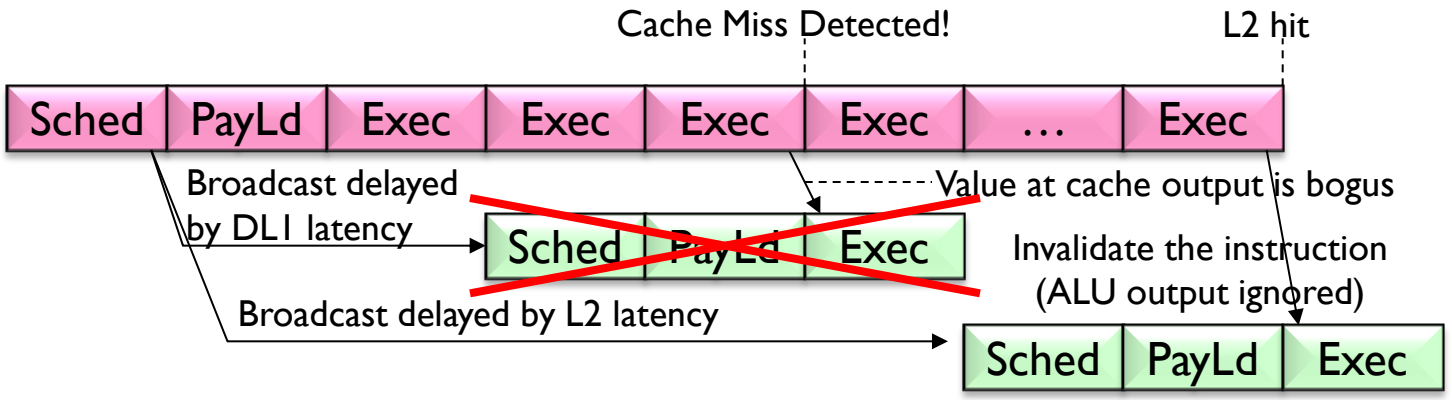
- Caches work pretty well
  - Hit rates are high (otherwise we wouldn't use caches)
  - Assume all loads hit in the cache



R1 = 16[$sp]

R2 = R1 + #4

Load-Hit Mis-speculation

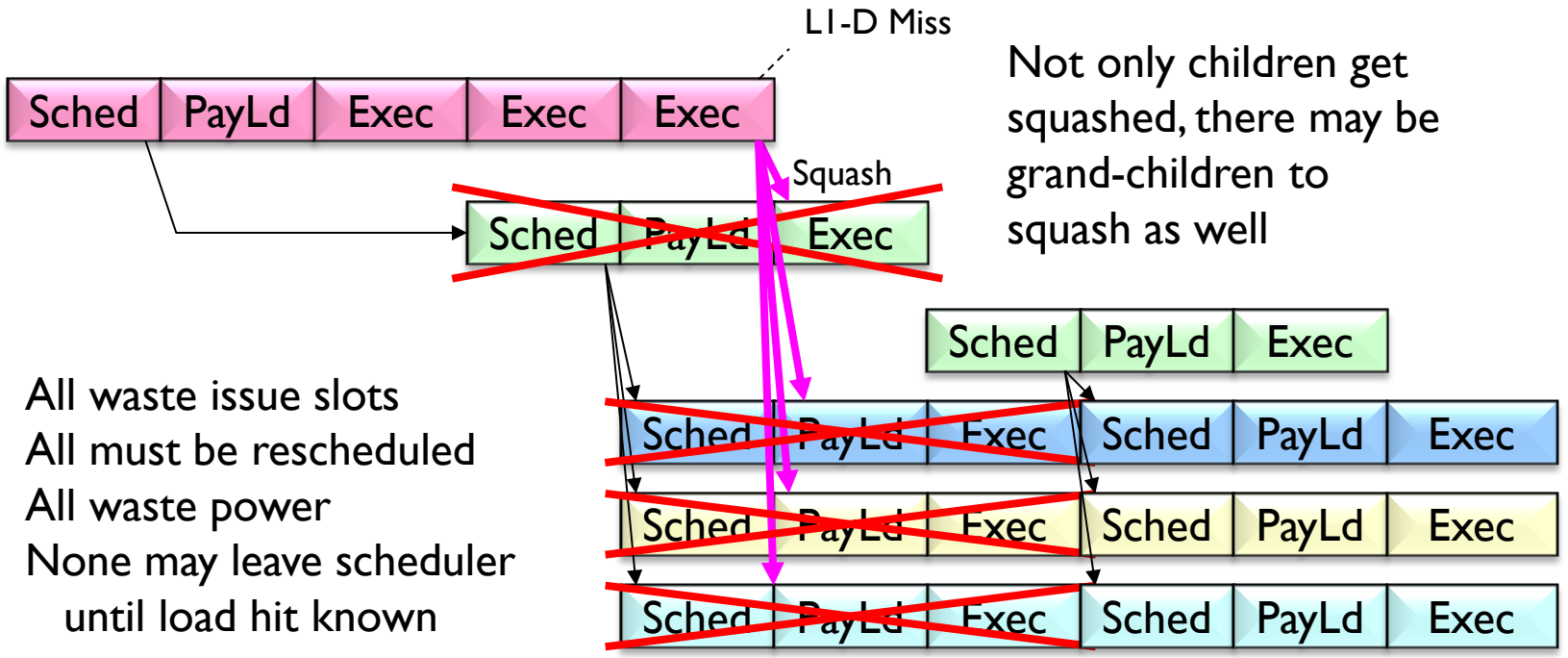


Each mis-scheduling wastes an issue slot: the tag broadcast bus, payload RAM read port, writeback/bypass bus, etc. could have been used for another instruction

There could be a miss at the L2 and again at the L3 cache.
A single load can waste multiple issuing opportunities.

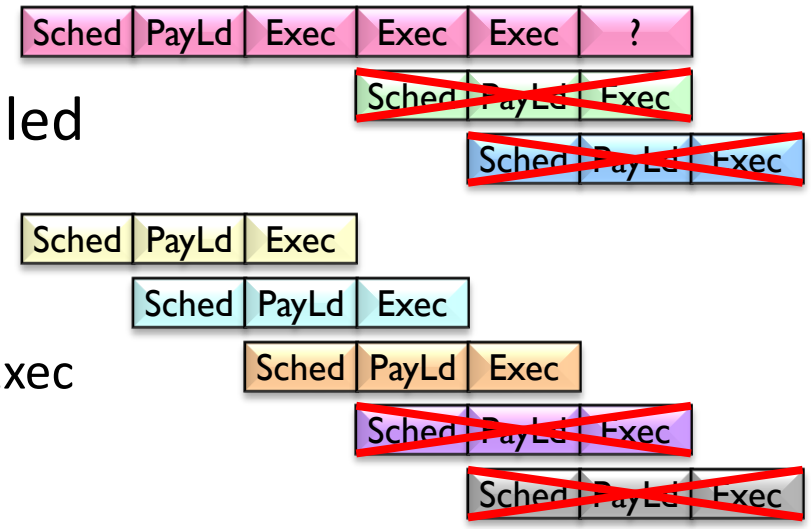
It's hard, but we want this for performance

"But wait, there's more!"



Squashing (1/3)

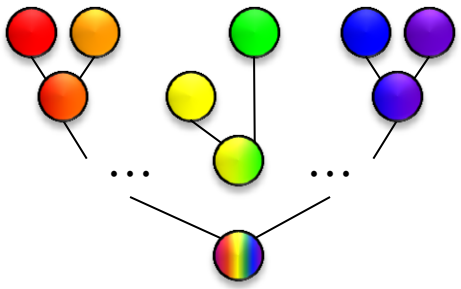
- Squash "in-flight" between schedule and execute
  - Relatively simple (each RS remembers that it was issued)
- Insns. stay in scheduler
  - Ensure they are not re-scheduled
  - Not too bad
    - Dependents issued ***in order***
    - Mis-speculation known before Exec



May squash non-dependent instructions

Squashing (2/3)

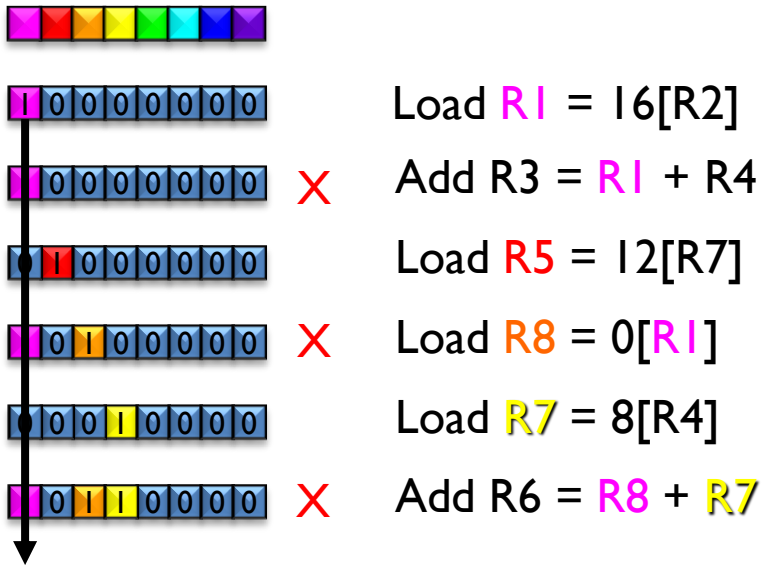
- Selective squashing with "load colors"
  - Each load assigned a unique color
  - Every dependent "inherits" parents' colors
  - On load miss, the load broadcasts its color
    - Anyone in the same color group gets squashed
- An instruction may end up with many colors



Tracking colors requires huge number of comparisons

Squashing (3/3)

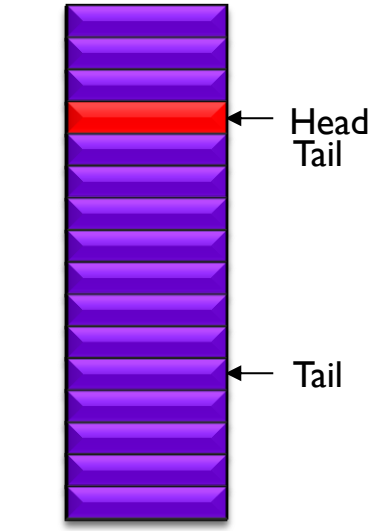
- Can list "colors" in unary (bit-vector) form
  - Each insn.'s vector is bitwise OR of parents' vectors



Allows squashing just the dependents

Scheduler Allocation (1/3)

- Allocate in order, deallocate in order
  - Very simple!
- Reduces effective scheduler size
  - Insns. executed out-of-order
    … RS entries cannot be reused

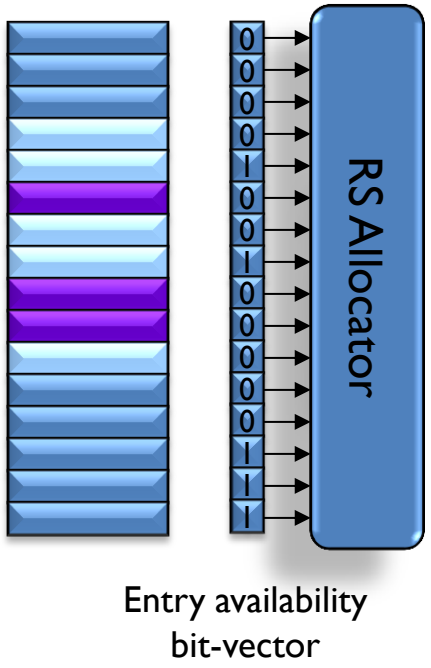


Circular Buffer

Can be terrible if load goes to memory

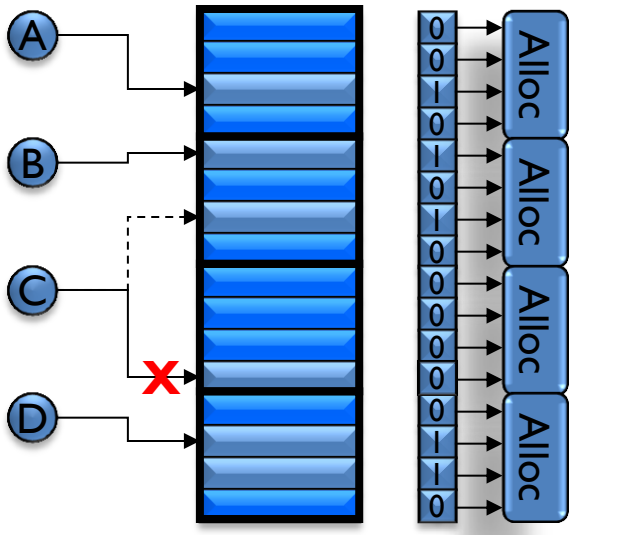
Scheduler Allocation (2/3)

- Arbitrary placement improves utilization
- Complex allocator
  - Scan availability to find N free entries
- Complex write logic
  - Route N insns. to arbitrary entries



Scheduler Allocation (3/3)

- Segment the entries
  - One entry per segment may be allocated per cycle
  - Each allocator does 1-of-4
    - instead of 4-of-16 as before
  - Write logic is simplified
- Still possible inefficiencies
  - Full segments block allocation
  - Reduces dispatch width

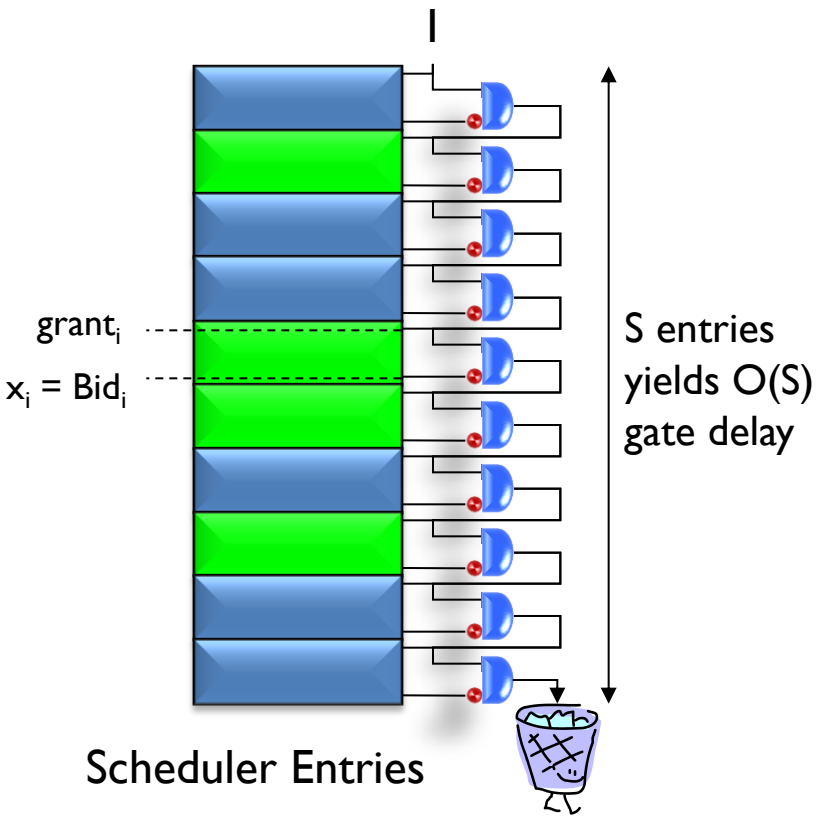


Free RS entries exist, just not in the correct segment

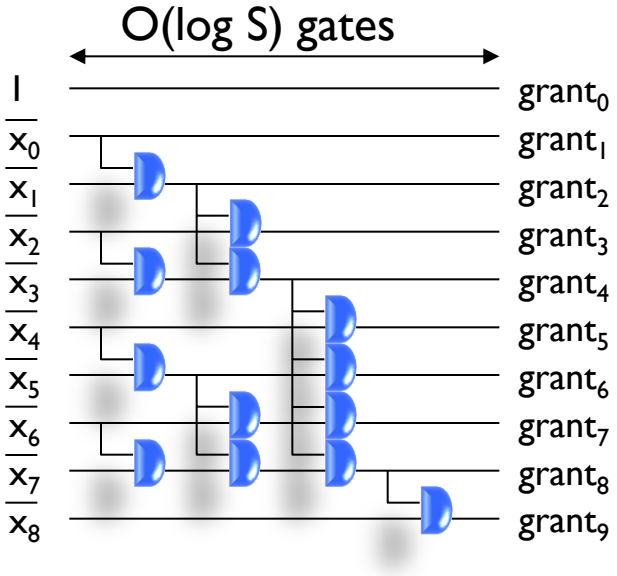
Select Logic

- Goal: minimize DFG height (execution time)
- NP-Hard
  - Precedence Constrained Scheduling Problem
  - Even harder: entire DFG is not known at scheduling time
    - Scheduling insns. may affect scheduling of not-yet-fetched insns.
- Today's designs implement heuristics
  - For performance
  - For ease of implementation

Simple Select Logic



$Grant_0 = I$
$Grant_1 = !Bid_0$
$Grant_2 = !Bid_0 \ \& \ !Bid_1$
$Grant_3 = !Bid_0 \ \& \ !Bid_1 \ \& \ !Bid_2$
$Grant_{n-1} = !Bid_0 \ \& \ \ldots \ \& \ !Bid_{n-2}$



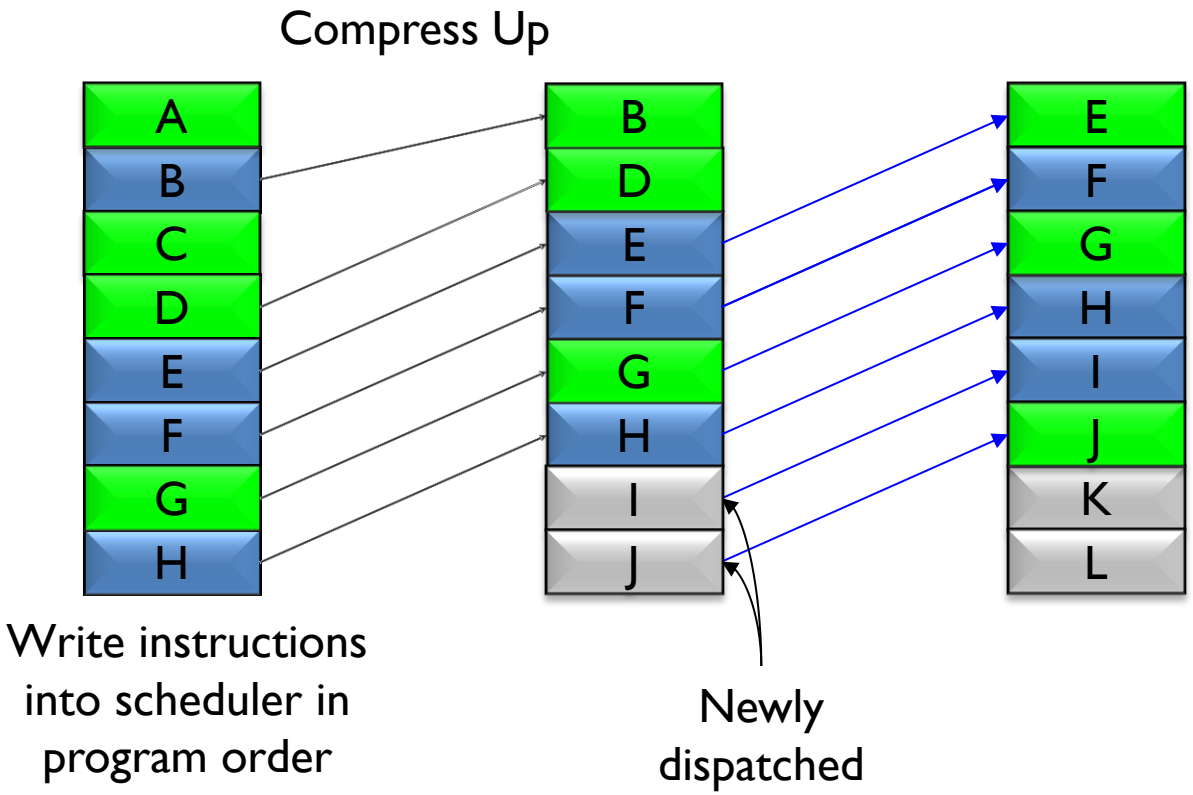
Random Select

- Insns. occupy arbitrary scheduler entries
  - First ready entry may be the oldest, youngest, or in middle
  - Simple static policy results in "random" schedule
    - Still "correct" (no dependencies are violated)
    - Likely to be far from optimal

Oldest-First Select

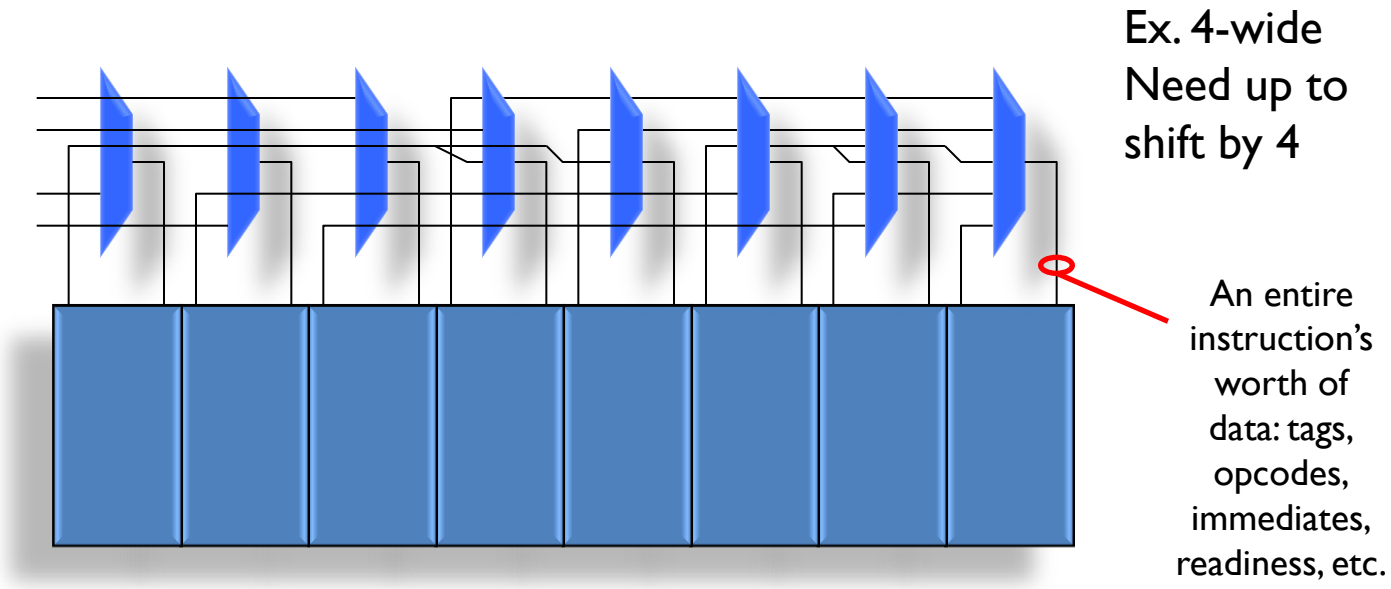
- Newly dispatched insns. have few dependencies
  - No one is waiting for them yet
- Insns. in scheduler are likely to have the most deps.
  - Many new insns. dispatched since old insn's rename
- Selecting ***oldest*** likely satisfies more dependencies
  - … finishing it sooner is likely to make more insns. ready

Implementing Oldest First Select (1/3)

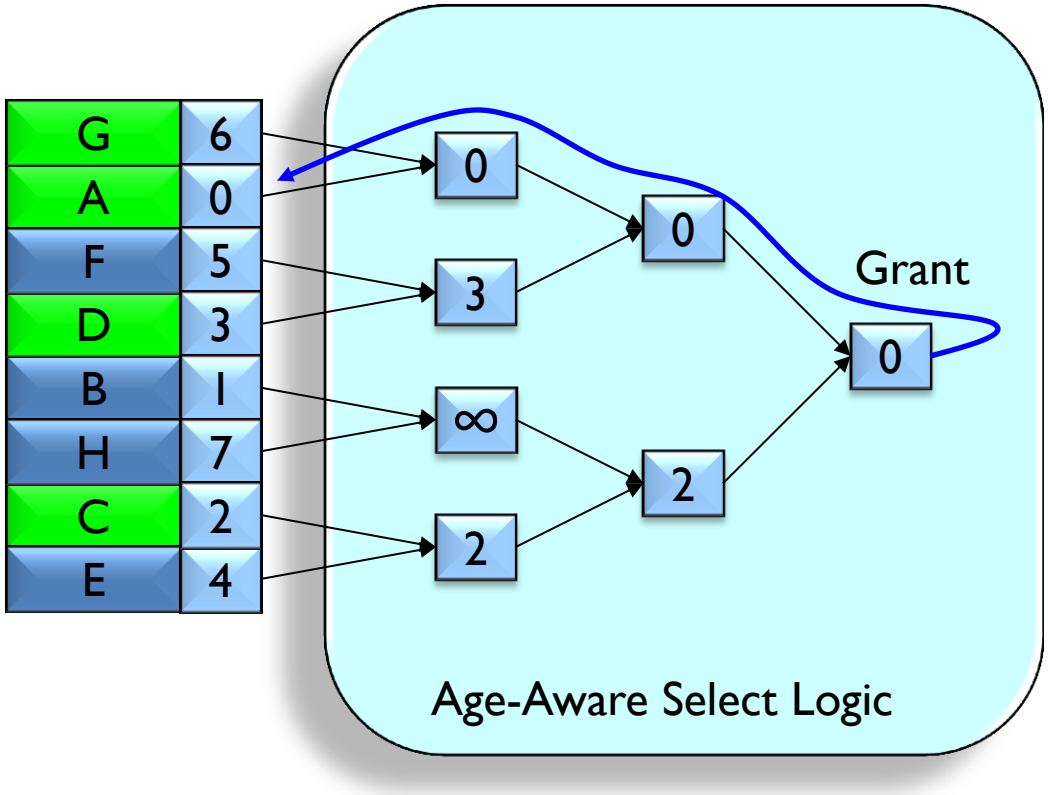


Implementing Oldest First Select (2/3)

- Compressing buffers are very complex
  - Gates, wiring, area, power

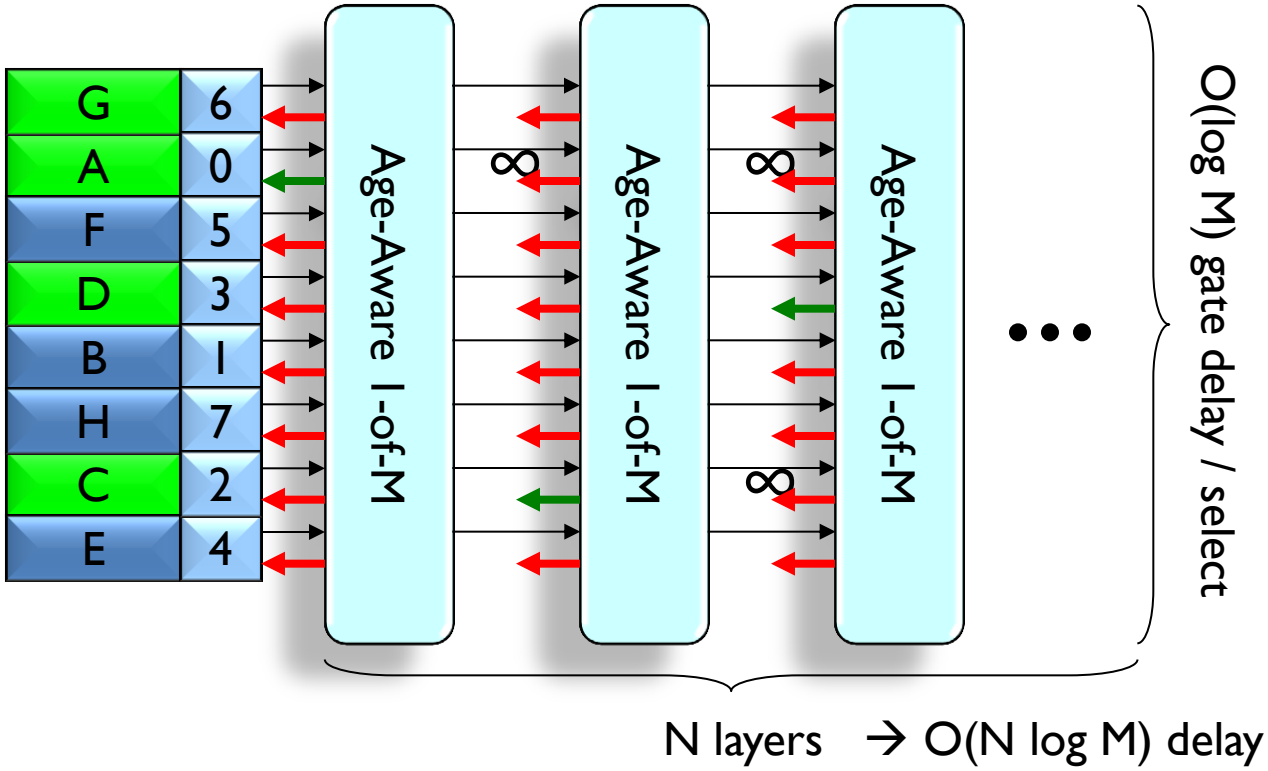


Implementing Oldest First Select (3/3)



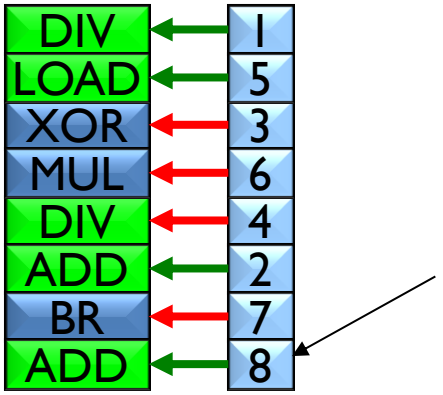
Must broadcast grant age to instructions

Problems in N-of-M Select (1/2)



Problems in N-of-M Select (2/2)

- Select logic handles functional unit constraints
  - Maybe two instructions ready this cycle
    … but both need the divider

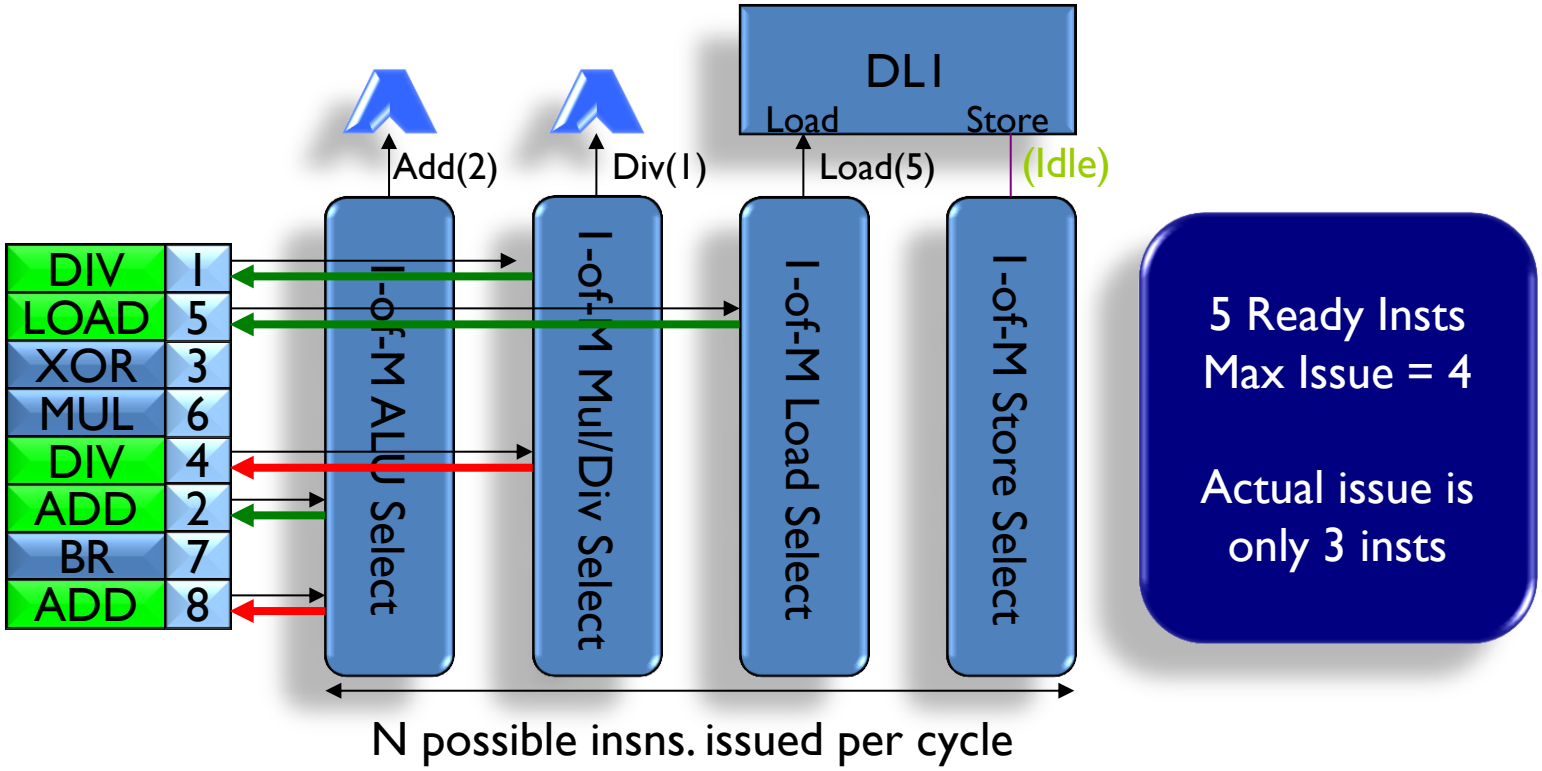


Assume issue width = 4

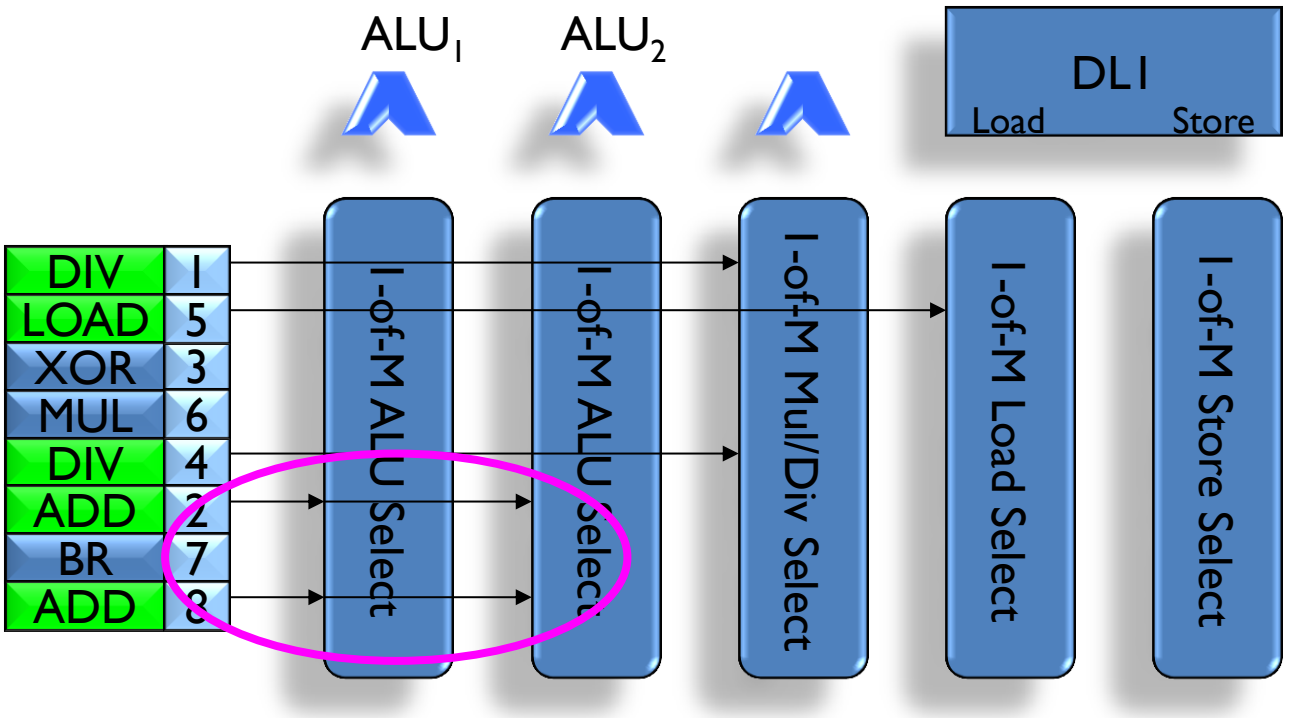
Four ***oldest and ready*** instructions

ADD is the 5th oldest ready instruction, but it should be issued because only one of the ready divides can issue this cycle

Partitioned Select

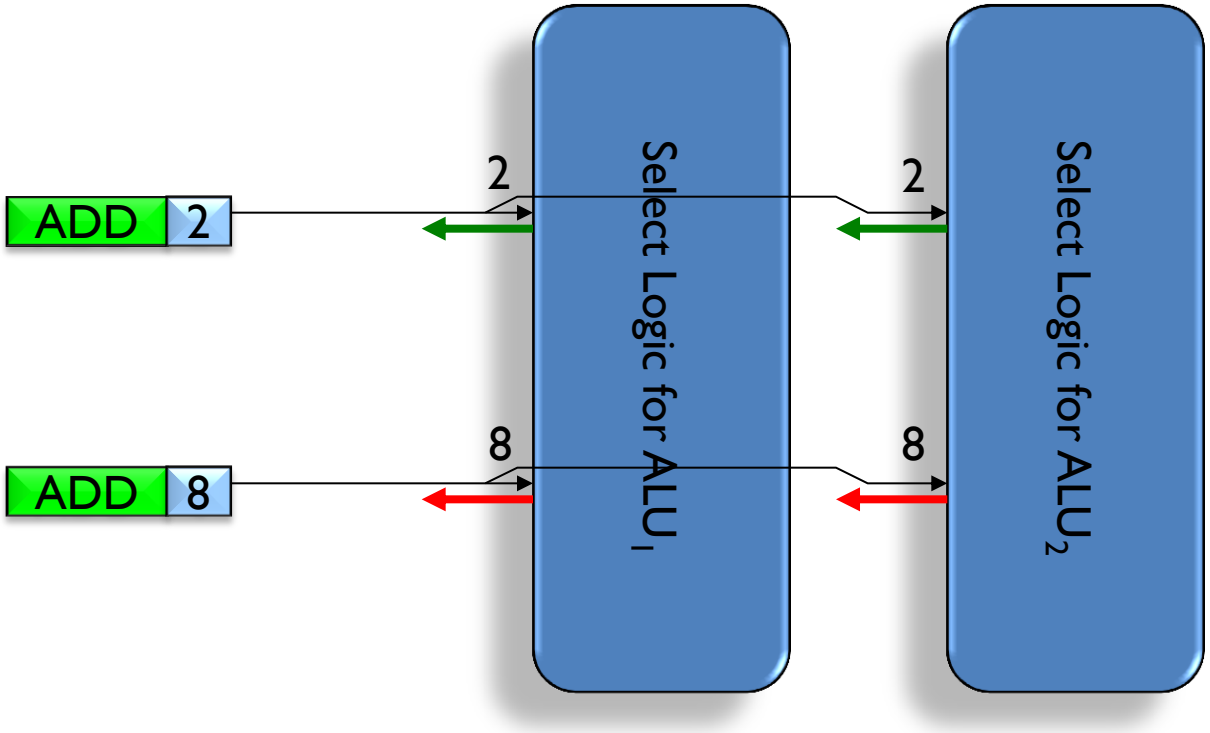


Multiple Units of the Same Type



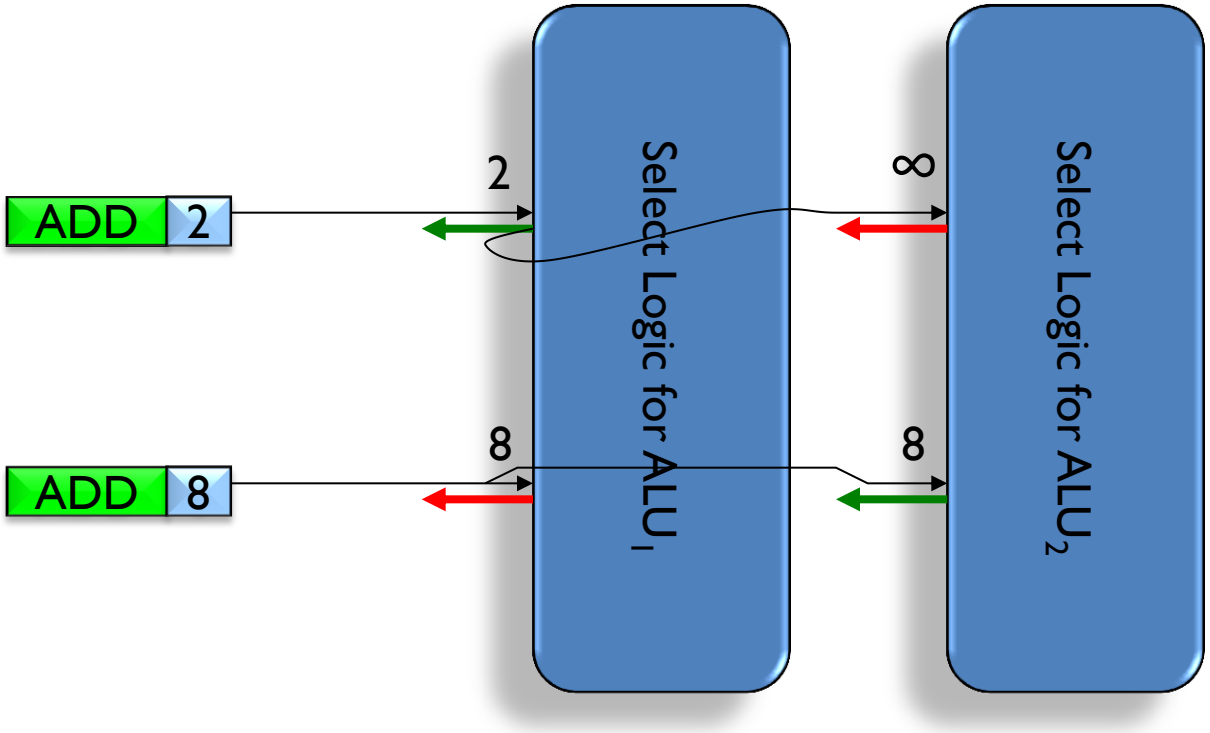
Possible to have multiple popular FUs

Bid to Both?



No! Same inputs → Same outputs

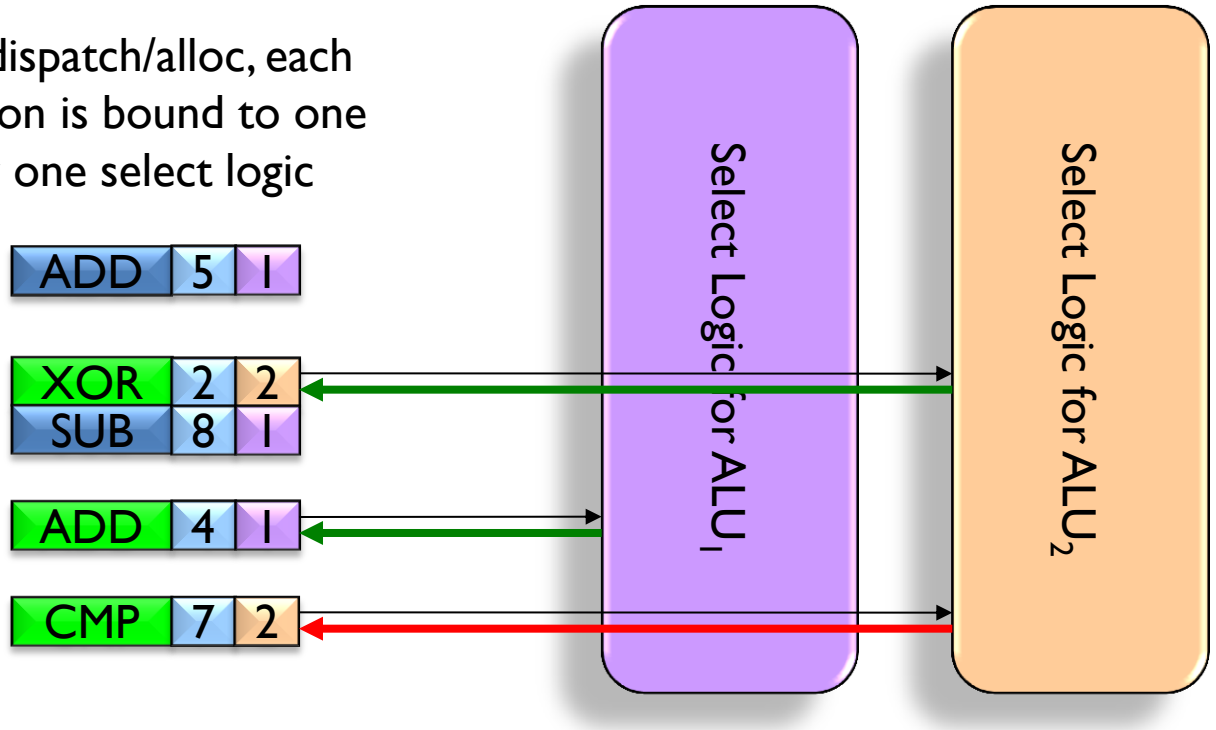
Chain Select Logics



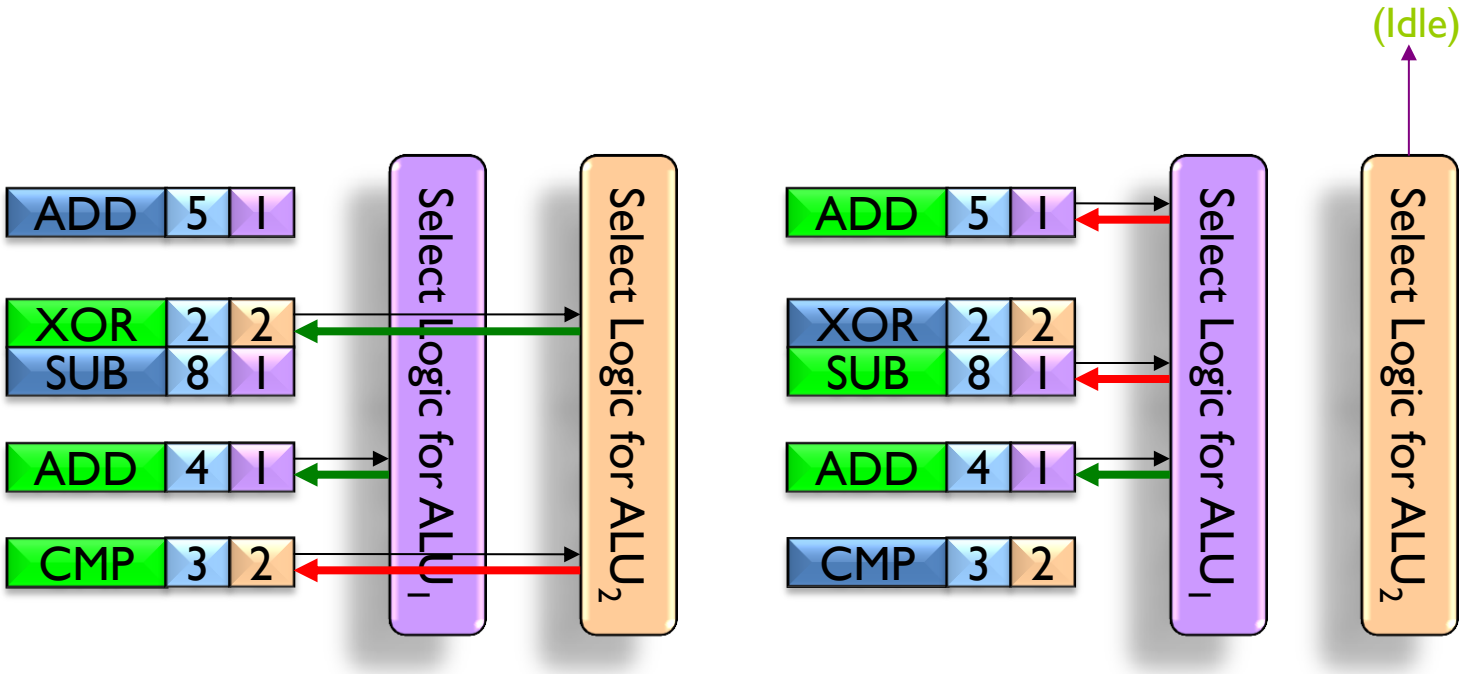
Works, but doubles the select latency

Select Binding (1/2)

During dispatch/alloc, each instruction is bound to one and only one select logic



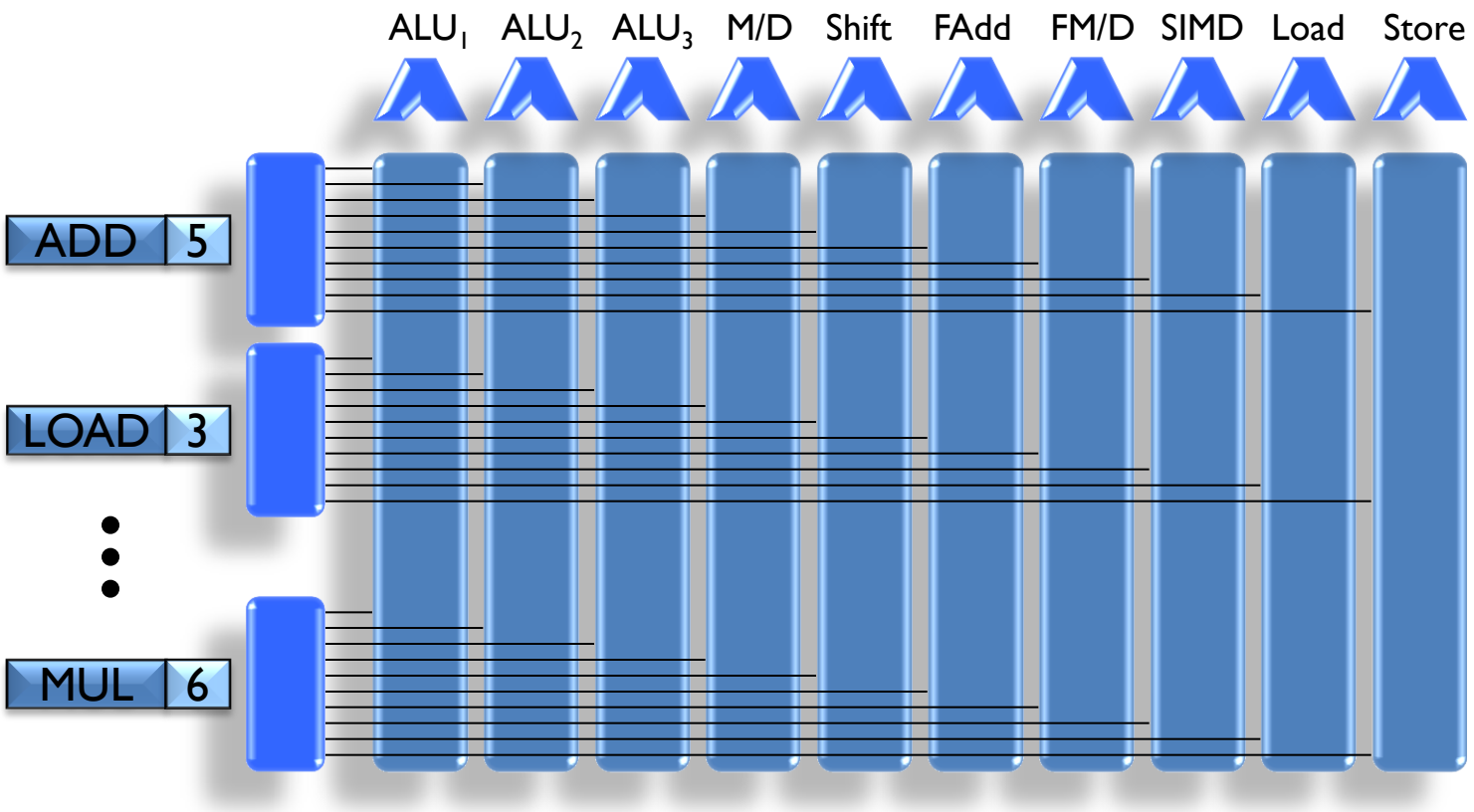
Select Binding (2/2)



Not-Quite-Oldest-First:
Ready insns are aged 2, 3, 4
Issued insns are 2 and 4

Wasted Resources:
3 instructions are ready
Only 1 gets to issue

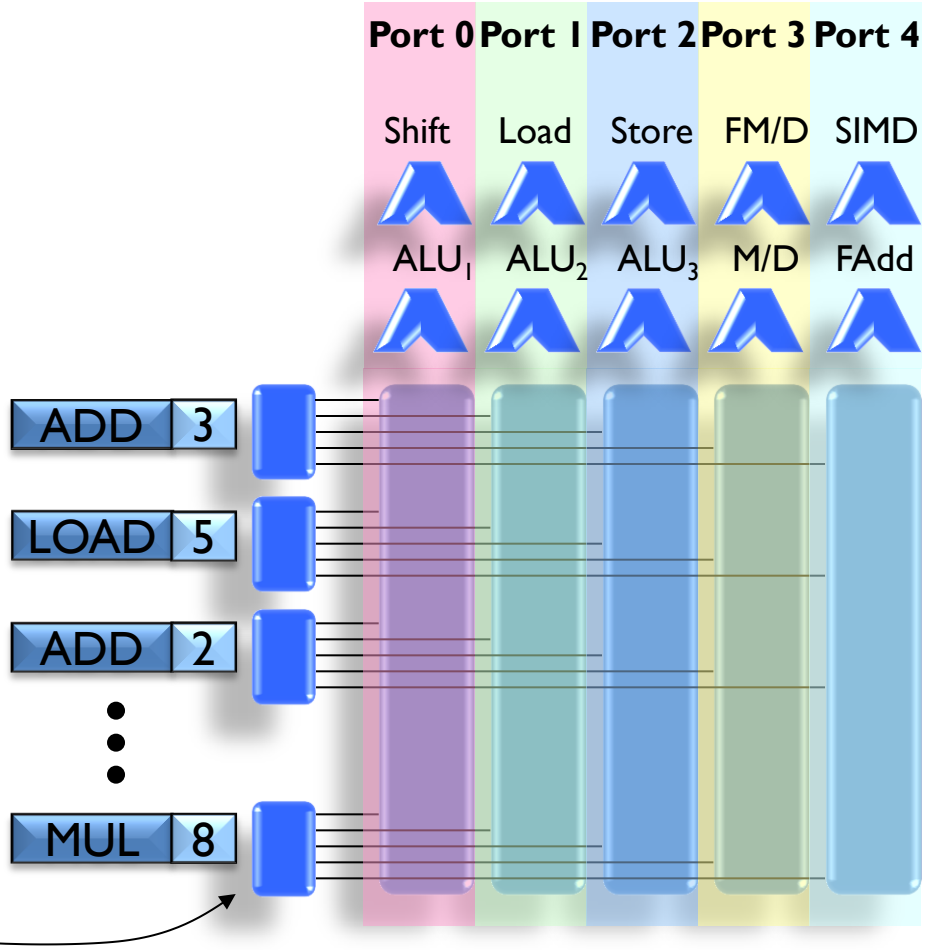
Make N Match Functional Units?



Too big and too slow

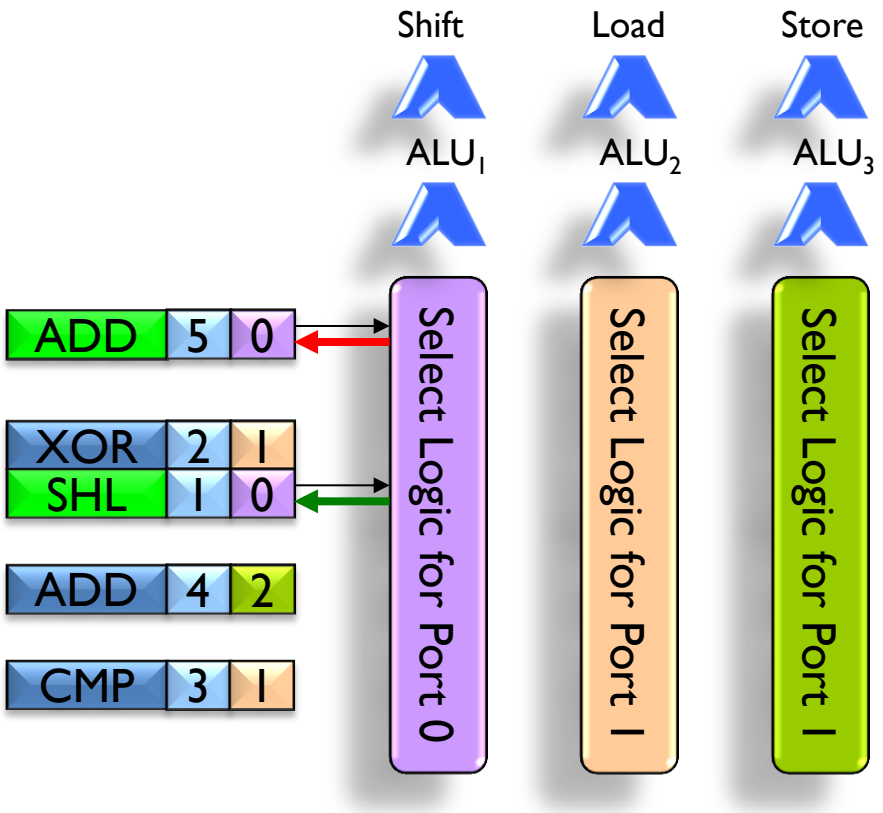
Execution Ports (1/2)

- Divide functional units into P groups
  - Called "*ports*"
- Area only $O(P^2M \log M)$, where $P << F$
- Logic for tracking bids and grants less complex (deals with P sets)



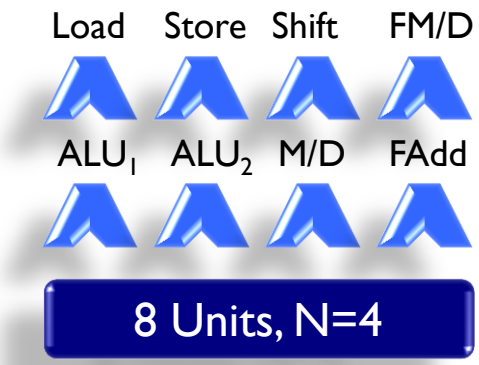
Execution Ports (2/2)

- More wasted resources
- Example
  - SHL issued on Port 0
  - ADD cannot issue
  - 3 ALUs are unused

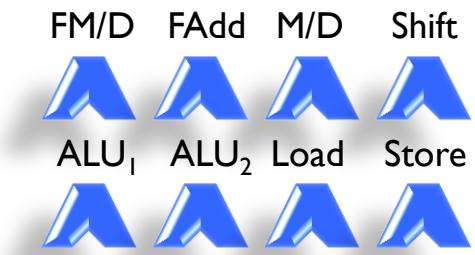


Port Binding

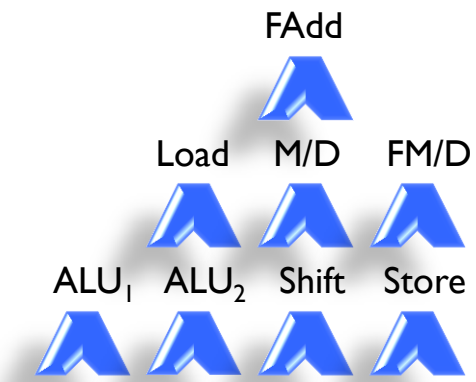
- Assignment of functional units to execution ports
  - Depends on number/type of FUs and issue width



Int/FP Separation
Only Port 3 needs
to access FP RF
and support 64/80 bits



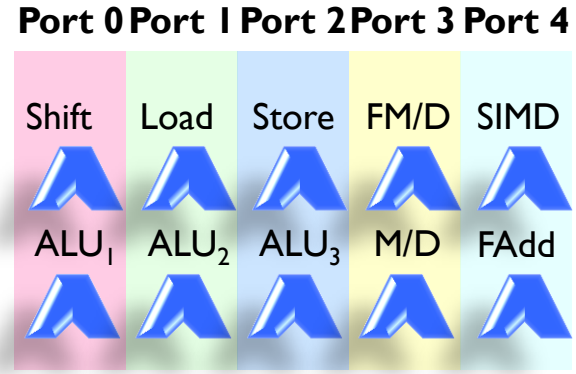
Even distribution of
Int/FP units, more
likely to keep all
N ports busy



Each port need not have
the same number of FUs;
should be bound based
on frequency of usage

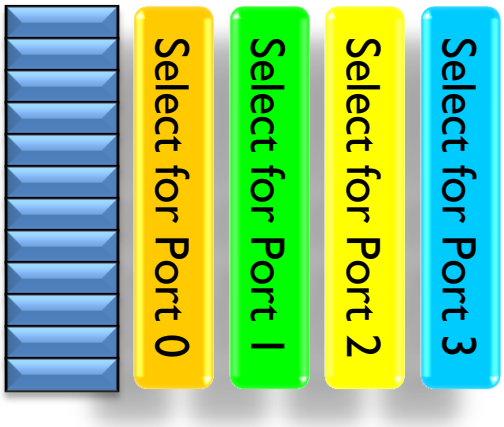
Port Assignment

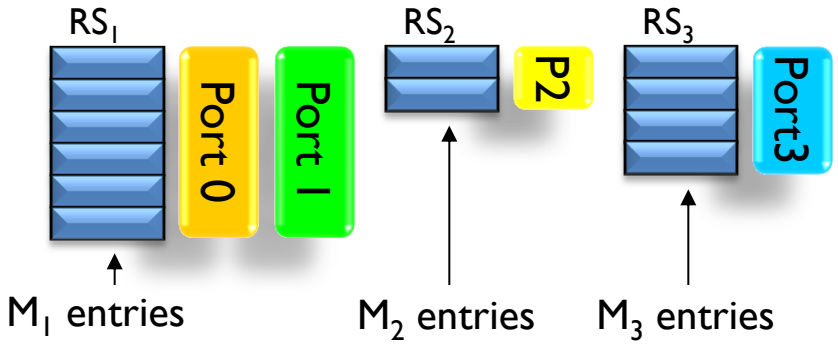
- Insns. get port assignment at dispatch
- For unique resources
  - Assign to the only viable port
  - Ex. Store must be assigned to Port 1
- For non-unique resources
  - Must make intelligent decision
  - Ex. ADD can go to any of Ports 0, 1 or 2
- Optimal assignment requires knowing the future
- Possible heuristics
  - random, round-robin, load-balance, dependency-based, …



Decentralized RS (1/4)

- Area and latency depend on number of RS entries
- Decentralize the RS to reduce effects:

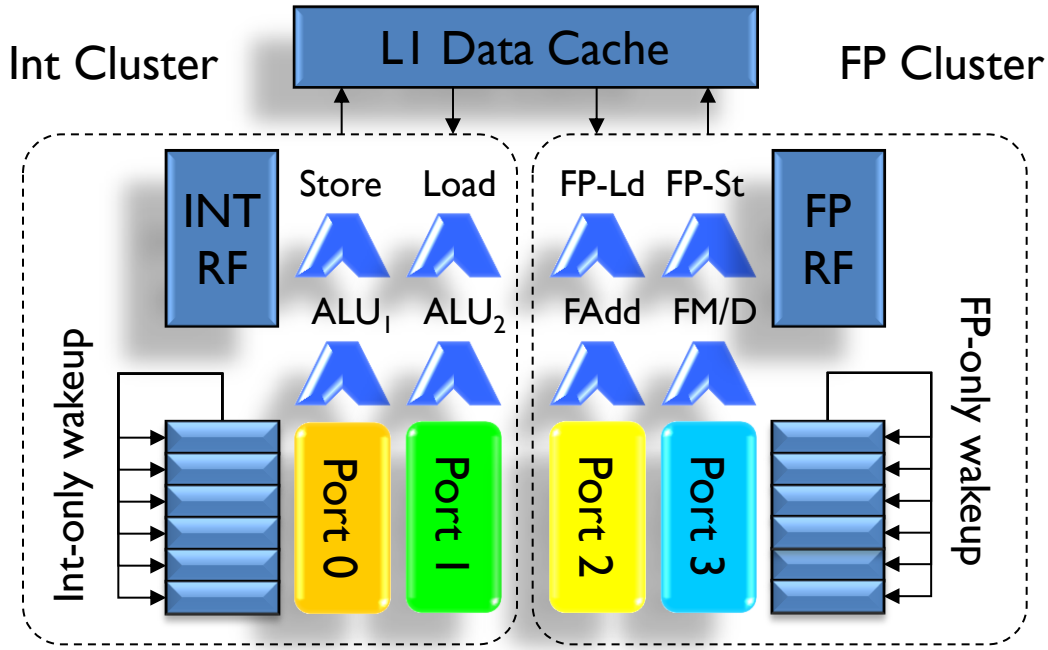




Select logic blocks for RS_i only have gate delay of $O(\log M_i)$

Decentralized RS (2/4)

- Natural split: INT vs. FP

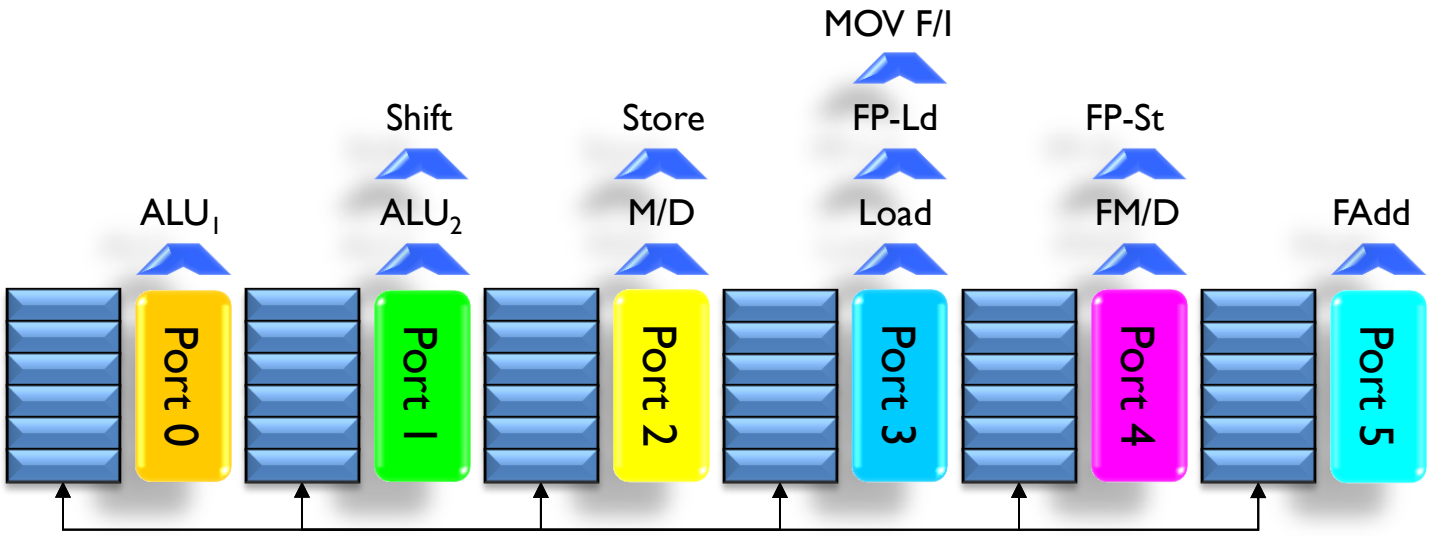


Often implies non-ROB based physical register file:

One "unified" integer PRF, and one "unified" FP PRF, each managed separately with their own free lists

Decentralized RS (3/4)

- Fully generalized decentralized RS



- Over-doing it can make RS and select smaller
  … but tag broadcast may get out of control

Can combine with INT/FP split idea

Decentralized RS (4/4)

- Each RS-cluster is smaller
  - Easier to implement, less area, faster clock speed
- Poor utilization leads to IPC loss
  - Partitioning must match program characteristics
  - Previous example:
    - Integer program with no FP instructions runs on 2/3 of issue width (ports 4 and 5 are unused)