

CS250 VLSI Systems Design

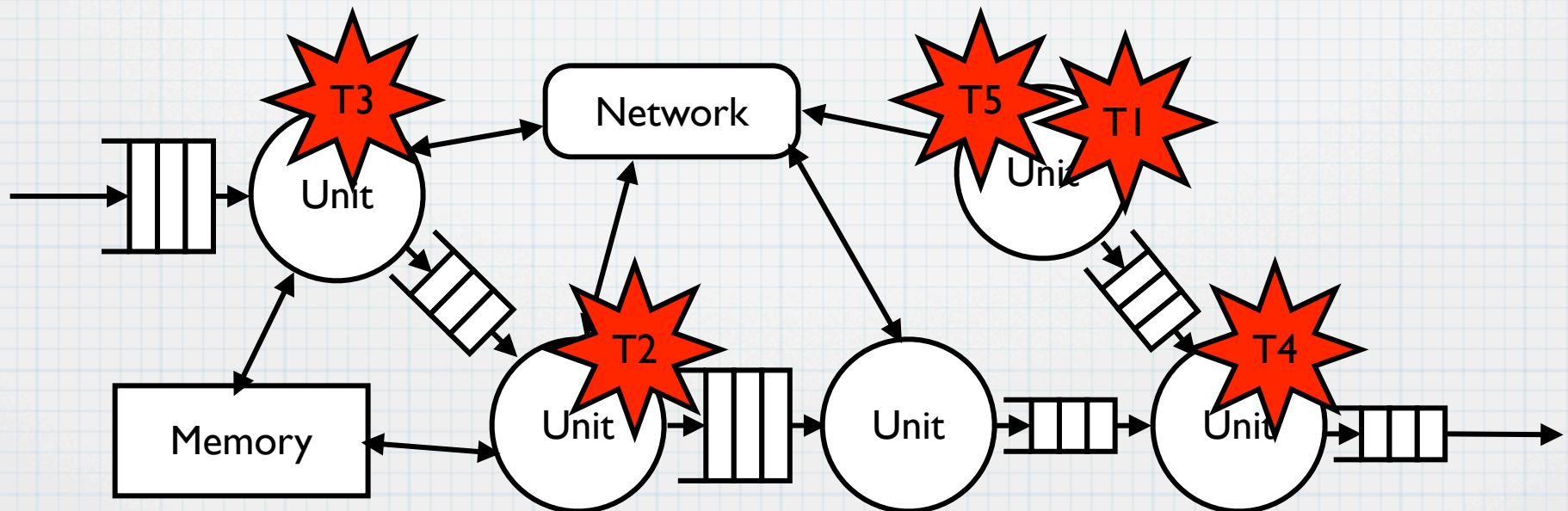
Lecture 10: Patterns for Processing Units and Communication Links

John Wawrzynek, Krste Asanovic,
with
John Lazzaro
and
Brian Zimmer (TA)

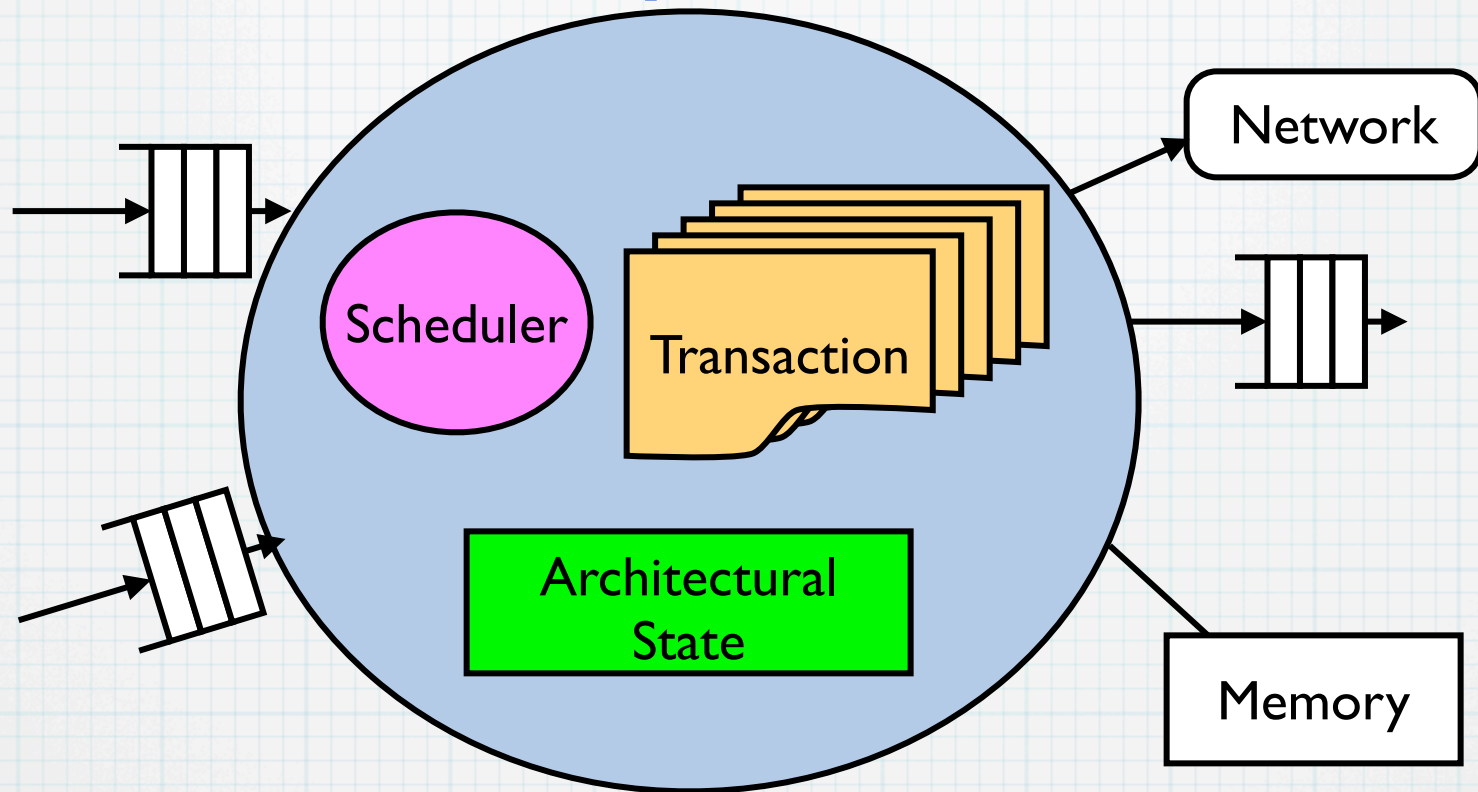
UC Berkeley
Fall 2011

Unit-Transaction Level (UTL)

- A UTL design's functionality is specified as sequences of atomic transactions performed at each unit, affecting only local state and I/O of unit
- i.e., *serializable*: can reach any legal state by single-stepping entire system, one transaction at a time
- High-level UTL spec admits various mappings into RTL with various cycle timings and overlap of transactions' executions



Transactional Specification of Unit

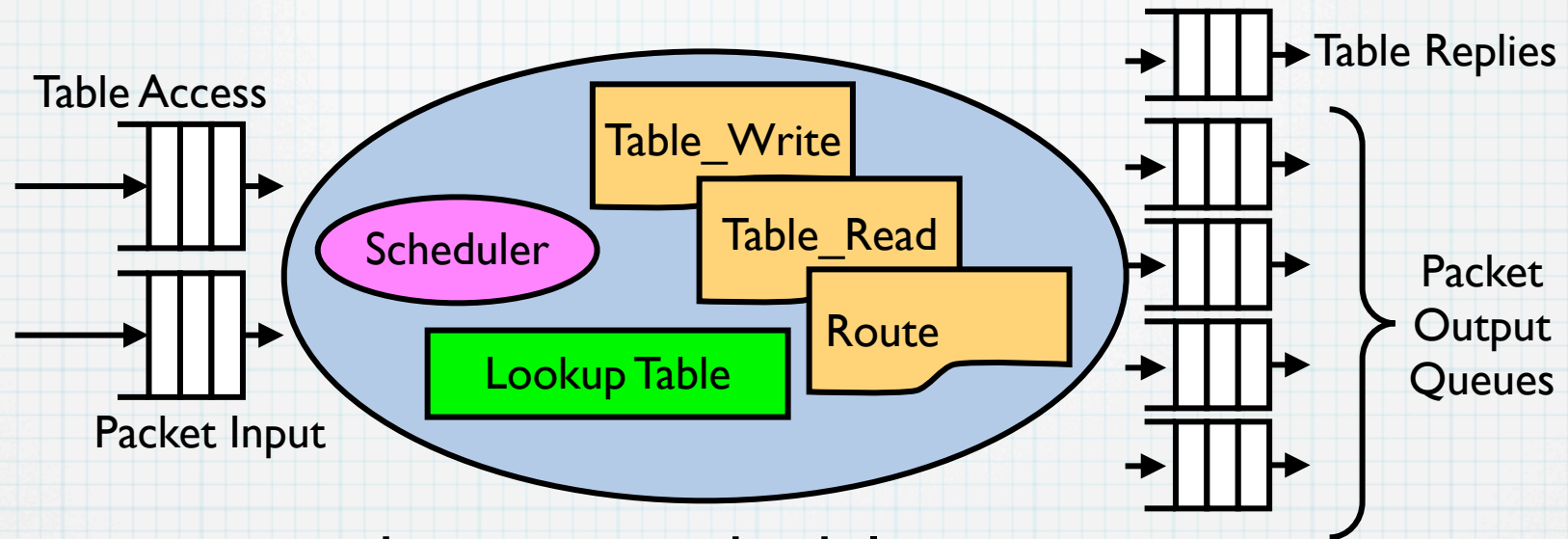


- Each transaction has a combinational guard function defined over local state and state of I/O indicating when it can fire
 - e.g., only fire when head of input queue present and of certain type
- Transaction mutates local state and performs I/O when it fires
- Scheduler is combinational function that picks next ready transaction to fire

Architectural State

- The architectural state of a unit is that which is visible from outside the unit through I/O operations
 - i.e., architectural state is part of the spec
 - (this is the target for “black-box” testing)
- When a unit is refined into RTL, there will usually be additional *microarchitectural* state that is not visible from outside
 - Intra-transaction sequencing logic
 - Pipeline registers
 - Internal caches and/or buffers
 - (this is the target for “white-box” testing)

UTL Example: Route Lookup

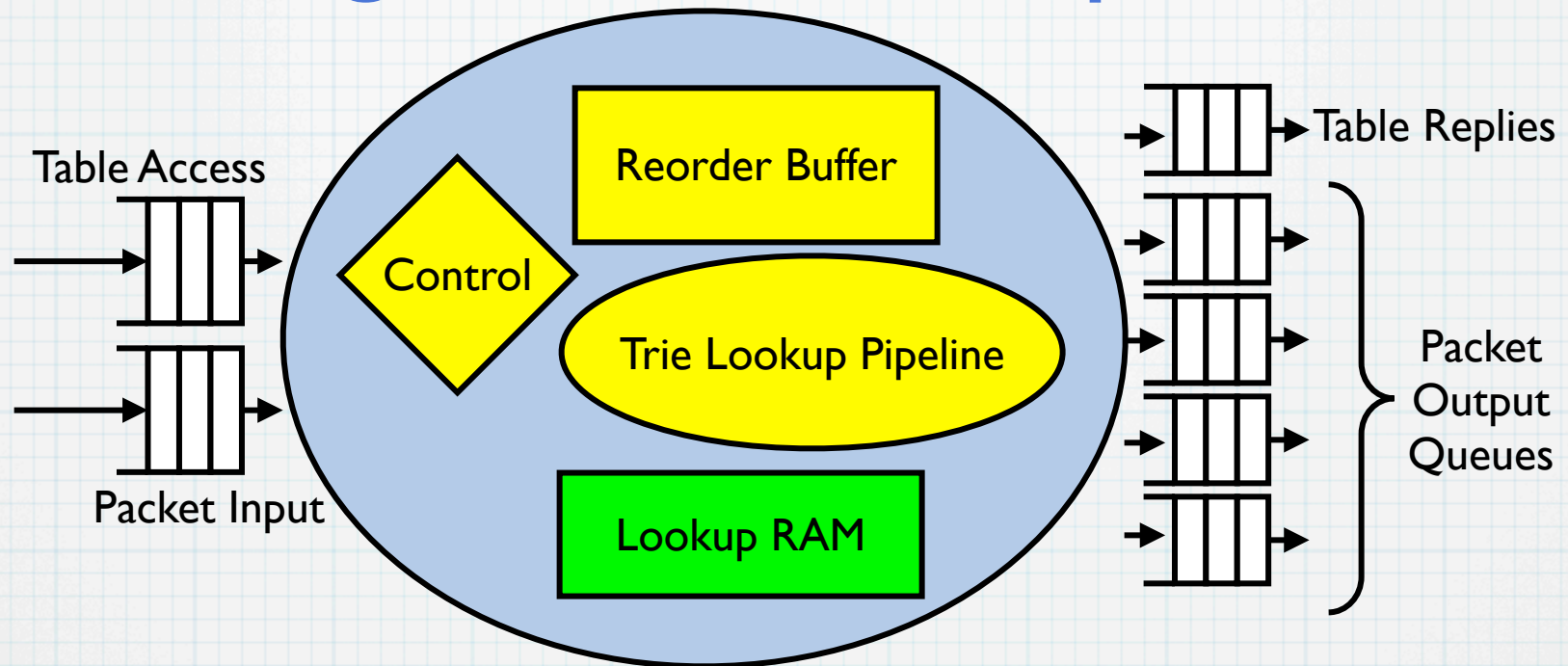


Transactions in decreasing scheduler priority

- **Table_Write** (request on table access queue)
 - Writes a given 12-bit value to a given 12-bit address
- **Table_Read** (request on table access queue)
 - Reads a 12-bit value given a 12-bit address, puts response on table reply queue
- **Route** (request on packet input queue)
 - Looks up header in table and places routed packet on correct output queue

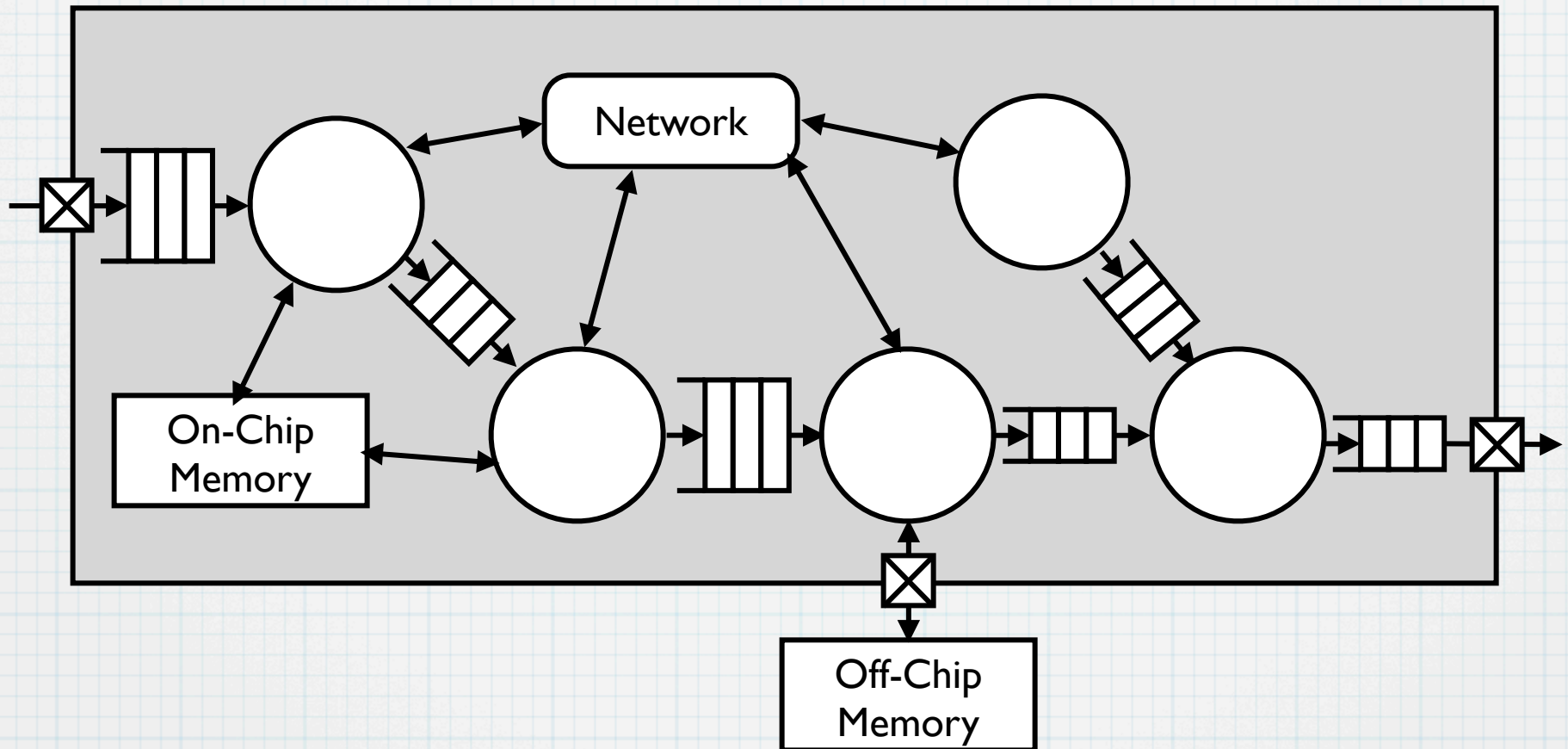
This level of detail is all the information we really need to understand what the unit is supposed to do! Everything else is implementation.

Refining Route Lookup to RTL



- The reorder buffer, the trie lookup pipeline's registers, and any control state are microarchitectural state that should not affect function as viewed from outside
- Implementation must ensure atomicity of UTL transactions:
 - Reorder buffer ensures packets flow through unit in order
 - Must also ensure table write doesn't appear to happen in middle of packet lookup, e.g., wait for pipeline to drain before performing write

System Design Goal: Rate Balancing



- System performance limited by application requirements, on-chip performance, off-chip I/O, or power/energy
- Want to balance throughput of all units (processing, memory, networks) so none too fast or too slow

Rate-Balancing Patterns

To make unit faster, use parallelism

- Unrolling (for processing units)
- Banking (for memories)
- Multiporting (for memories)
- Widen links (for networks)
- *I.e., Use more resources by expanding in space, shrinking in time*

To make unit slower, use time-multiplexing

- Replace dedicated links with a shared bus (for networks)
- Replace dedicated memories with a common memory
- Replace multiport memory with multiple cycles on single port
- Multithread computations onto a common pipeline
- Schedule a dataflow graph onto a single ALU
- *I.e., Use less resources by shrinking in space, expanding in time*

Stateless Stream Unit Unrolling

(Stream is an ordered sequence)

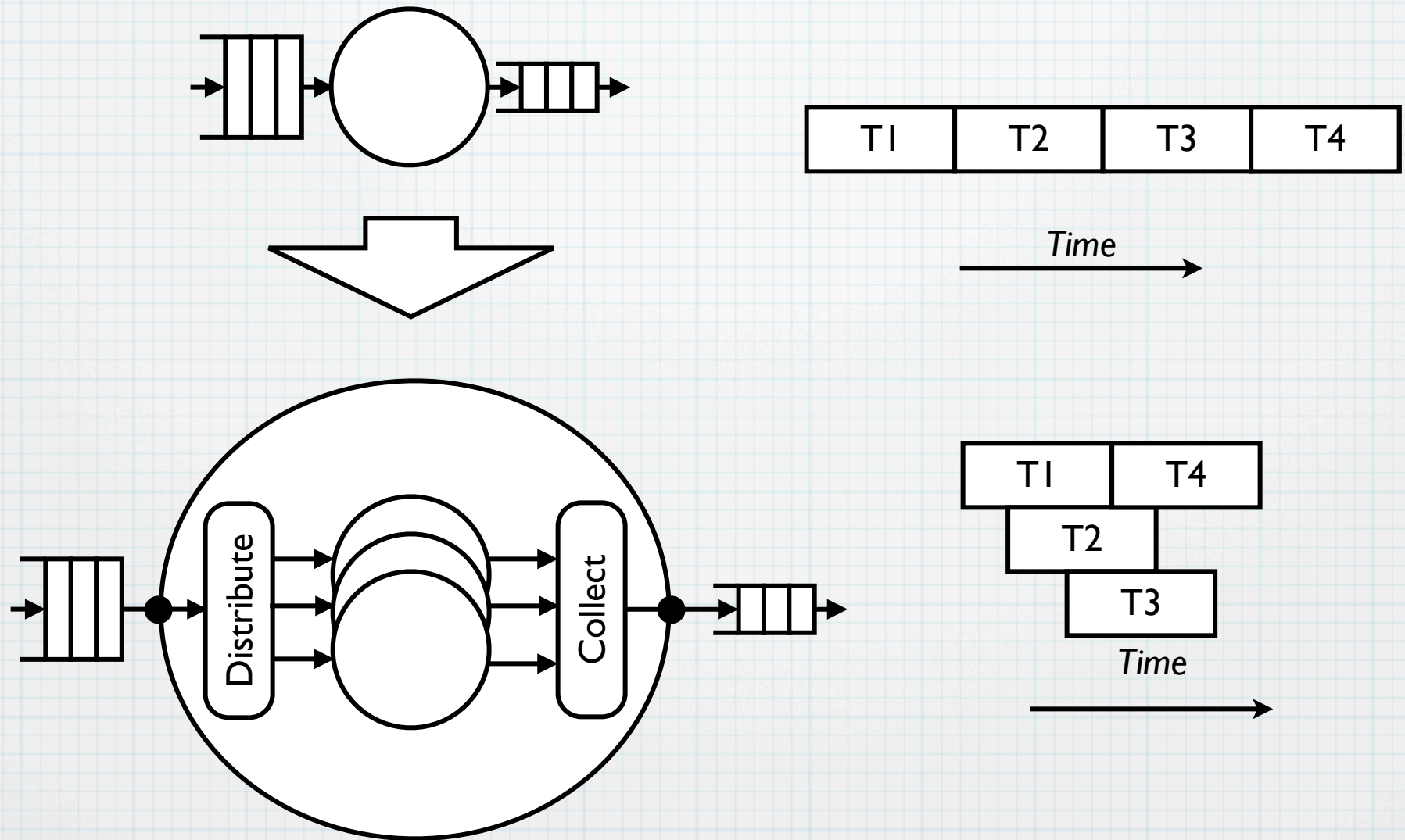
Problem: A stateless unit processing a single input stream of requests has insufficient throughput.

Solution: Replicate the unit and stripe requests across the parallel units. Aggregate the results from the units to form a response stream.

Applicability: Stream unit does not communicate values between independent requests.

Consequences: Requires additional hardware for replicated units, plus networks to route requests and collect responses. Latency and energy for each individual request increases due to additional interconnect cost.

Stateless Stream Unit Unrolling



Variable-Latency Stateless Stream Unit Unrolling

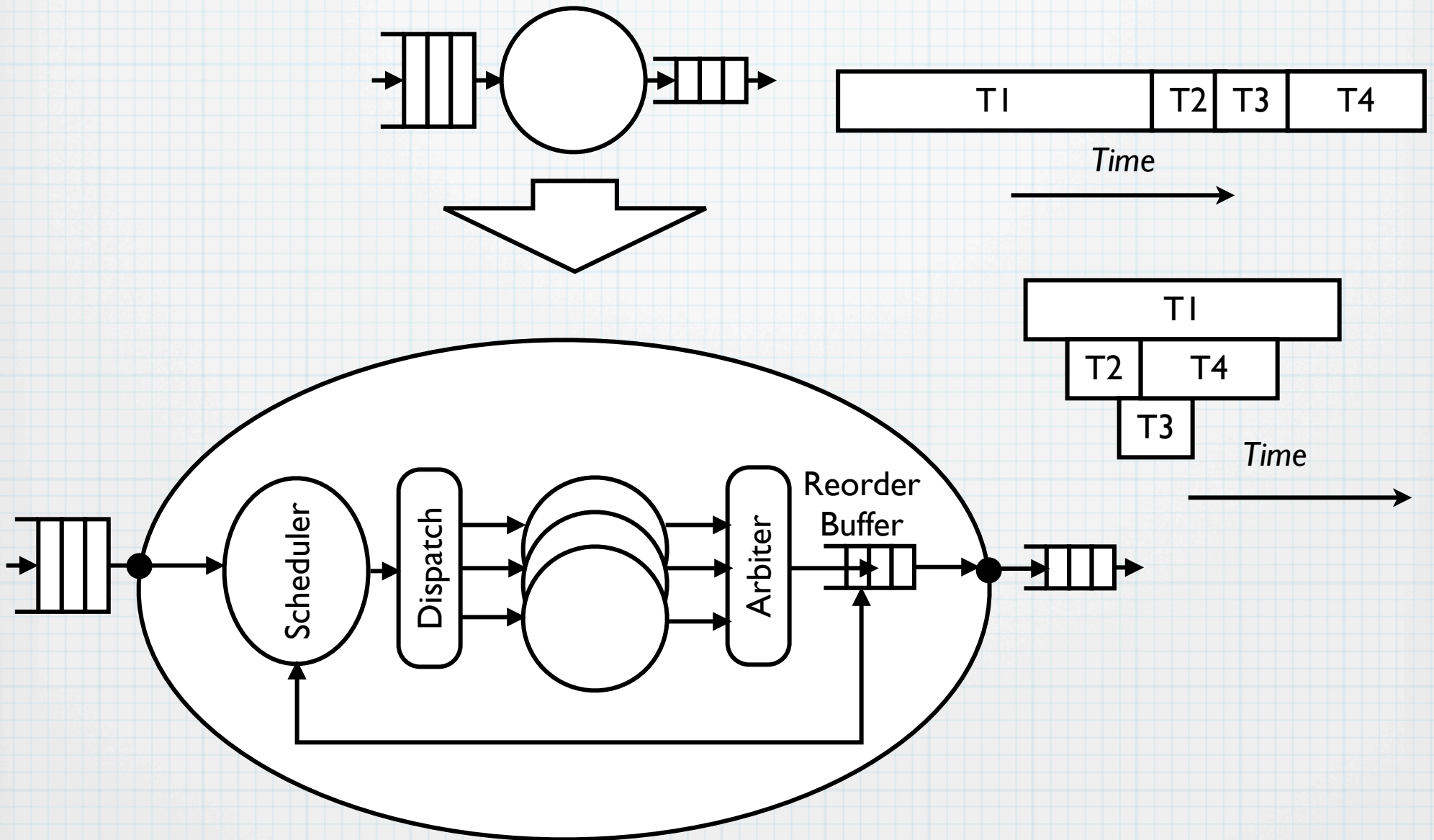
Problem: A stateless stream unit processing a single input stream of requests has insufficient throughput, and each request takes a variable amount of time to process.

Solution: Replicate the unit. Allocate space in output reorder buffer in stream order, then dispatch request to next available unit. Unit writes result to allocated slot in output reorder buffer when completed (possibly out-of-order), but results can only be removed in stream order.

Applicability: Stream unit does not communicate values between independent requests.

Consequences: Additional hardware for replicated units plus added scheduler, buffer, and interconnects. Need scheduler to find next free unit and possibly an arbiter for reorder buffer write ports. Latency and energy for each individual request increases due to additional buffers and interconnect.

Variable-Latency Stateless Stream Unit Unrolling



Time Multiplexing

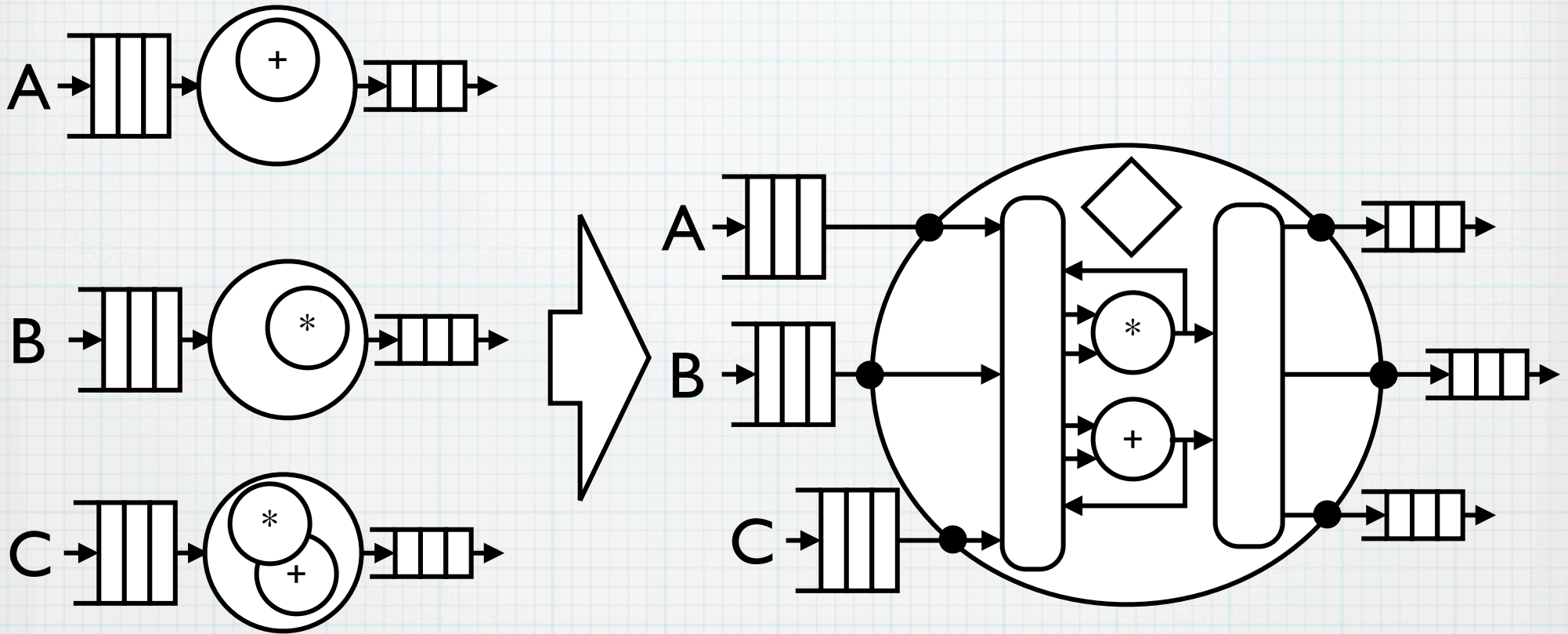
Problem: Too much hardware used by several units processing independent transactions.

Solution: Provide only a single unit and time-multiplex hardware within unit to process independent transactions.

Applicability: Original units have similar functionality and required throughput is low.

Consequences: Combined unit has to provide superset of functionality of original units. Combined unit has to provide architectural state for all architectural state in original units (microarchitectural state, such as pipeline registers, can be shared). Control logic has to arbitrate usage of shared resources among independent transactions, and provide any performance guarantees.

Time Multiplexing



Other Forms of Rate Balancing

- Increase/reduce voltage
 - Trade dynamic power for performance
- Increase/reduce L , V_t , T_{ox} , etc.
 - Trade transistor leakage power for performance

Processing Unit Design Patterns

Control+Datapath

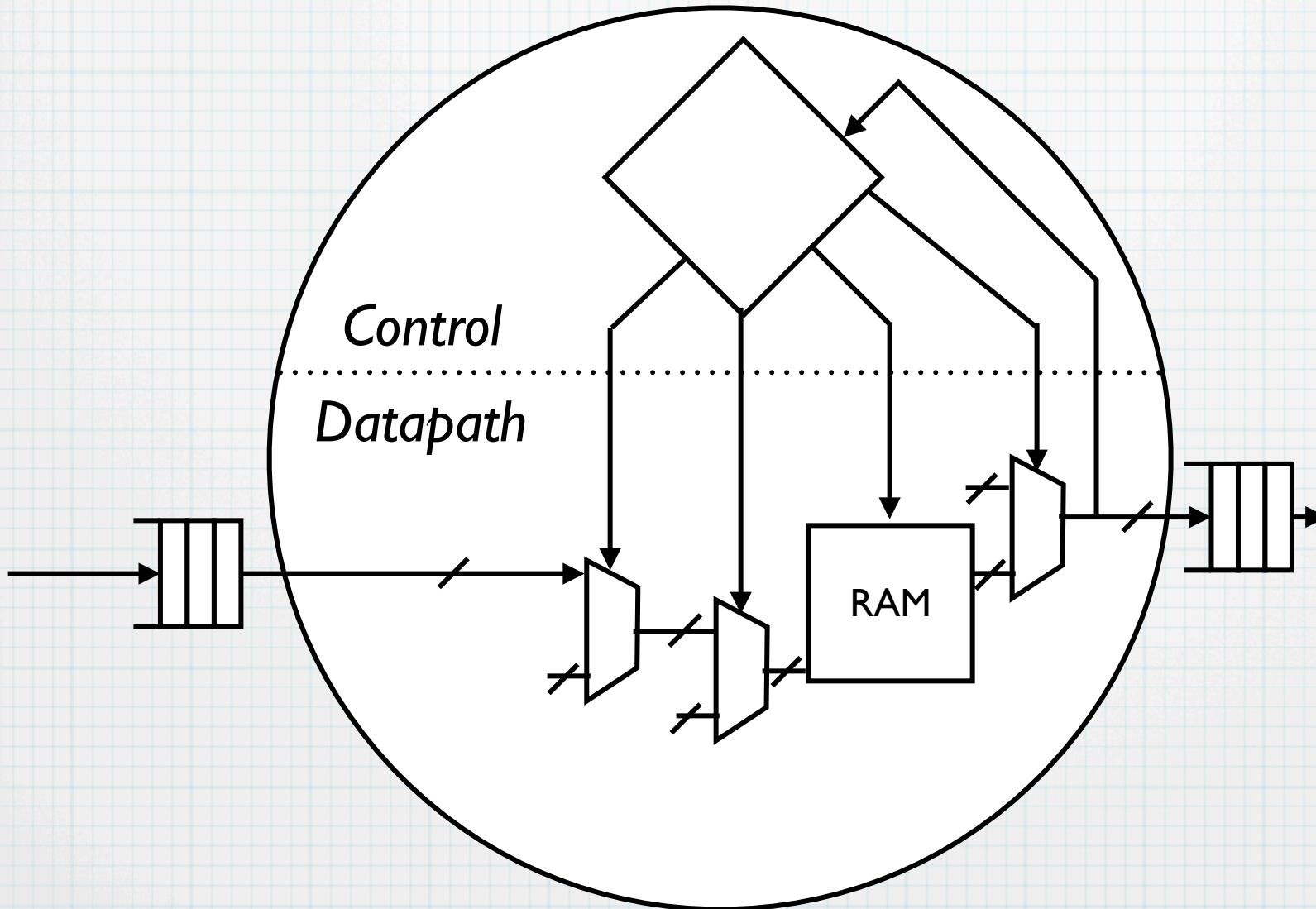
Problem: Arithmetic operations within transaction require large functional units and memories connected by wide buses. Sequencing of operations within transaction is complex.

Solution: Split processing unit into 1) *datapath*, which contains functional units, data memories, and their interconnect, and 2) *control*, which contains all sequencing logic.

Applicability: Where there is a clear divide between control logic and data processing logic, with relatively few signals crossing this divide, and mostly from control to datapath not vice versa.

Consequences: Most design errors are confined to the control portion. Same datapath design can perform many different transactions by changing control sequencing. Paths between control and datapath, particularly from datapath back to control, are often timing critical.

Control+Datapath



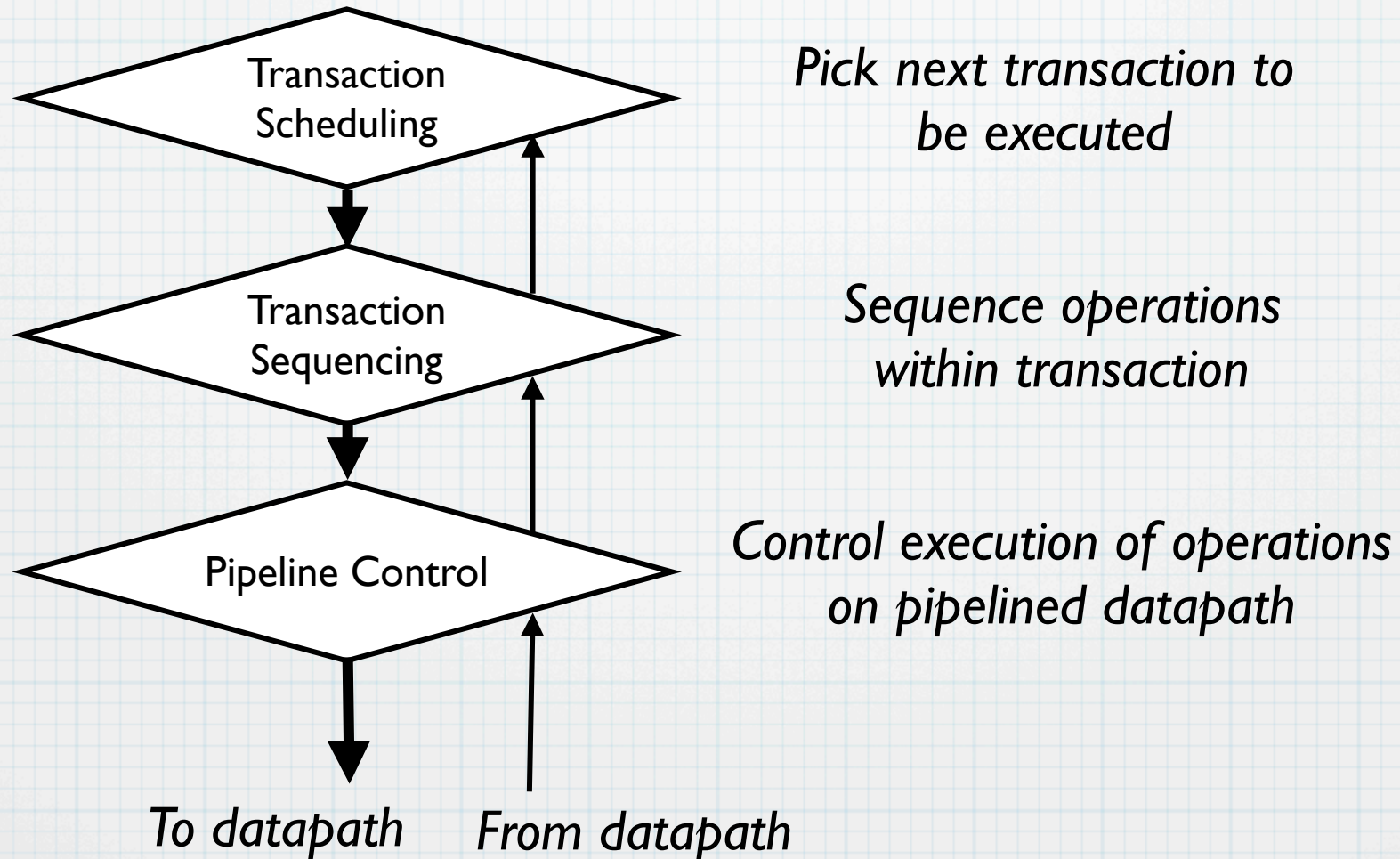
Controller Patterns

For synchronous control of local datapath

- **State Machine Controller**
 - control lines generated by state machine
- **Microcoded Controller**
 - single-cycle datapath, control lines in ROM/RAM
- **In-Order Pipeline Controller**
 - control pipelined datapath, dynamic interaction between stages
- **Out-of-Order Pipeline Controller**
 - operations within a control stream might be reordered internally
- **Threaded Pipeline Controller**
 - multiple control streams, one execution pipeline

Control Decomposition

Can divide control functions into three categories:



State Machine Controller

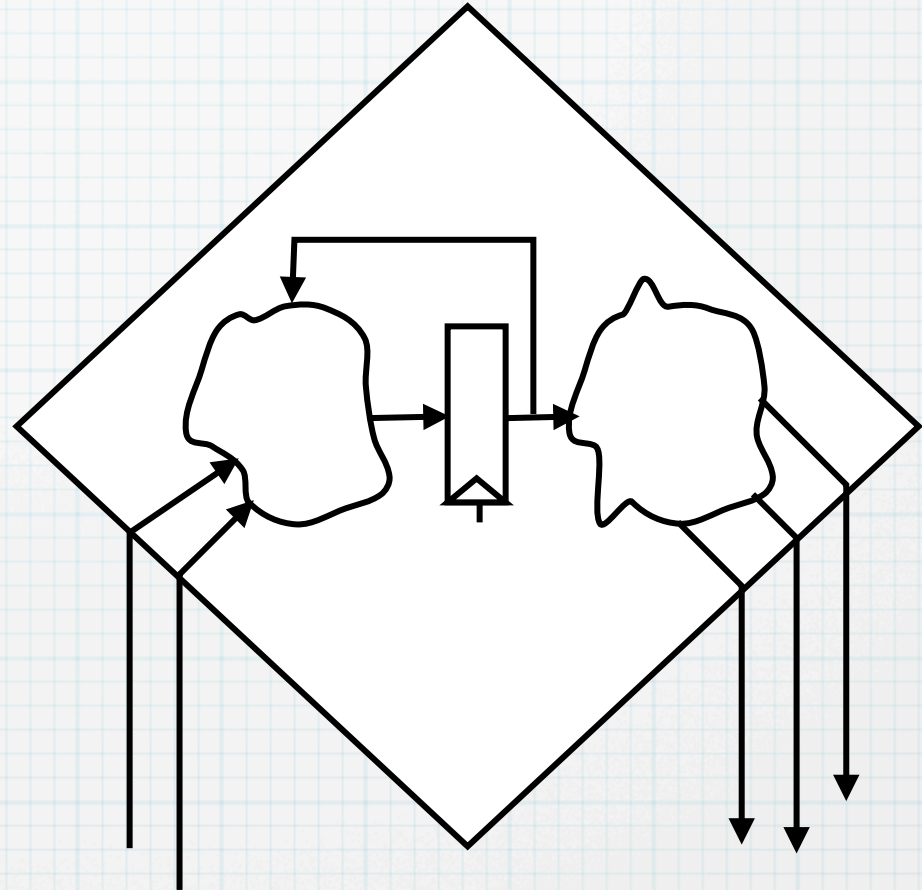
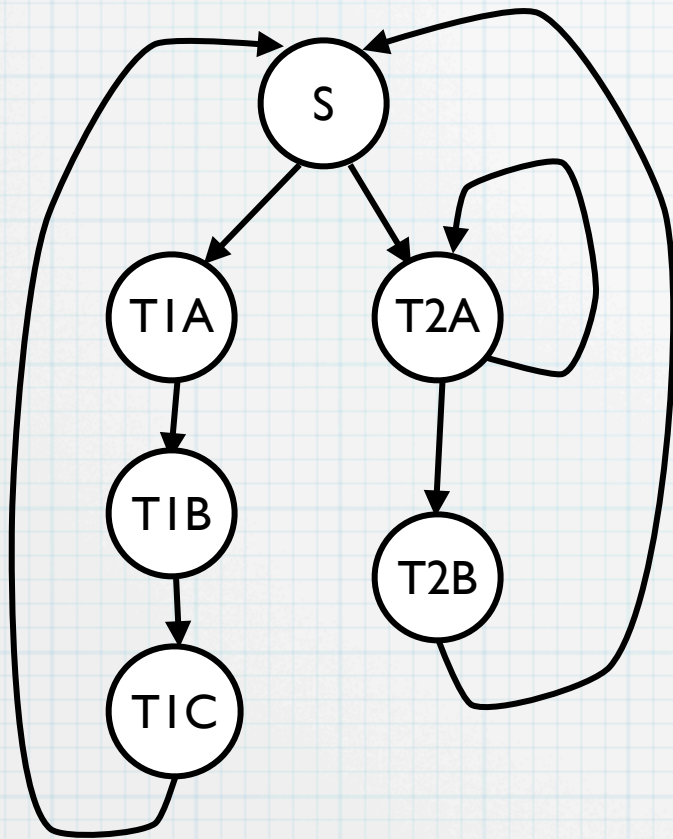
Problem: Control for a simple unit that performs a single transaction at a time.

Solution: Construct state machine with a common initial state to select next transaction, and a separate path for each transaction to sequence operations for that transaction.

Applicability: Where datapath is not highly pipelined and where unit only executes one non-overlapping transaction at a time. Combinational control logic can expand dramatically as number of states increases, so limited to less pipelined and less concurrent units.

Consequences: State machine can be more compact and faster than a microcode controller for small state machines. Changes in unit functionality can cause large changes in size/speed of state machine.

State Machine Controller



Microcoded Controller

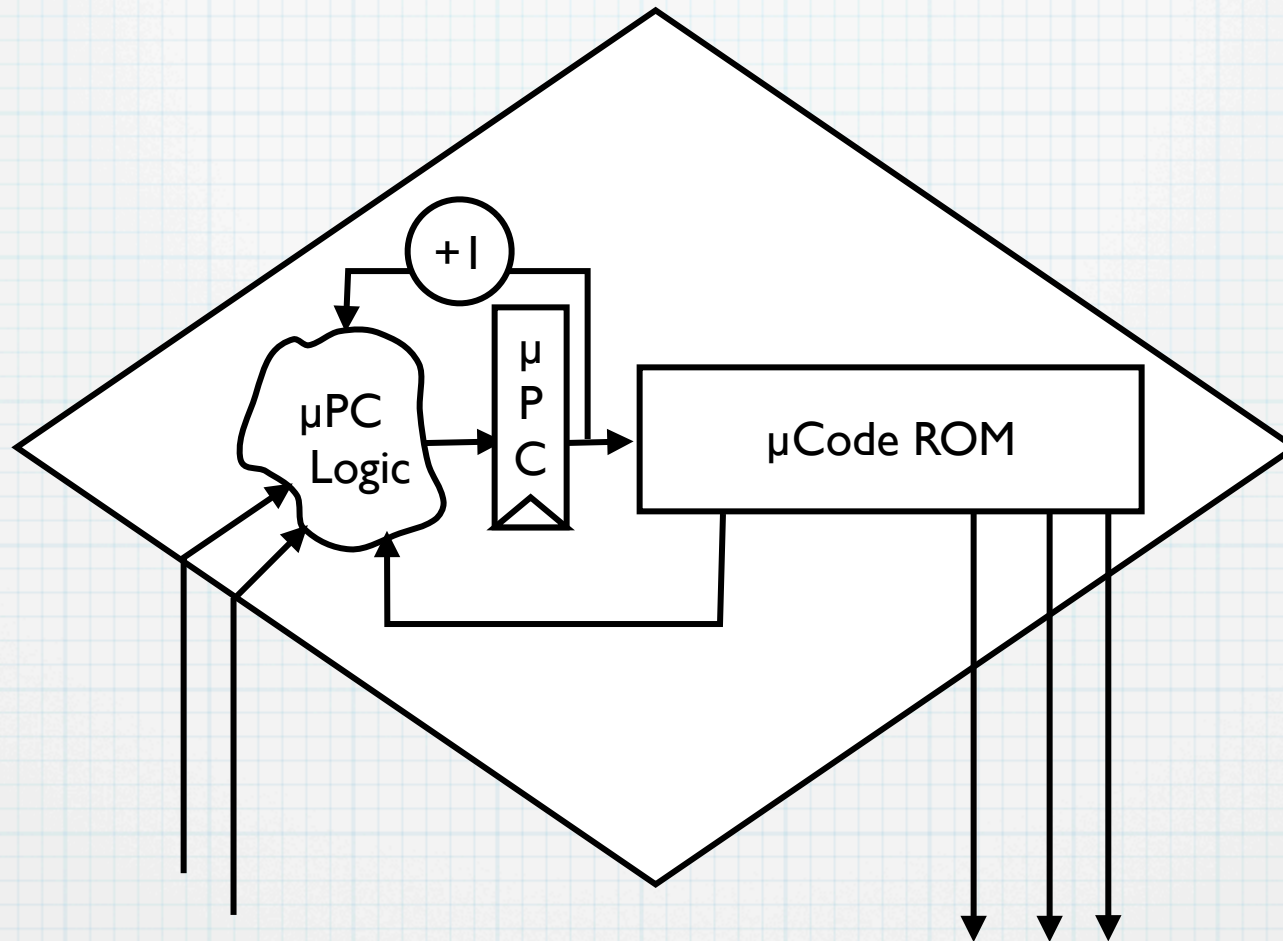
Problem: Control for a complex unit that performs a single transaction at a time.

Solution: Encode control lines in a ROM structure with a small state machine to sequence through locations in ROM. Microcode dispatch function selects next transaction to execute, and each transaction executed by sequence in microcode ROM. Can also use RAM structure to allow post-fabrication modifications to control.

Applicability: Where unit only executes one non-overlapping transaction at a time, but where control is complex. Particularly useful in technology where ROM bits are significantly cheaper than logic gates.

Consequences: Microcode easily modified to make changes in unit functionality. Unit cycle time can be limited by critical path from ROM readout to ROM address input (can use pipelined microcode engine to speed throughput inbetween control hazards).

Microcoded Controller



In-Order Pipeline Controller

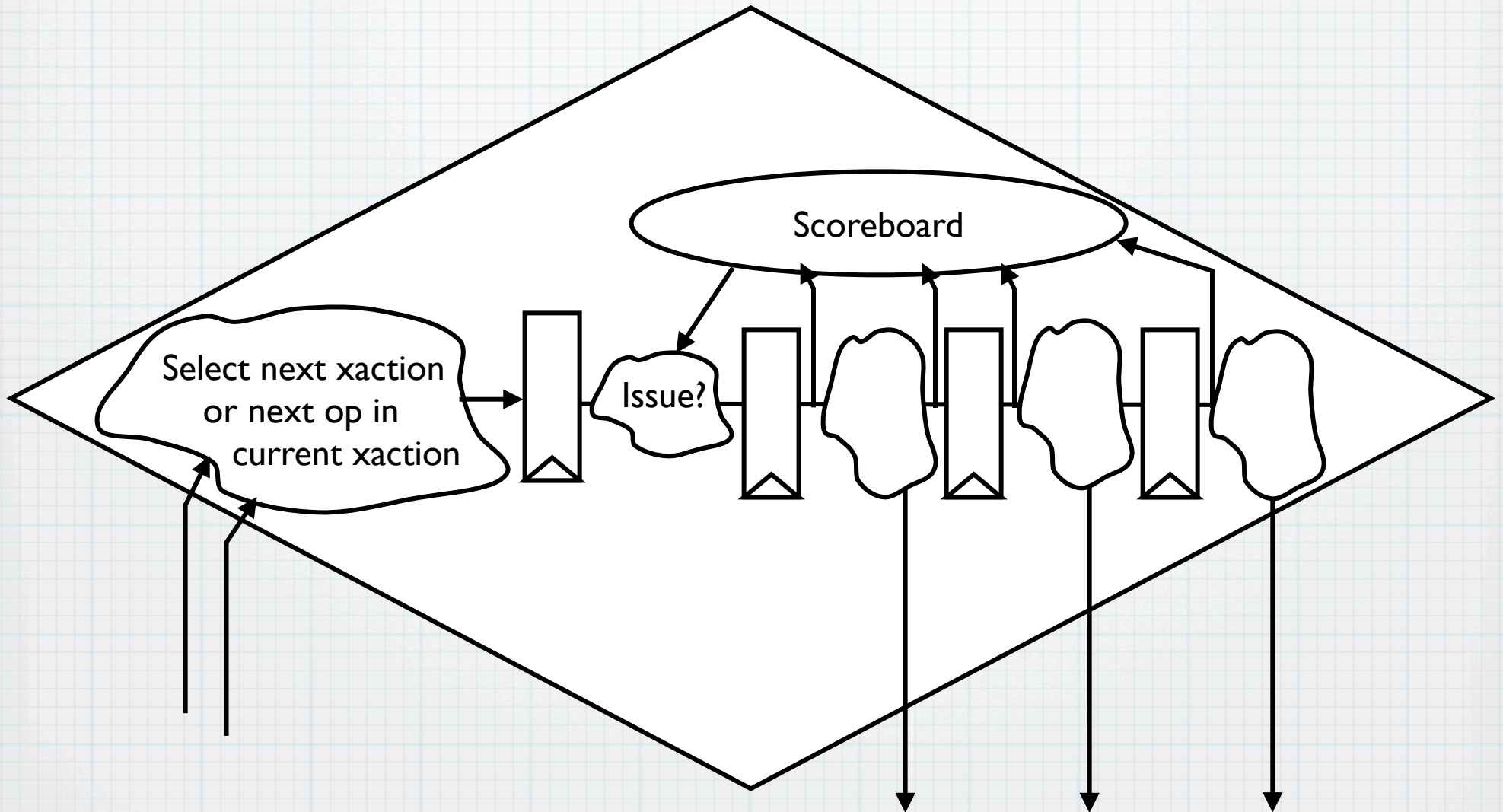
Problem: Control for a complex pipelined unit that can overlap execution of multiple transactions, and multiple operations within one transaction.

Solution: Generate control signals for each stage of pipeline using control state pipelined along with data state. Use dynamic *scoreboard* (part of which may be the pipelined control state) to track operations in flight in pipeline. Next operation can only enter pipeline when scoreboard indicates this would not create a pipelining hazard (structural, data, or control).

Applicability: Where unit's datapath is pipelined and either sequence of transactions or sequence of operations within a transaction is dynamically determined by input data. Where in-order processing is required, or sufficient for performance goals.

Consequences: The datapath design mandates the hazards generated by an executing operation, and can cause large growth in scoreboard complexity and reduction in performance unless hazards on common sequences are avoided.

In-Order Pipeline Controller



Out-of-Order Processing Unit

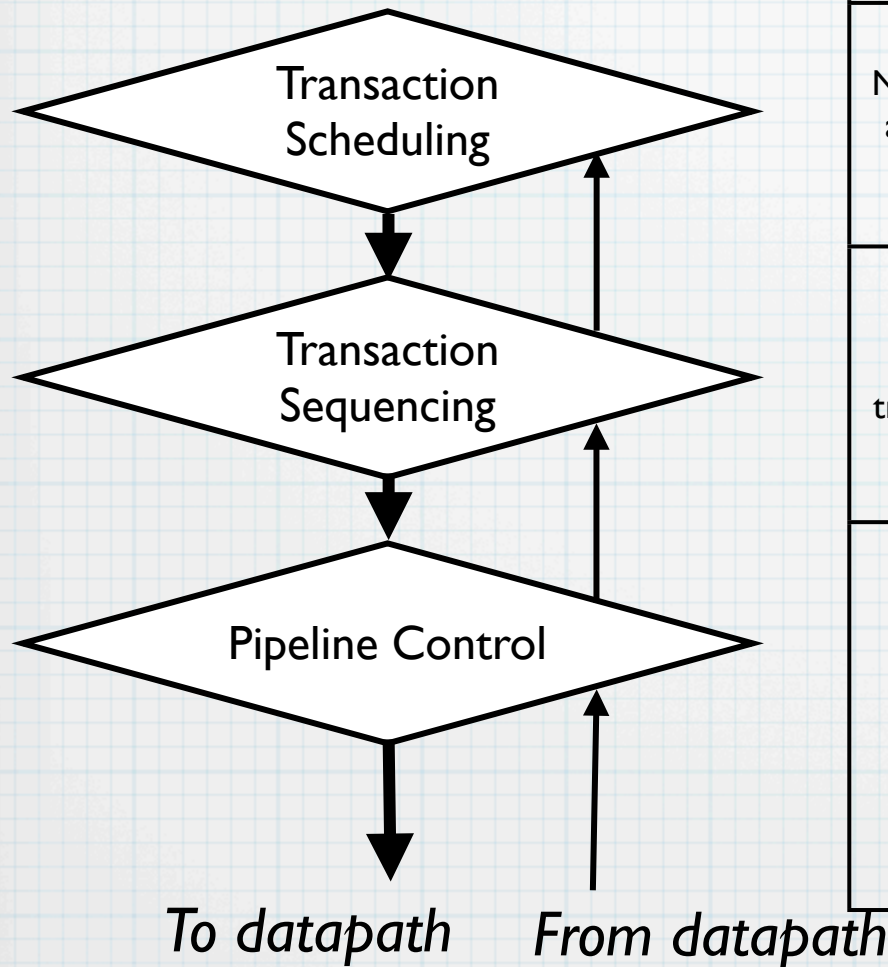
- When in-order gives insufficient throughput, buffer operations and issue out-of-order with respect to hazards.

Threaded Processing Unit

- Multiplex multiple transaction streams onto single hardware unit.
- One specific implementation of time-multiplexing.

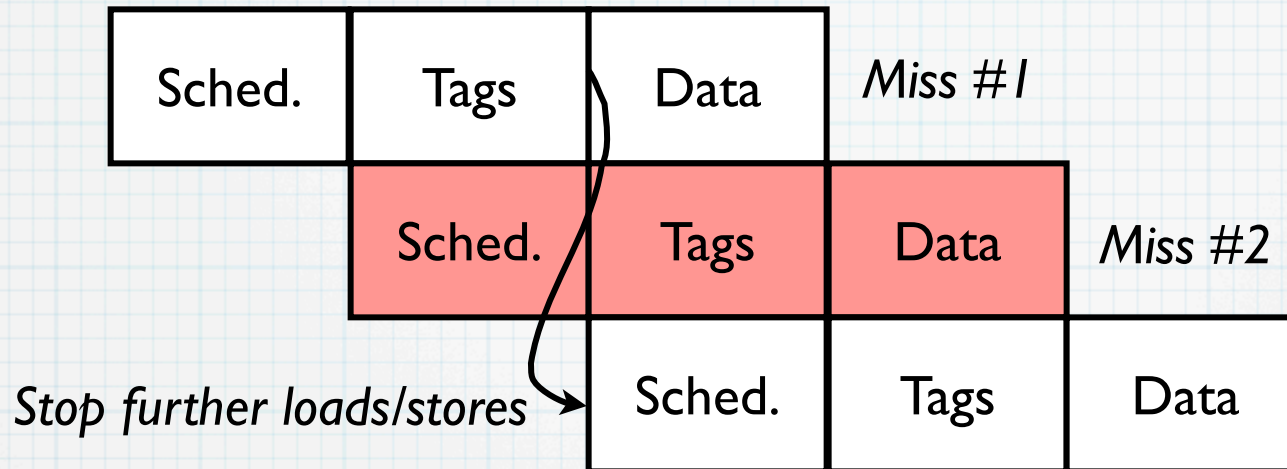
Taxonomy of Control Strategies

Increasing levels of control complexity build on each other.



SM	μ Code	In-Order	OoO	Threaded
Next state after idle state	Dispatch on transaction state to μ code address	Initialize current transaction state	Initialize current transaction state	Interleave transactions from multiple units
State transitions	Step through μ code	Sequence through transaction states (either FSM or μ Code)	same as In-Order	Same as In-Order, except multiple simultaneous actions expanded
N/A	N/A	Control state pipelined along with data. Scoreboard controls issue of next in-order operation	Control state pipelined along with data. Issue buffer executes operation out-of-order	Control state pipelined along with data. Issue next operation from ready action.

Skid Buffering



- Consider non-blocking cache implemented as a 3-stage pipeline: (scheduler, tag access, data access)
- CPU Load/Store not admitted into pipeline unless miss tag, reply queue, and victim buffer available in case of miss
- Hit/miss determined at end of Tags stage, 2nd miss can enter pipeline
- Solutions?
 - Could only allow one load/store every two cycles => low throughput
 - Skid buffering: Add additional victim buffer, miss tags, and replay queues to complete following transaction if miss. Stall scheduler whenever there is not enough space for *two* misses.