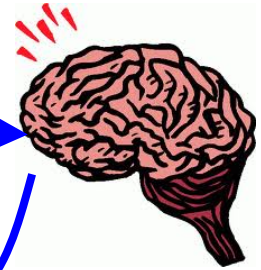
Programmable Switches

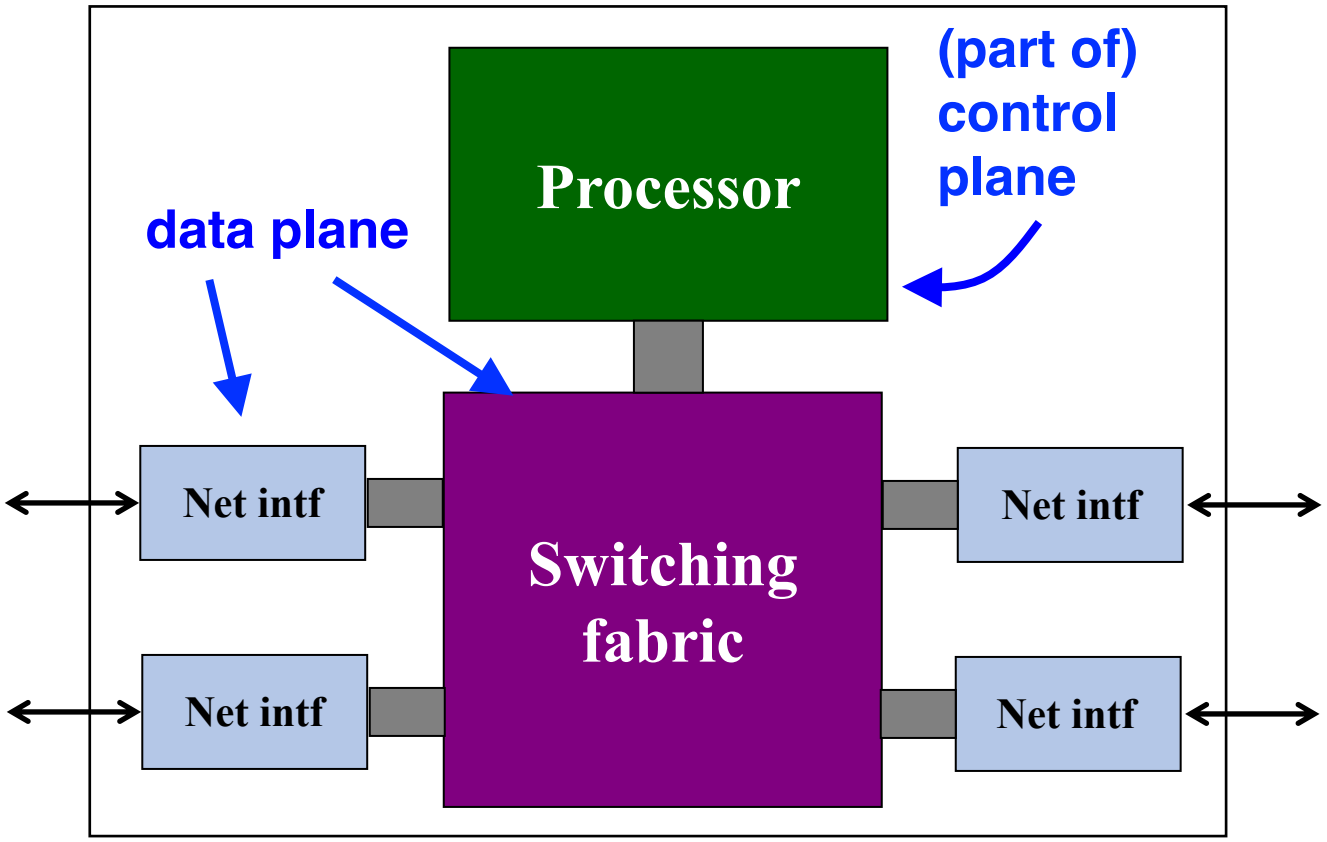
Lecture 13, Computer Networks (198:552)

RUTGERS
UNIVERSITY | NEW BRUNSWICK

SDN router data plane



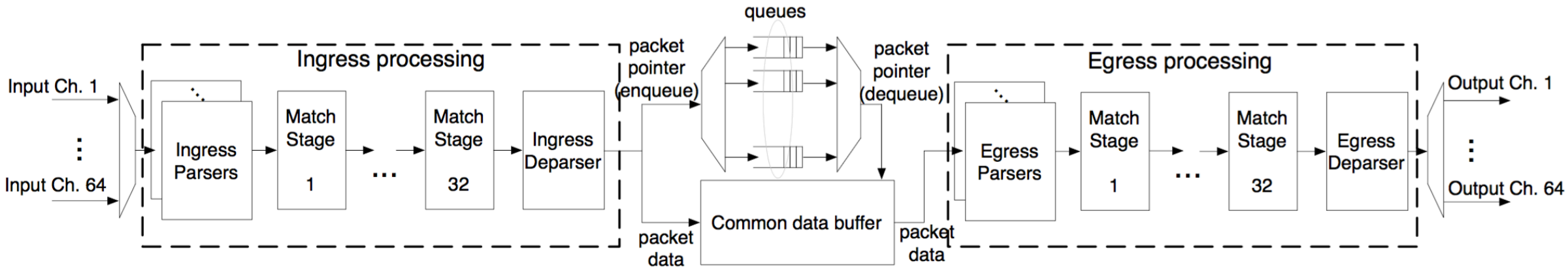
Part of the control plane



- Data plane implements per-packet decisions
  - On behalf of control & management planes
- Forward packets at high speed
- Manage contention for switch/link resources

Life of a packet: RMT architecture

- Many modern switches share similar architecture
  - FlexPipe, Xpliant, Tofino, …
- Pipelined packet processing with a 1 GHz clock



What data plane policies might we need?

- Parsing
  - Ex: Turn raw bits `0x0a000104fe` into IP header 10.0.1.4 and proto 254
- Stateless lookups
  - Ex: Send all packets with protocol 254 through port 5
- Stateful processing
  - Ex: If # packets sent from any IP in 10.0/16 exceeds 500, drop
- Traffic management
  - Ex: Packets from 10.0/16 have high priority unless rate > 10 Kb/s
- Buffer management
  - Ex: restrict all traffic outside of 10.0/16 to 80% of the switch buffer

Programmability

- Allow network designers/operators to specify all of the above
- Needs hardware design and language design
- Software pkt processing could incorporate all of these features
  - However: limited throughput, low port density, high power
- Key Q: *Can we achieve programmability with high performance?*

Programmability: Topics today

1: Packet parsing

2: Flexible stateless processing

3: Flexible stateful processing

4, if we have time: complex policies without perf penalties

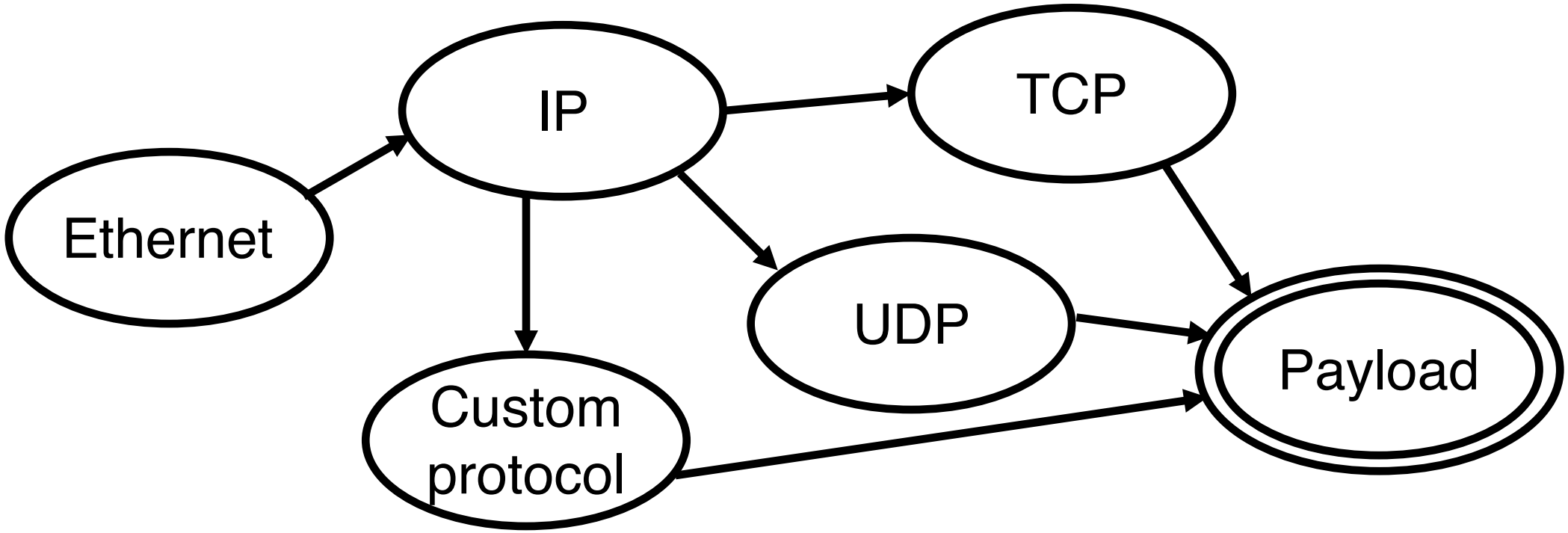
(1) Packet parsing: Need to generalize

- In the beginning, OpenFlow was simple: Match-Action
  - Single rule table on a fixed set of fields (12 fields in OF 1.0)
- Needed new encapsulation formats, different versions of protocols, additional measurement-related headers
- Number of headers ballooned to 41 in OF 1.4 specification!
  - With multiple stages of heterogenous tables

(1) Parsing abstractions

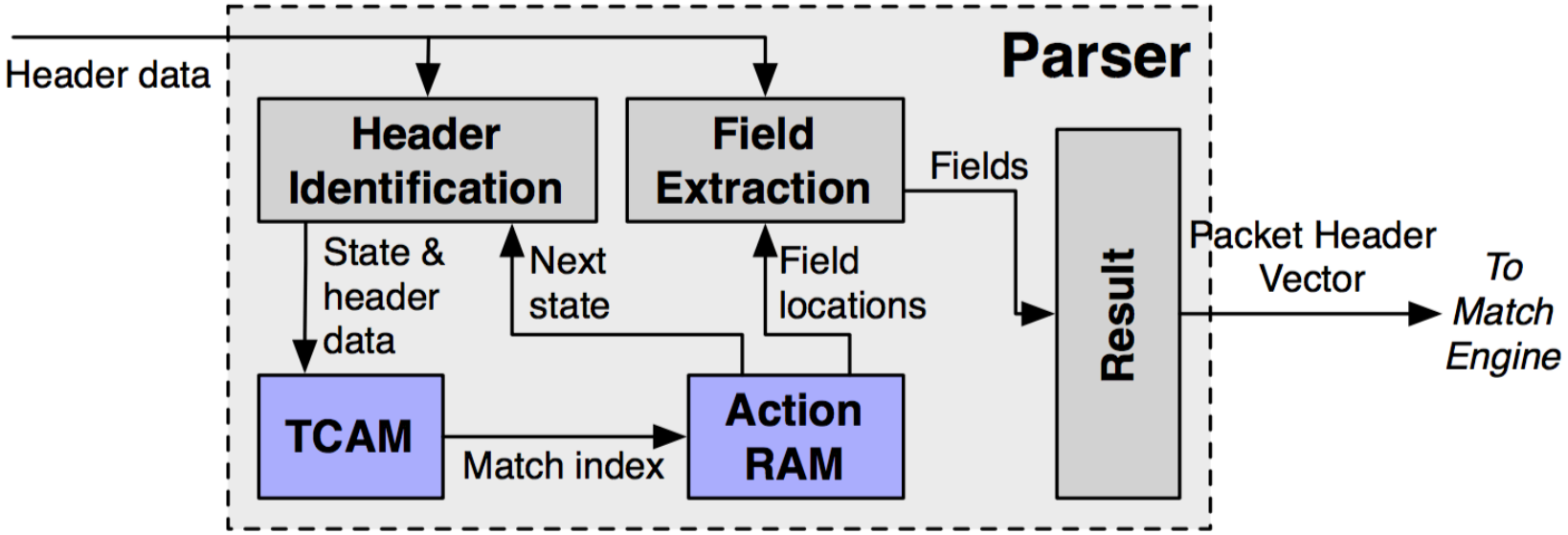
- Goal: can we make transforming bits to headers more flexible?

- A parser *state machine* where each state may emit headers



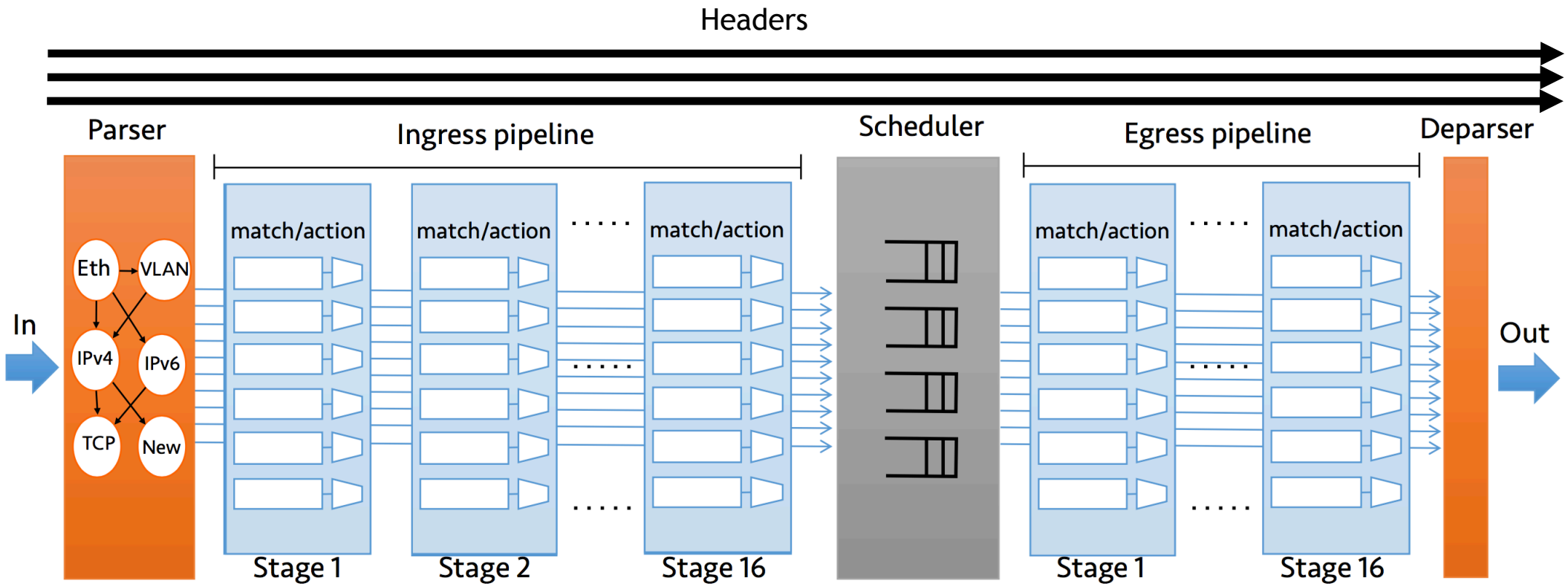
(1) Parsing implementation in hardware

- Use TCAM to store state machine transitions & hdr bit locations
- Extract fields into *packet header vector* in a separate action RAM



(2) How are the parsed headers used?

- Headers carried through the rest of the pipeline
  - To be used in general-purpose match-action tables



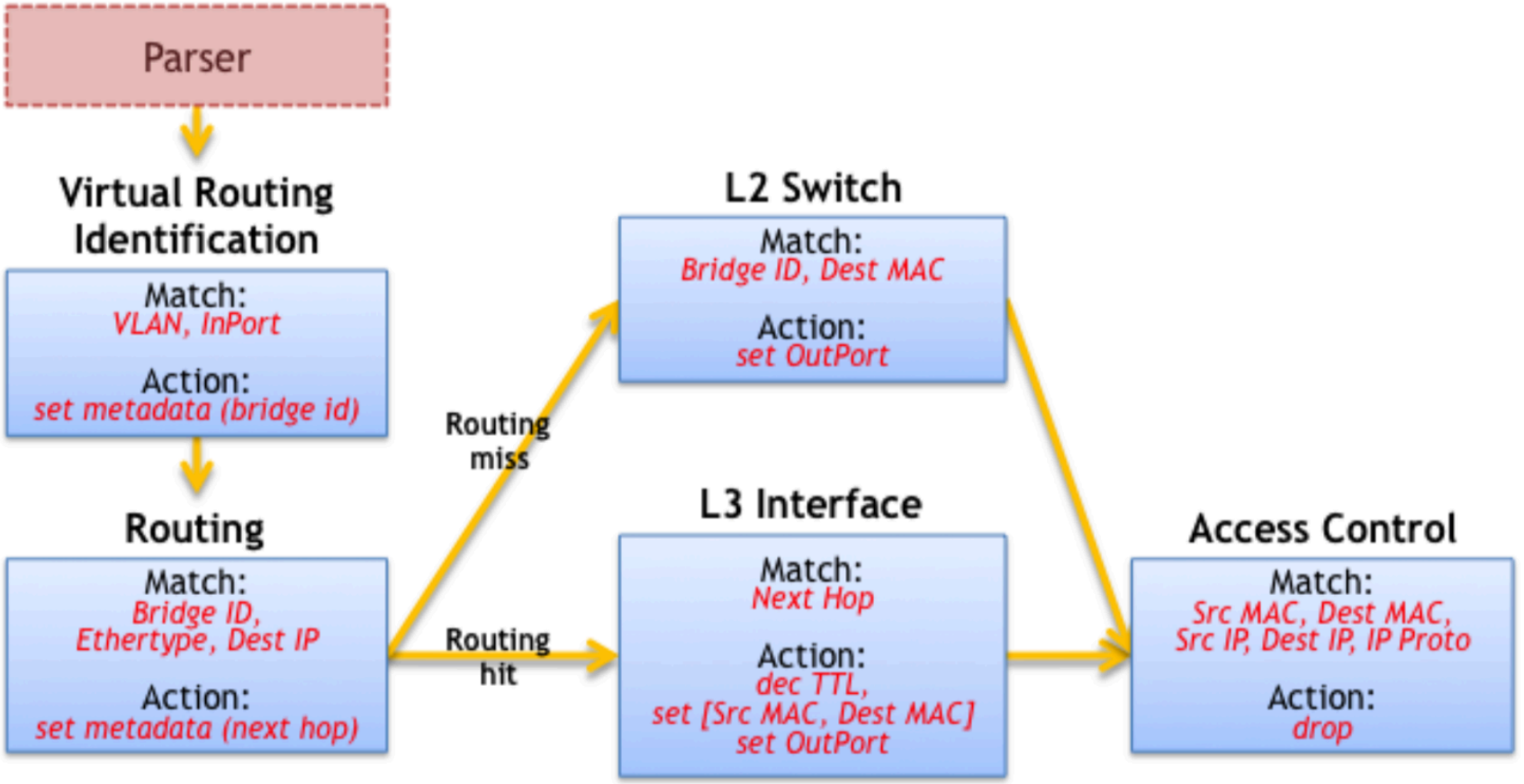
(2) Abstractions for stateless processing

- Goal: specify a set of tables & *control flow* between them

- Actions: more general than OpenFlow 1.0 forward/drop/count
  - Copy, add, remove headers
  - Arithmetic, logical, and bit-vector operations!

- Set *metadata* on packet header for control flow between tables

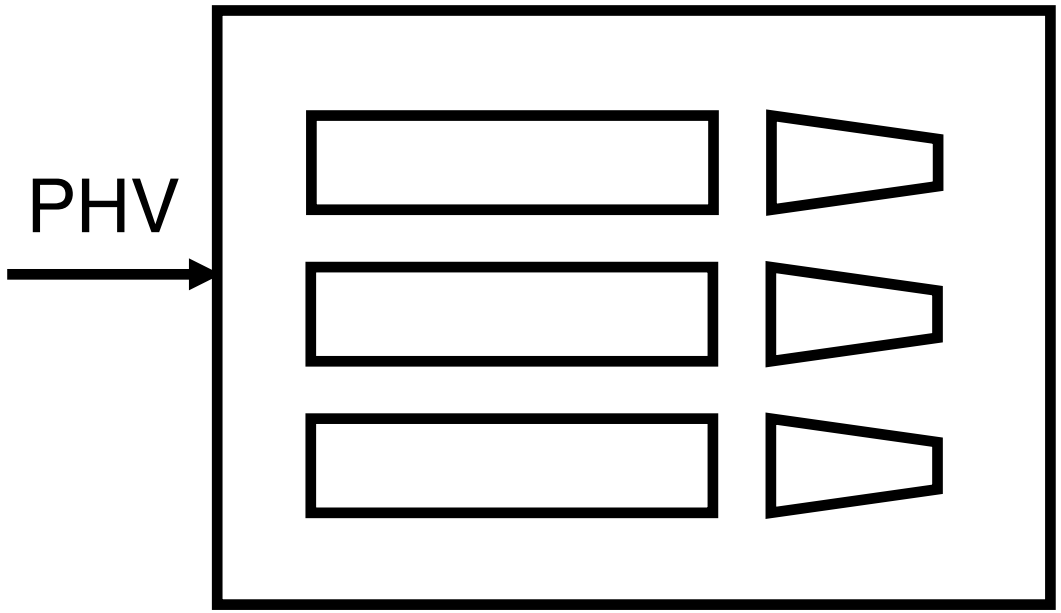
(2) Table dependency graph (TDG)



(2) Match-action table implementation

Mental model:

- Match and Action units supplied with the Packet Header Vector
- Each pipeline stage accesses its own local memory

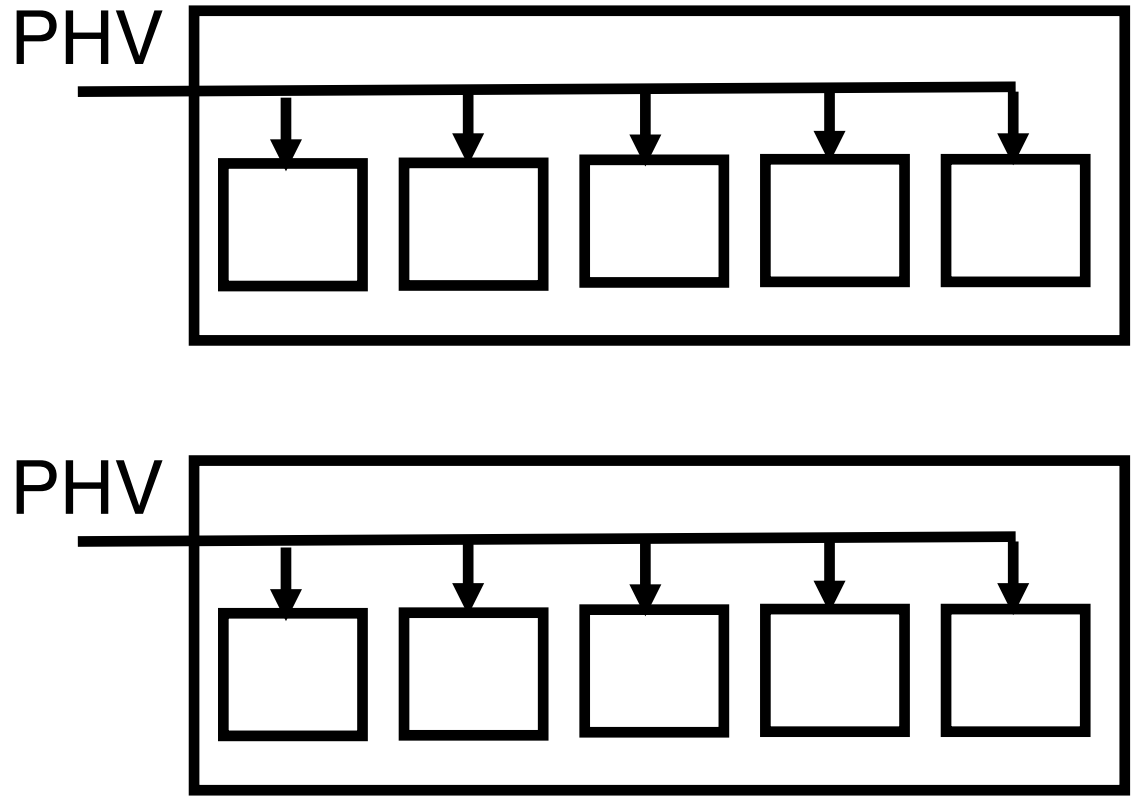


(2) Match-action table implementation

Hardware realization: separately configurable memory blocks

SRAM
- Exact match
- Action memory
- Statistics!

TCAM
- ternary match

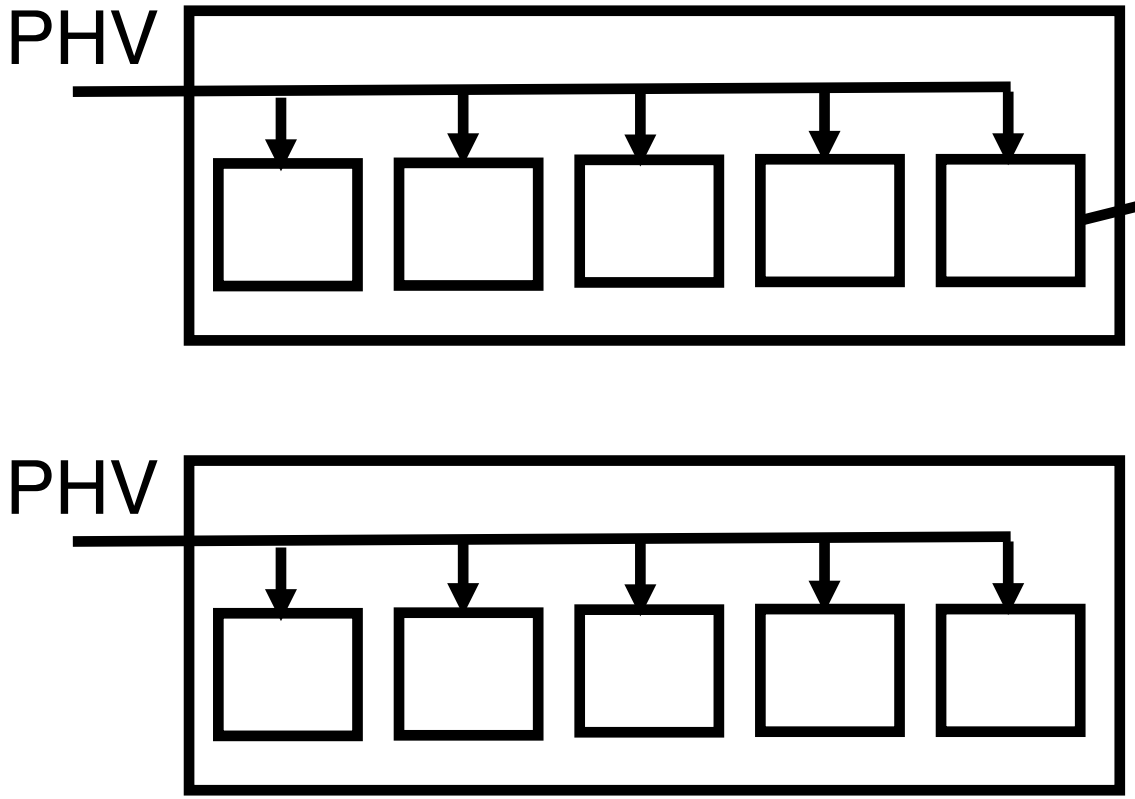


(2) Match-action table implementation

Hardware realization: separately configurable memory blocks

SRAM
- Exact match
- Action memory
- Statistics!

TCAM
- ternary match



Match RAM blocks also contain pointers to action memory and instructions

(1,2) Parse & pipeline specification with P4

- High-level goals
  - Allow reconfiguring packet processing in the field
  - Protocol independent
  - Target independent

- Declarative: specify parse graph and TDG
  - Headers, parsing, metadata
  - Tables, actions, control flow

- P4 separates table *configuration* from table *population*

(1,2) Parse & pipeline specification with P4

- Header and state machine spec

```
header_type
ethernet_t {

   fields {
         dstMac : 48;
         srcMac : 48;
         ethType : 16;
   }

}
```

```
header ethernet_t
ethernet;

parser start {

   extract(ethernet);
   return ingress;

}
```

(1,2) Parse & pipeline specification with P4

Rule Table

```
table forward {
    reads {
      ethernet.dstMac: exact;
    }
    actions {
      fwd;
      _drop;
     }
     size: 200;
}
```

Actions

```
action _drop() {
    drop();
}

action fwd(dport) {
    modify_field(standard_metadata.
      egress_spec, dport);
}
```

Control Flow

```
control ingress {
     apply(forward);
}
```

(3) Flexible *stateful* processing

- What if the action depends on previously seen (other) packets?
  - Example: send every 100th packet to a measurement server
  - Other examples: Flowlet switching, DNS TTL change tracking, XCP, …

- Actions in a single match-action table aren't expressive enough

- Example: `if (pkt.field1 + pkt.field2 == 10) { counter++; }`

(3) An example: “Flowlet” load balancing

- Consider the time of arrival of the current packet and the last packet *of the same flow*
- If current packet arrives 1 ms later than the last packet did, consider rerouting the packet to balance load
- Else, keep the packet on the same route as the last packet
- Q: why might you want to do this?

(3) Abstraction: Packet transaction

- A piece of code along with state that *runs to completion* on each packet before processing the next [Domino'16]

- Why is this challenging to implement on switch hardware?
  - Hint: Switch is clocked at 1 GHz!

(3) Abstraction: Packet transaction

- A piece of code along with state that *runs to completion* on each packet before processing the next [Domino'16]

- Why is this challenging to implement on switch hardware?
  - Hint: Switch is clocked at 1 GHz!

- (1) Switch must process a new packet every 1 ns
  - Transaction code may not run completely in one pipeline stage

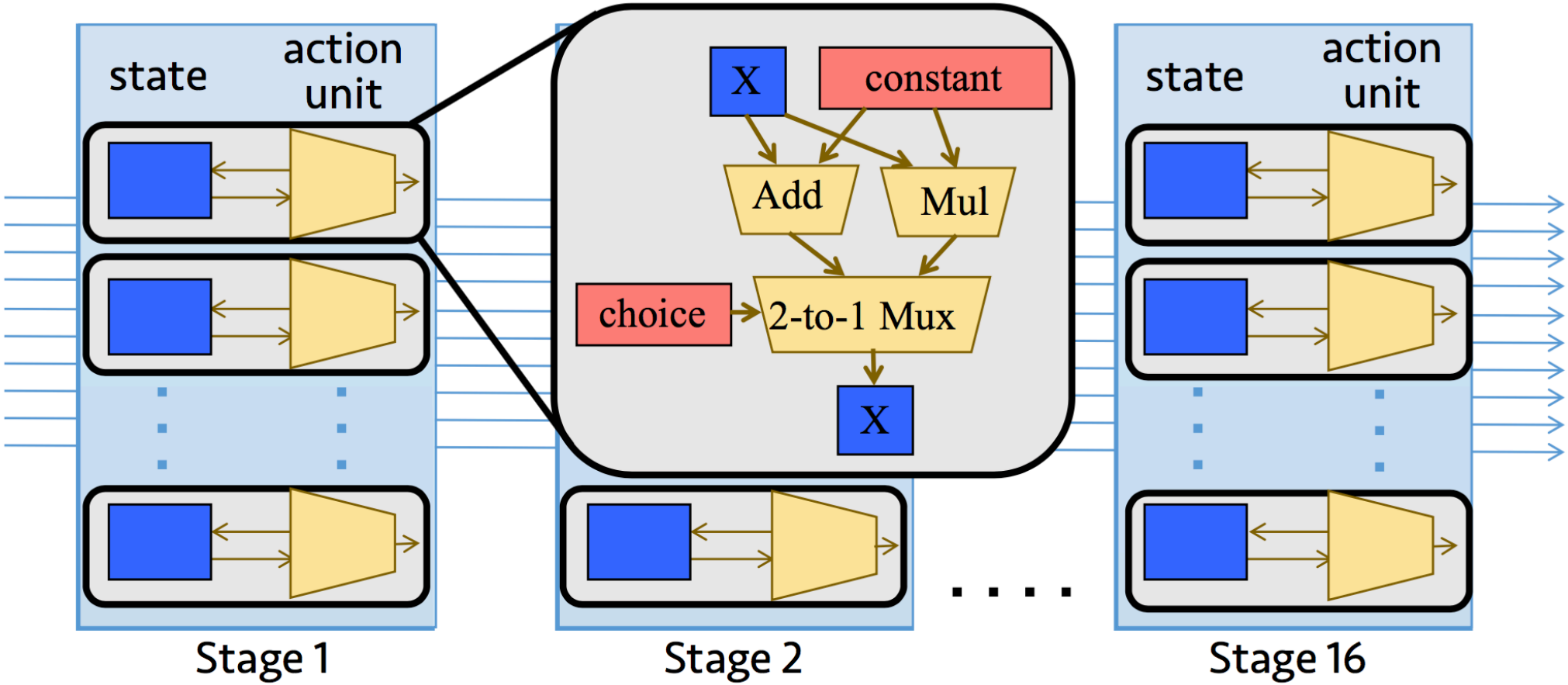
(3) Abstraction: Packet transaction

- A piece of code along with state that *runs to completion* on each packet before processing the next [Domino'16]

- Why is this challenging to implement on switch hardware?
  - Hint: Switch is clocked at 1 GHz!

- (1) Switch must process a new packet every 1 ns
  - Transaction code may not run completely in one pipeline stage
- (2) Read and write to state must happen in the same pipeline stage
  - Need *atomic operation* in hardware

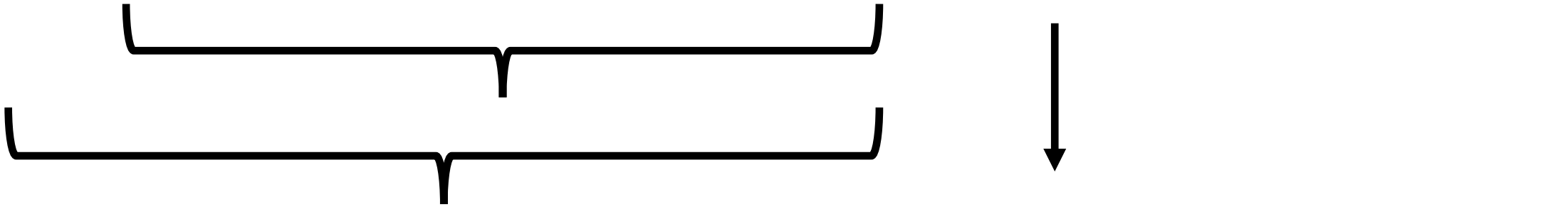
(3) Insight #1: Stateful atoms



The atoms constitute the switch's action instruction set: run under 1 ns

(3) Insight #2: Pipeline the stateless actions

- `if (pkt.field1 + pkt.field2 == 10) { counter ++; }`



Stateless operations (whose results depend only on the current packet) can execute over multiple stages

Only the stateful operation must run atomically in one pipeline stage

- Have a compiler do this analysis for us ☺

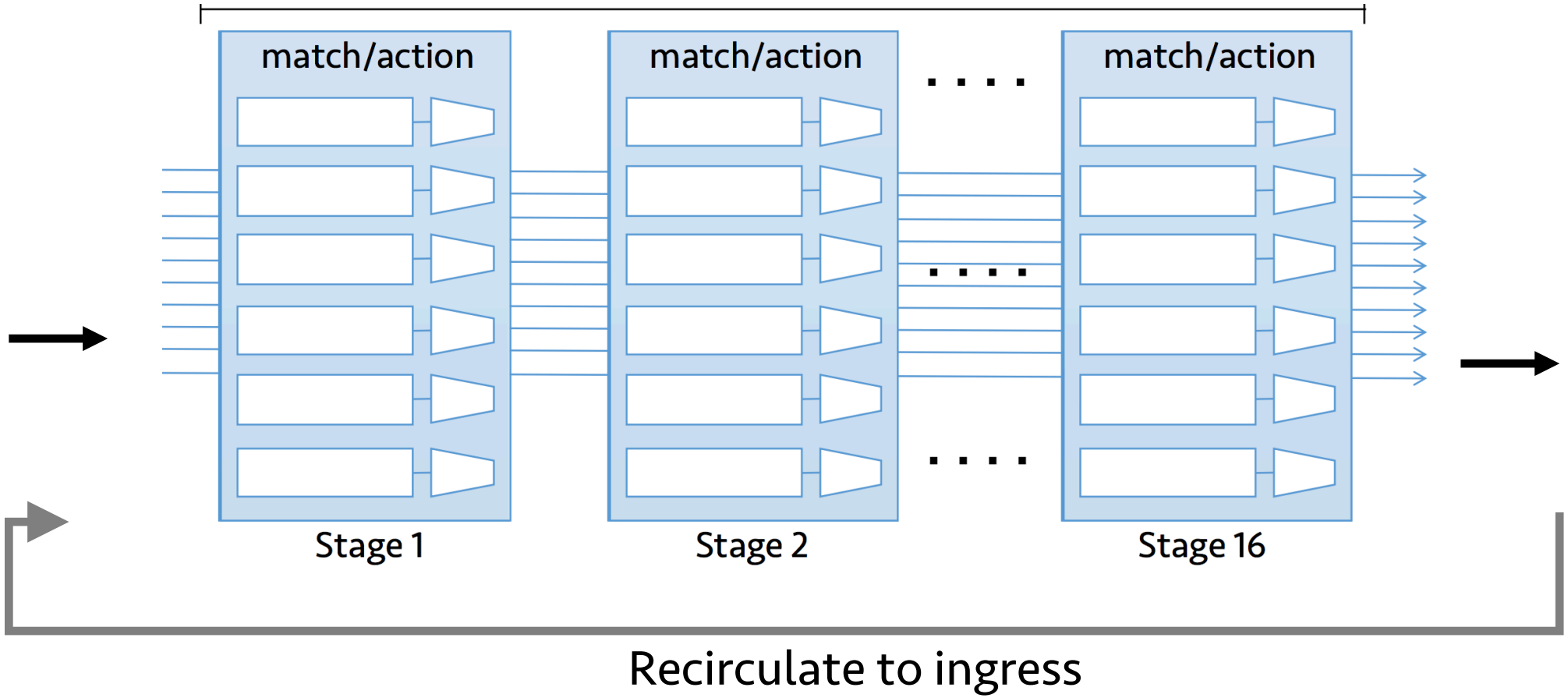
(4) Implementing complex policies

- What if you have a very large P4 (or Domino) program?

- Ex: too many logical tables in TDG
- Ex: logical table keys are too wide
  - Sharing memory across stages leads to paucity in physical tables
- Ex: too many (stateless) actions per logical table
  - Sharing compute across stages leads to paucity in physical tables

- Solution in RMT architecture: *Re-circulation*

(4) Re-circulation “extends” the pipeline



But throughput drops by 2x!

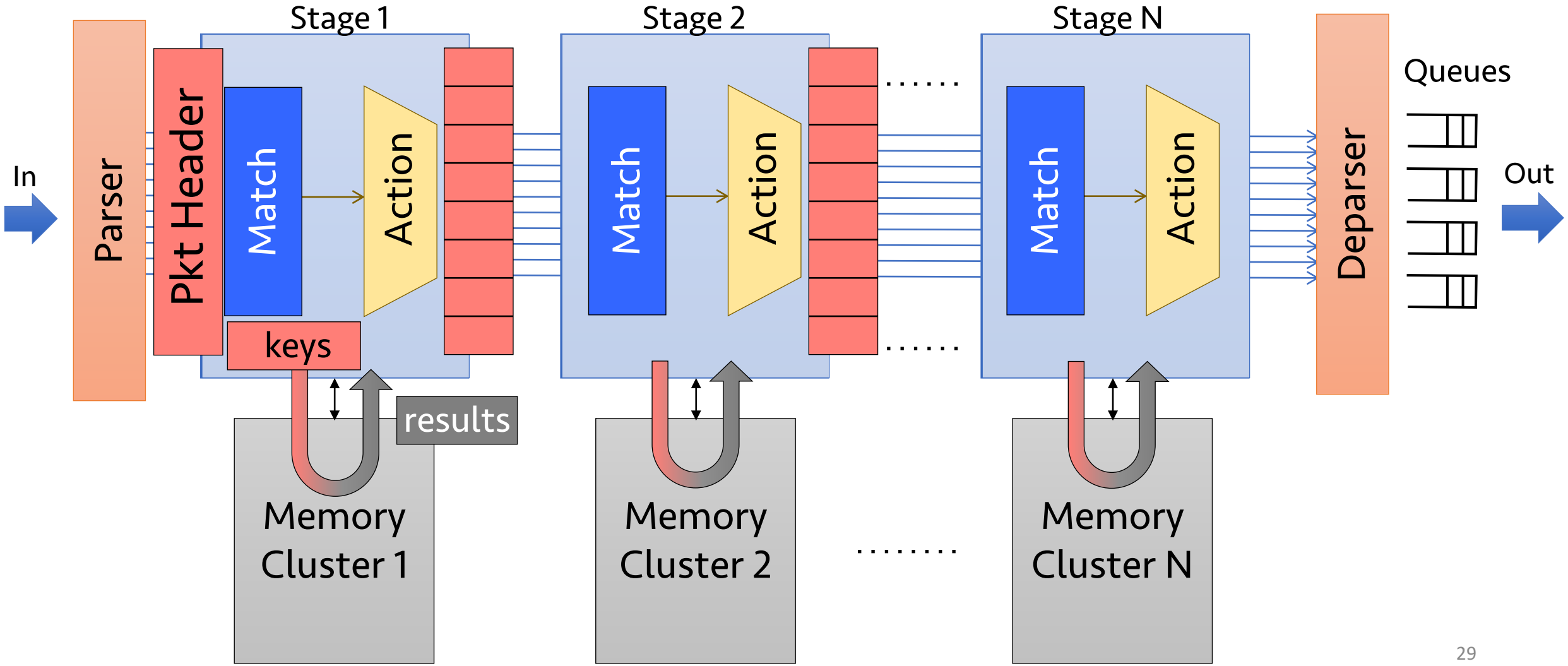
(4) Decouple pkt compute & mem access!

- Allow packet processing to run to completion on separate physical processors [dRMT'17]

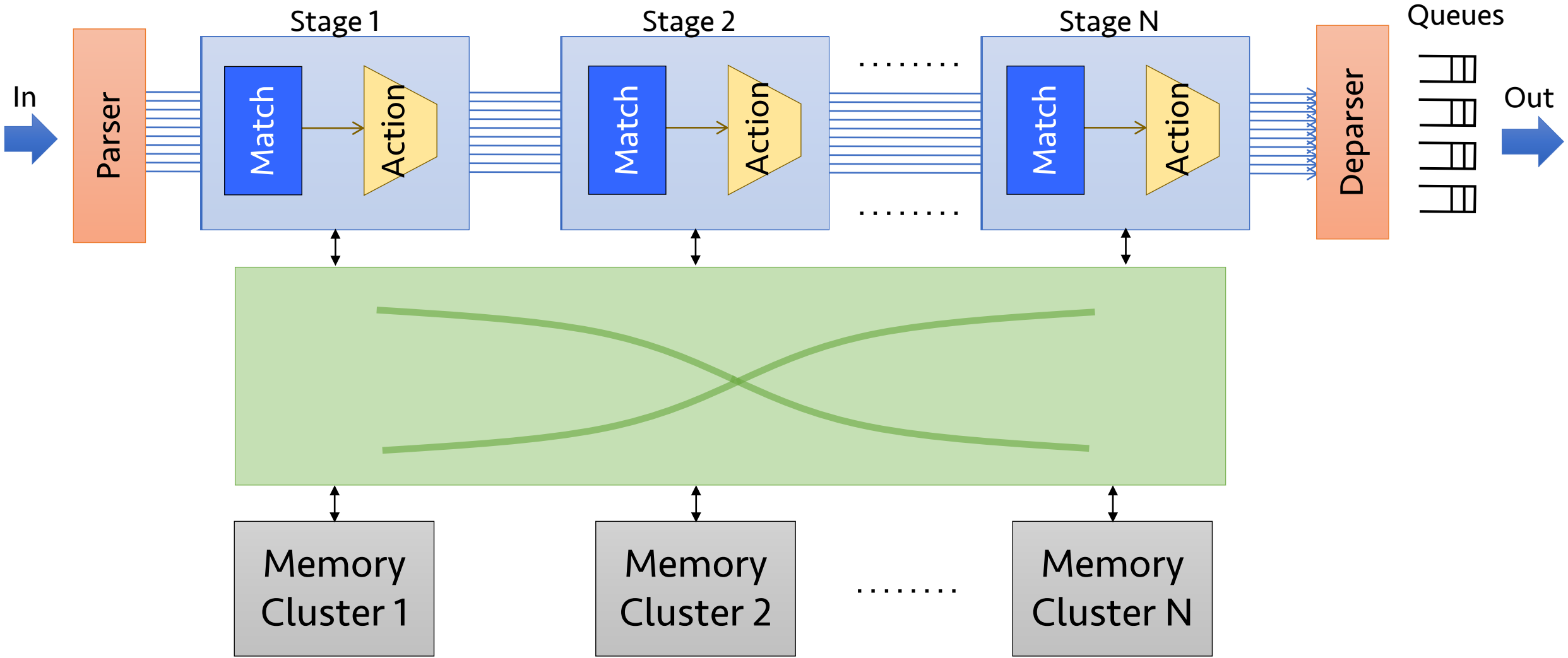
- Aggregate per-stage memory clusters into a shared memory pool
  - *Crossbar* enables all processors to access each memory

- *Schedule* instructions on each core to avoid contention

(4) RMT: compute and memory access



(4) dRMT: Memory disaggregation



(4) dRMT: Compute disaggregation