Buffering Techniques

Greg Stitt

ECE Department

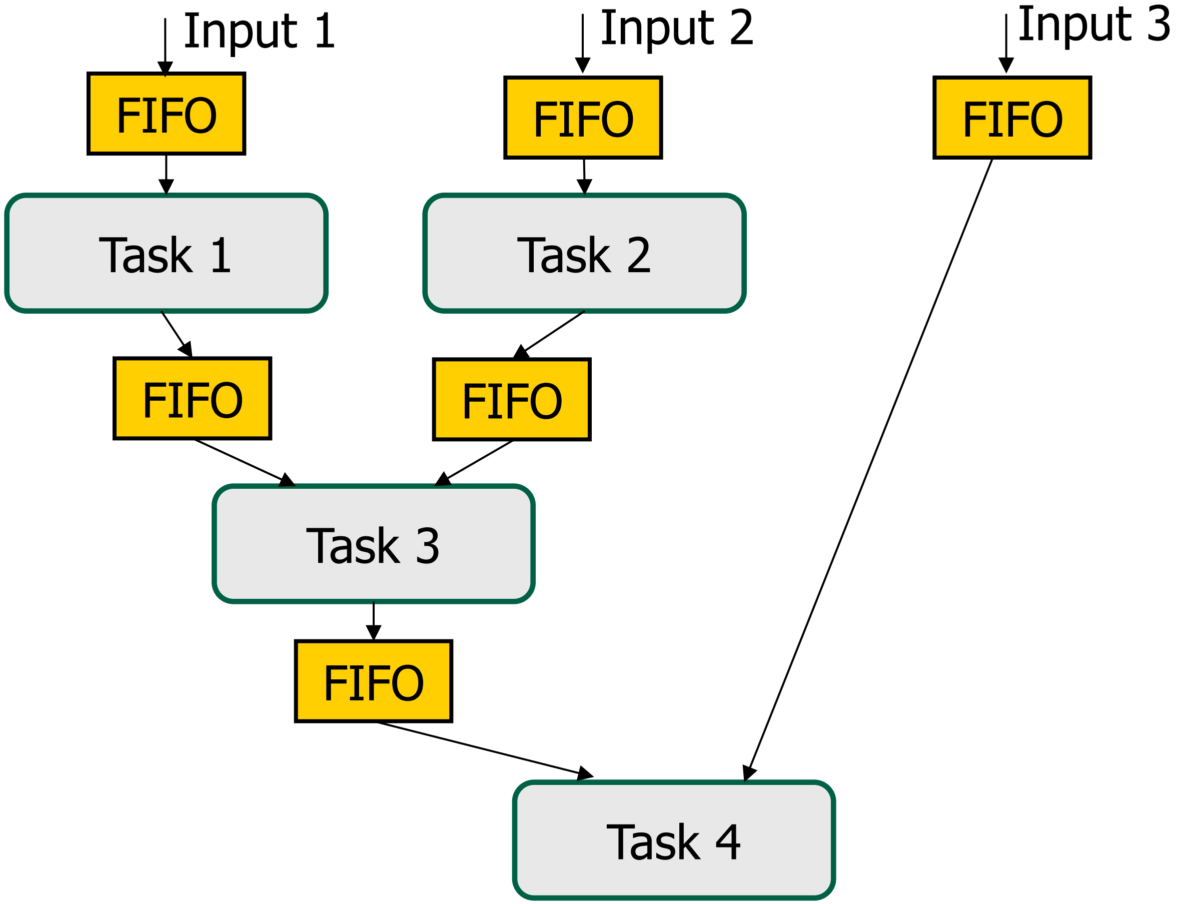
University of Florida

Buffers

- Purposes
  - Flow control for tasks with different production and consumption rates
  - Metastability issues
    - Memory clock likely different from circuit clock
    - Buffer stores data at one speed, circuit reads data at another
  - Stores “windows” of data, delivers to datapath
    - Window is set of inputs needed each cycle by pipelined circuit
    - Generally, more efficient than datapath requesting needed data
      - i.e. Push data into datapath as opposed to pulling data from memory
  - Conversion between memory and datapath widths
    - E.g. Bus is 64-bit, but datapath requires 128-bits every cycle
      - Input to buffer is 64-bit, output from buffer is 128 bits
      - Buffer doesn’t say it has data until receiving pairs of inputs

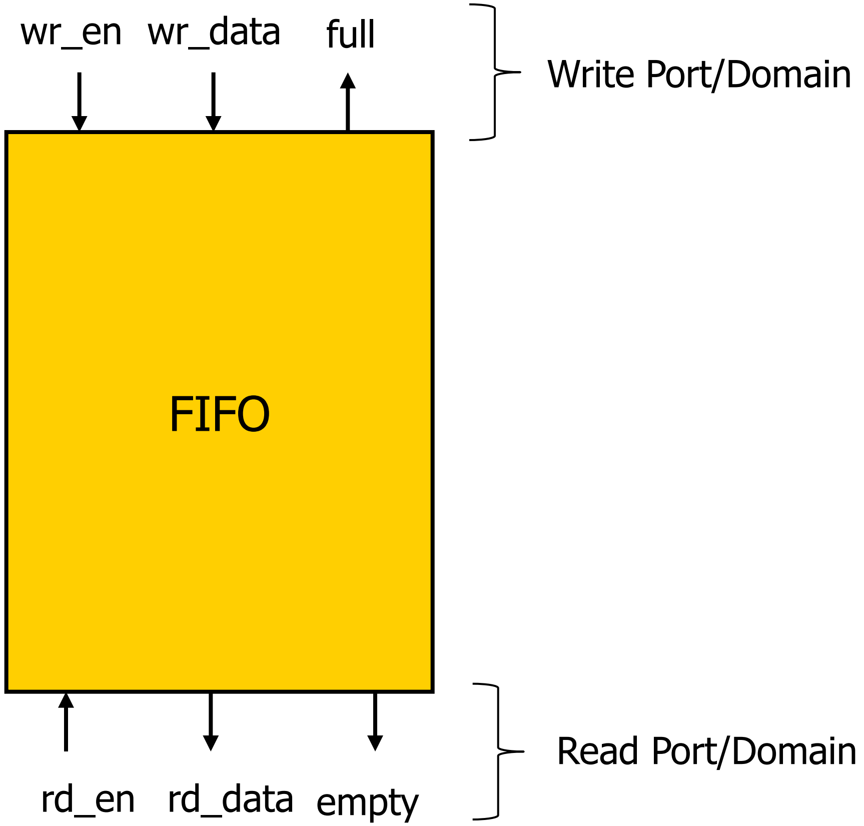
FIFOs

- Most common buffer: First-in First-out (FIFO)
  - Used for synchronizing execution of parallel tasks
    - i.e., “flow control”



FIFOs

- Beautifully simple control
  - Producer writes if not full (full provides “back pressure”)
  - Consumer reads if not empty



FIFO Sizing

- Tasks often produce/consume data at different rates
  - Tasks can also just be streams of data
  - e.g., Ethernet data, AXI bus data, memory data

- We often need to “size” the FIFO accordingly to handle different rates without back pressure
  - Fp = max message rate of producer (msg/s)
  - Fc = max message rate of consumer (msg/s)
  - We use abstract message to generalize the problem

FIFO Sizing

- Three common situations:

- 1) Fp <= Fc, assuming consumer is always ready
    - FIFO only needed if tasks on different clock domains
        - FIFO depth of 1 (or 0)
    - Why traditional pipelines use registers instead of FIFOs
        - The execution between pipeline operations (i.e. tasks) is synchronized implicitly every clock cycle, no need for a FIFO

- 2) Fp > Fc, assuming a constant producer stream
    - FIFO would have to be infinitely large to avoid filling up
    - If this is unavoidable, producer must monitor the full flag
        - i.e., “back pressure” to slow down a producer

FIFO Sizing

- 3) Fp > Fc, but producer sends data in bursts
  - E.g., Ethernet capable of 10/40/100 Gb/s, but data not available every cycle
  - We'll use B to represent the maximum number of messages that a producer might send consecutively (i.e., a burst)
  - We'll also assume that after a burst, the FIFO has time to empty
- How to determine FIFO depth that avoids back pressure?
  - Fp – Fc = rate that FIFO fills ups
    - e.g., after 1 second, FIFO would contain Fp – Fc messages
  - Next, we need to combine the "fill rate" with the duration of the burst B
    - B = # of consecutive messages
    - Fp = msg / s
    - Burst duration = B / Fp

FIFO Sizing

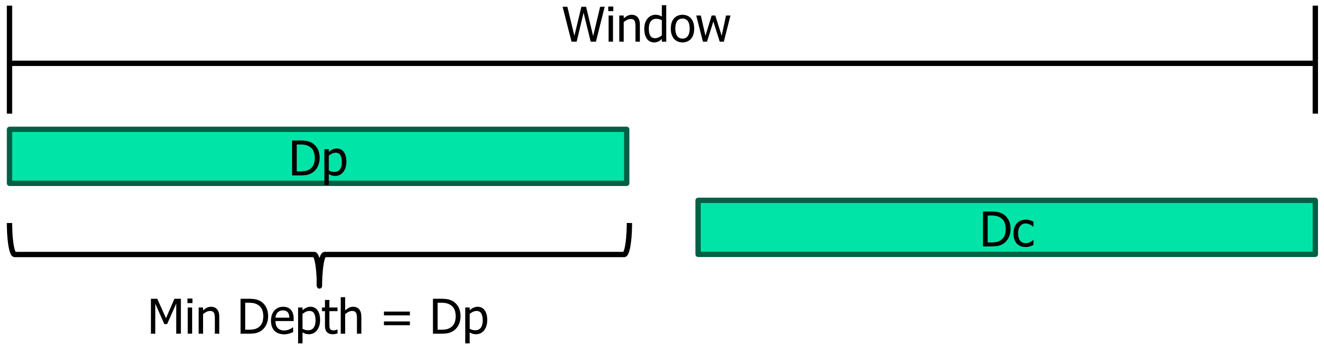
- FIFO depth = fill_rate * burst_duration
  = (Fp – Fc) * B / Fp
  = B * (1 – Fc / Fp)
  - E.g.:
    - Fp = 1M msgs / s
    - Fc = 250k msgs / s
    - B = 1000 msgs
    - FIFO Depth = 750
  - Remember: you could also use a smaller depth with back pressure

FIFO Sizing

- Similar problem
  - If time between bursts is sufficient for FIFO to drain, this means producer data rate and consumer data rate are equal within *some* window of time
  - Dp = Producer data rate (msgs / window)
  - Dc = Consumer data rate (msgs / window)
  - Dp = Dc
  - But, don't know when data produced/consumed within window
- What is the minimum FIFO depth to avoid back pressure?
  - Sliding window of time
    - Can't produce more than Dp messages within *any* window
  - Worst case situation:
    - All Dp messages arrive at beginning of window
    - All Dc messages are consumed at end of window

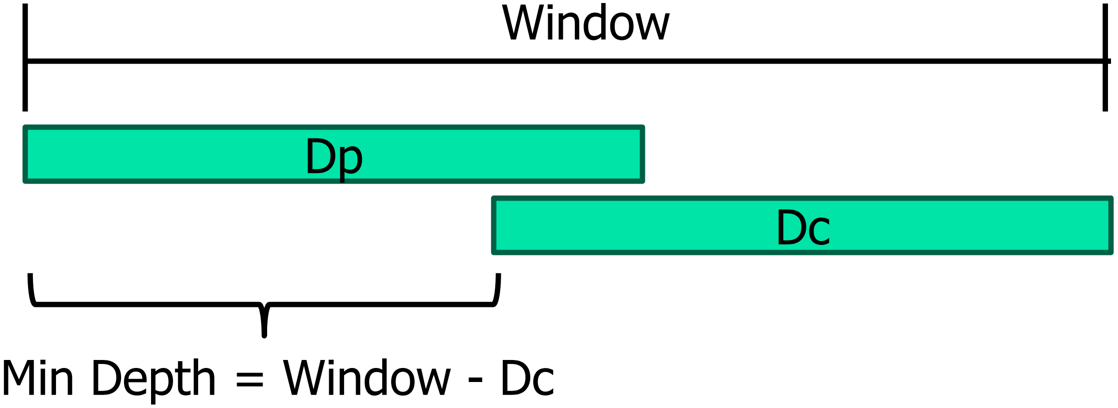
FIFO Sizing

- Consider several situations:
- 1) Window is much longer than time to produce/consume all messages



FIFO Sizing

- 2) Window small enough to create overlap between Dp and Dc

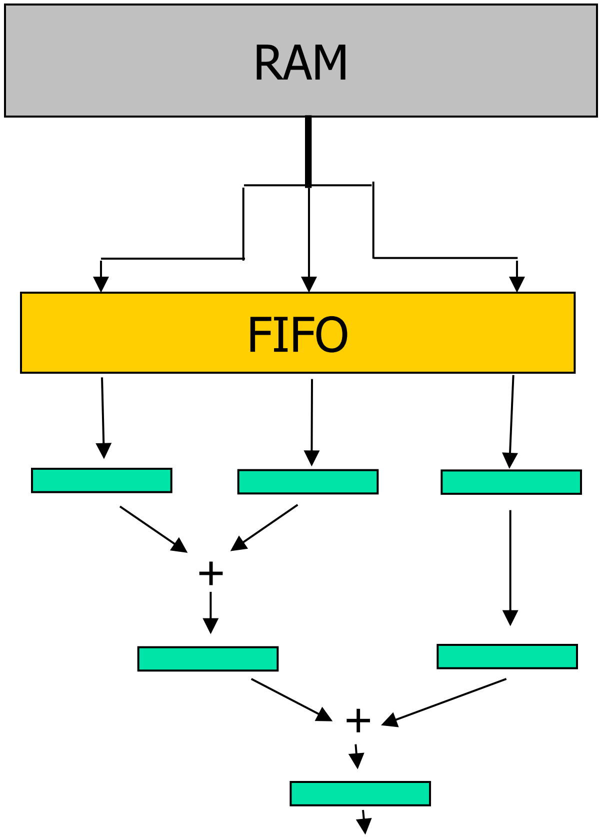


- Two situations can be combined:
  - **FIFO depth = min(window – Dc, Dp)**

- Examples for Dp = Dc = 10 messages:
  - Window = 100 cycles => FIFO depth = 10
  - Window = 15 cycles => FIFO depth = 5
  - Window = 10 cycles => FIFO depth = 0 (not needed unless different clocks)

FIFOs

- FIFOs often used between RAM and pipeline
  - Outputs data in order read from memory

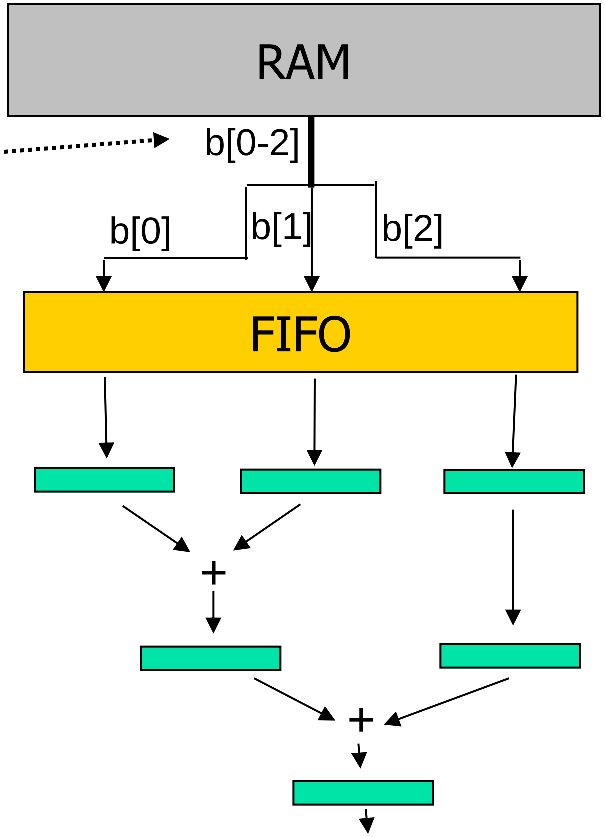


```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

FIFOs

- FIFOs often used between RAM and pipeline
  - Outputs data in order read from memory

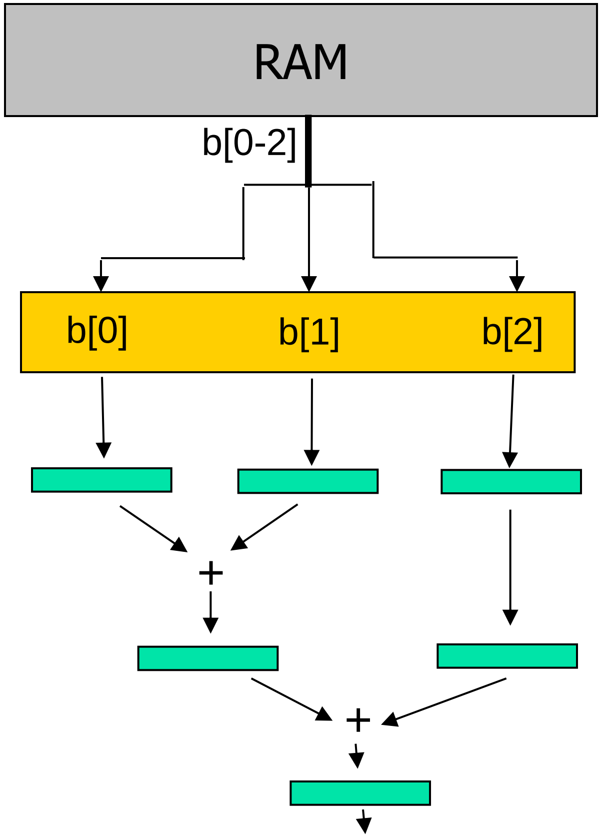
First window read from memory



```
for (i=0; i < 100; I++)
  a[i] = b[i] + b[i+1] + b[i+2];
```

FIFOs

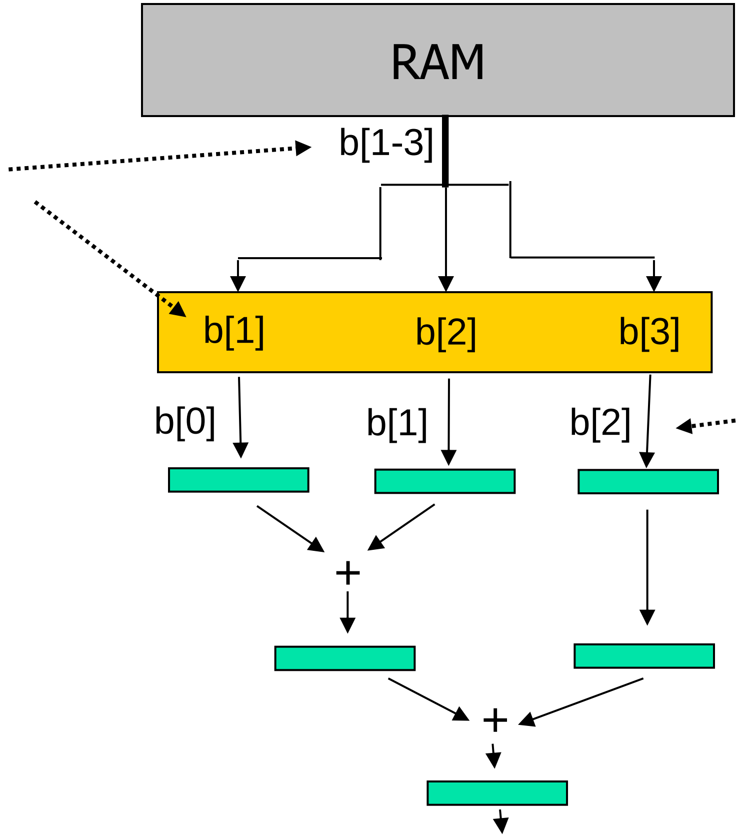
- FIFOs often used between RAM and pipeline
  - Outputs data in order read from memory



```
for (i=0; i < 100; I++)
  a[i] = b[i] + b[i+1] + b[i+2];
```

FIFOs

- FIFOs often used between RAM and pipeline
  - Outputs data in order read from memory



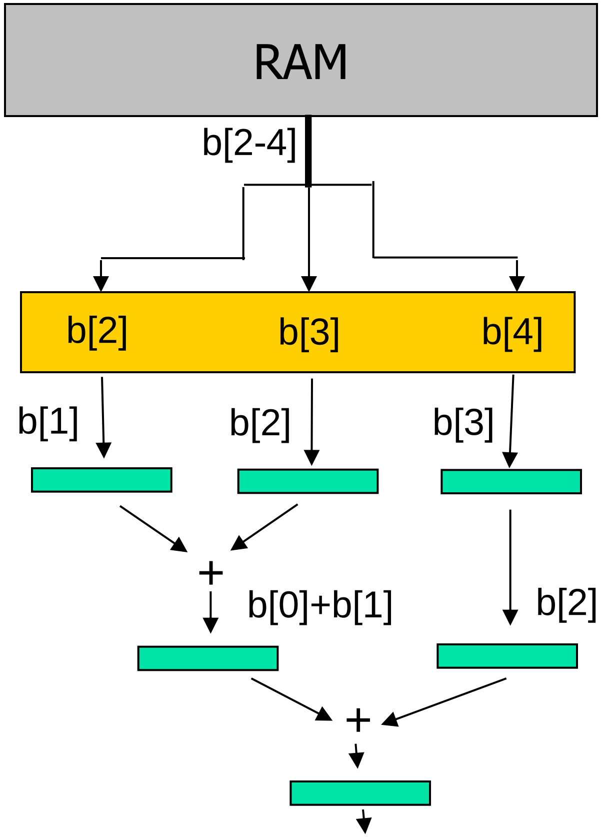
Second window read from RAM

First window pushed to datapath

```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

FIFOs

- FIFOs often used between RAM and pipeline
    - Outputs data in order read from memory



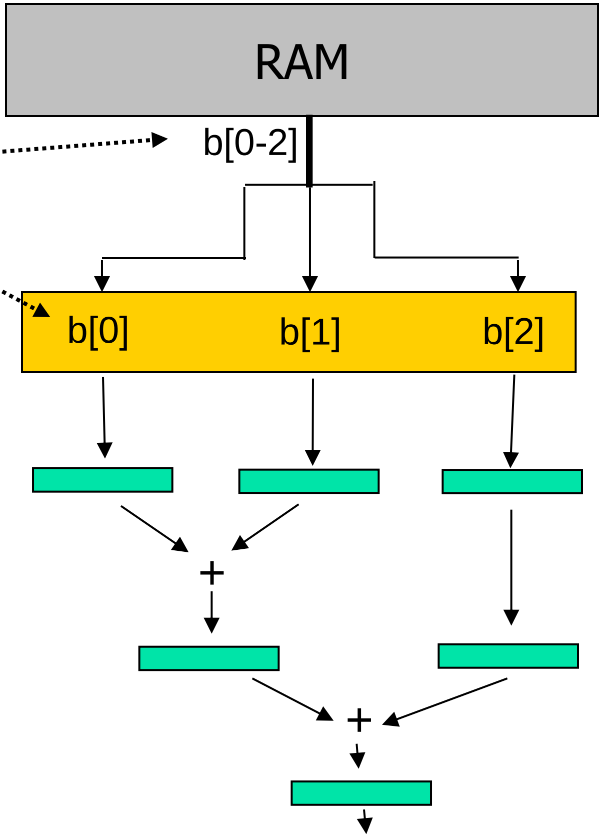
```
for (i=0; i < 100; I++)
  a[i] = b[i] + b[i+1] + b[i+2];
```

FIFOs

- Timing issues
  - Memory bandwidth too small
    - Circuit stalls or wastes cycles while waiting for data
  - Memory bandwidth larger than data consumption rate of circuit
    - May happen if area exhausted

FIFOs

- Memory bandwidth too small



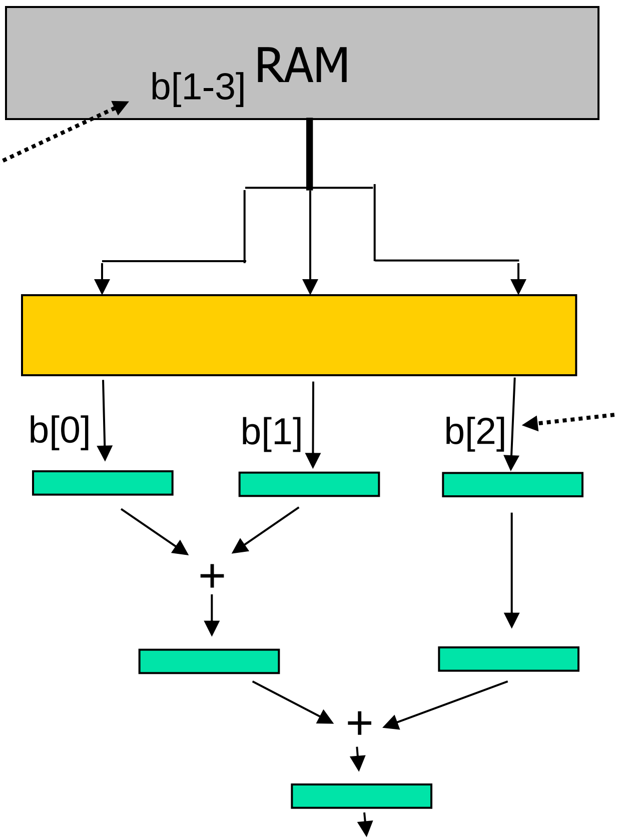
First window read from memory into FIFO

```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

FIFOs

- Memory bandwidth too small

2nd window requested from memory, but not transferred yet



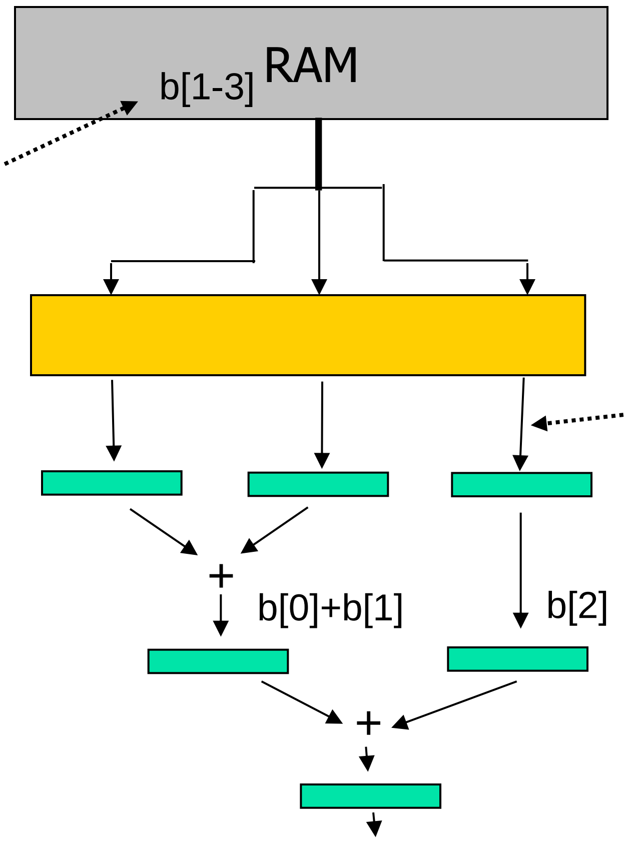
```
for (i=0; i < 100; I++)
  a[i] = b[i] + b[i+1] + b[i+2];
```

1st window pushed to datapath

FIFOs

- Memory bandwidth too small

2nd window requested from memory, but not transferred yet



```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

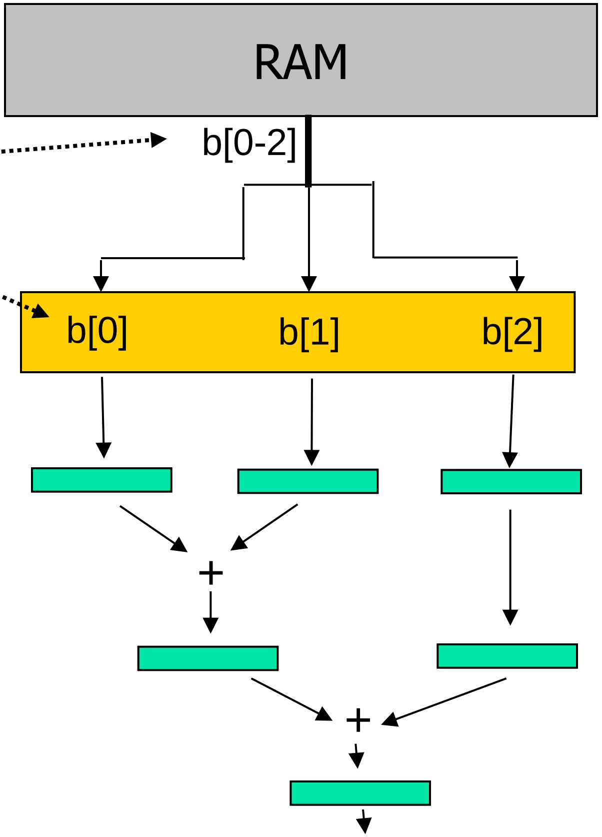
No data ready (wasted cycles)

Alternatively, could have prevented 1st window from proceeding (stall cycles) - *necessary if feedback in pipeline*

FIFOs

- Memory bandwidth larger than data consumption rate

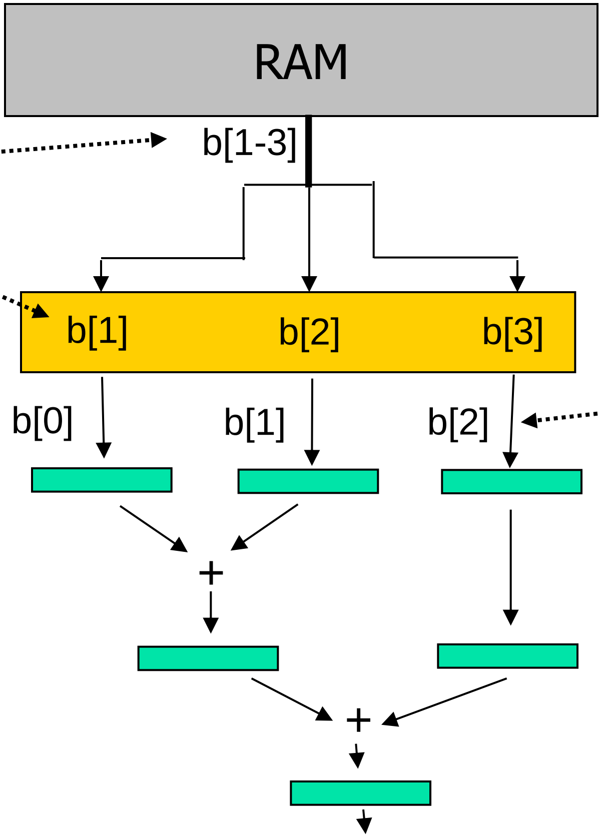
1st window read from memory into FIFO



```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

FIFOs

- Memory bandwidth larger than data consumption rate



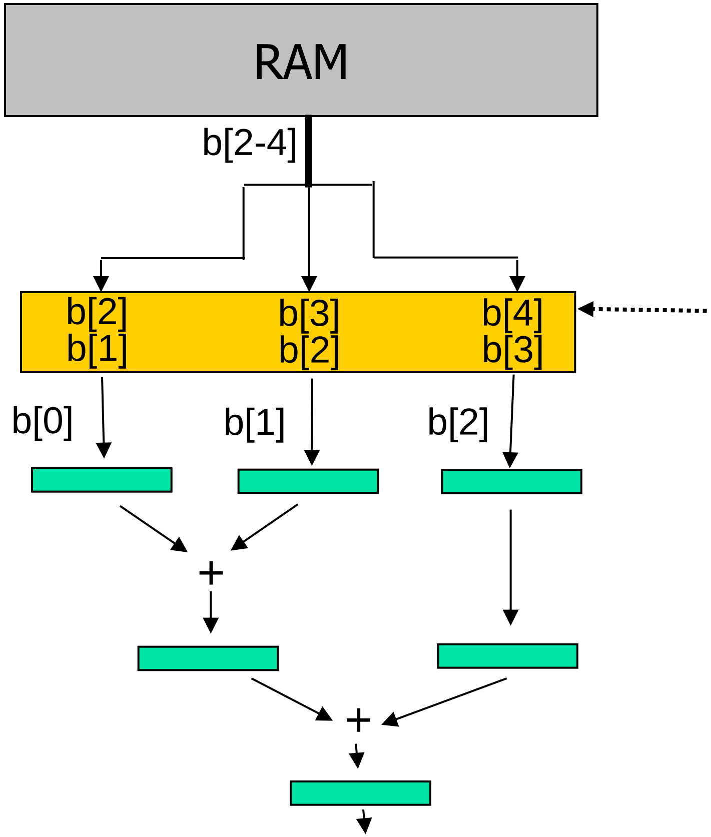
```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

2nd window read from memory into FIFO.

1st window pushed to datapath

FIFOs

- Memory bandwidth larger than data consumption rate



```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

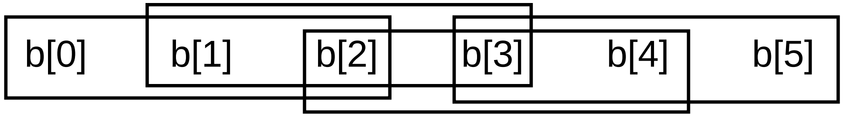
Data arrives faster than circuit can process it – FIFOs begin to fill

If FIFO full, memory reads stop until not full

Improvements

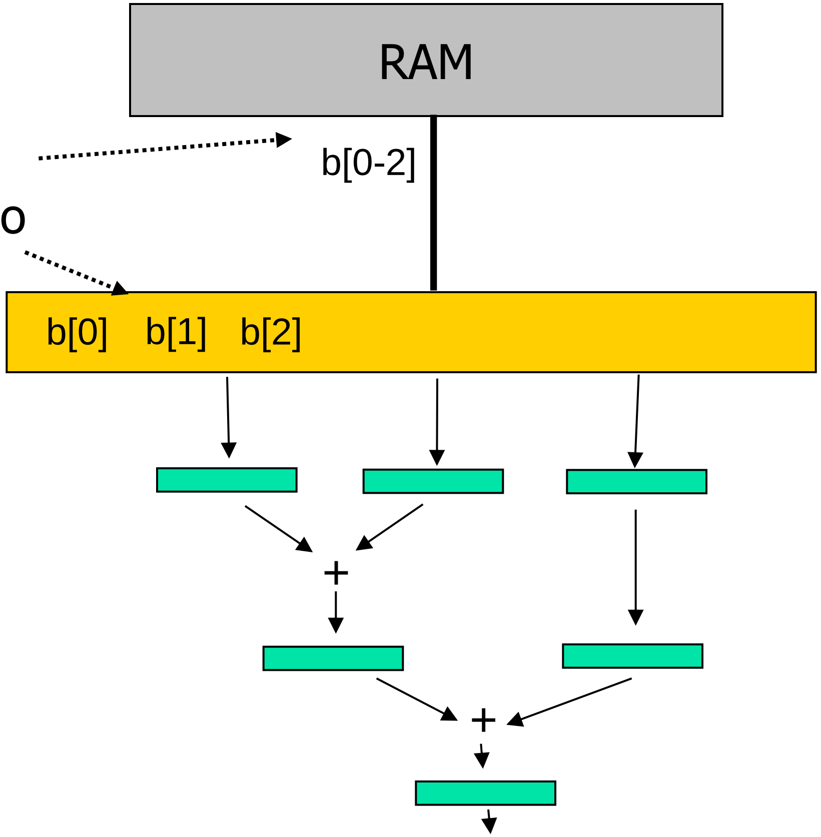
- Do we need to fetch entire window from memory for each iteration?
  - Only if windows of consecutive iterations are mutually exclusive
- Commonly, consecutive iterations have overlapping windows
  - Overlap represents "reused" data
  - Ideally, would be fetched from memory just once
- Smart Buffer *[Guo, Buyukkurt, Najjar LCTES 2004]*
  - Part of the ROCCC compiler
  - Analyzes memory access patterns
  - Detects "sliding windows", reused data
  - Prevents multiple accesses to same data

```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```



Smart Buffer

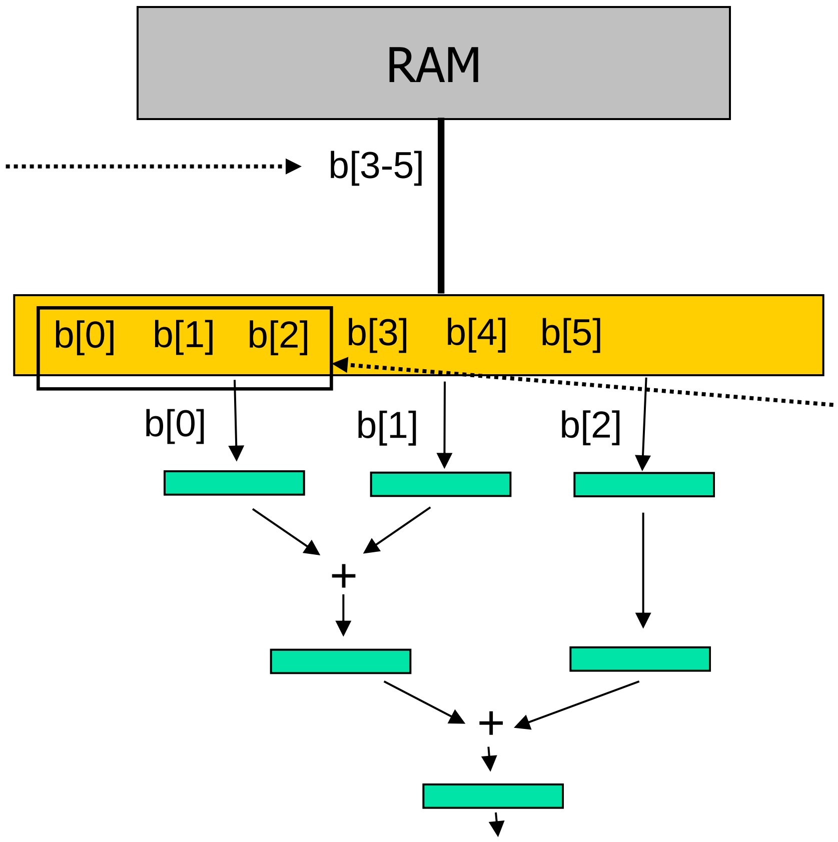
1st window read from memory into smart buffer



```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

Smart Buffer

Continues reading needed data, but *does not* reread b[1-2]

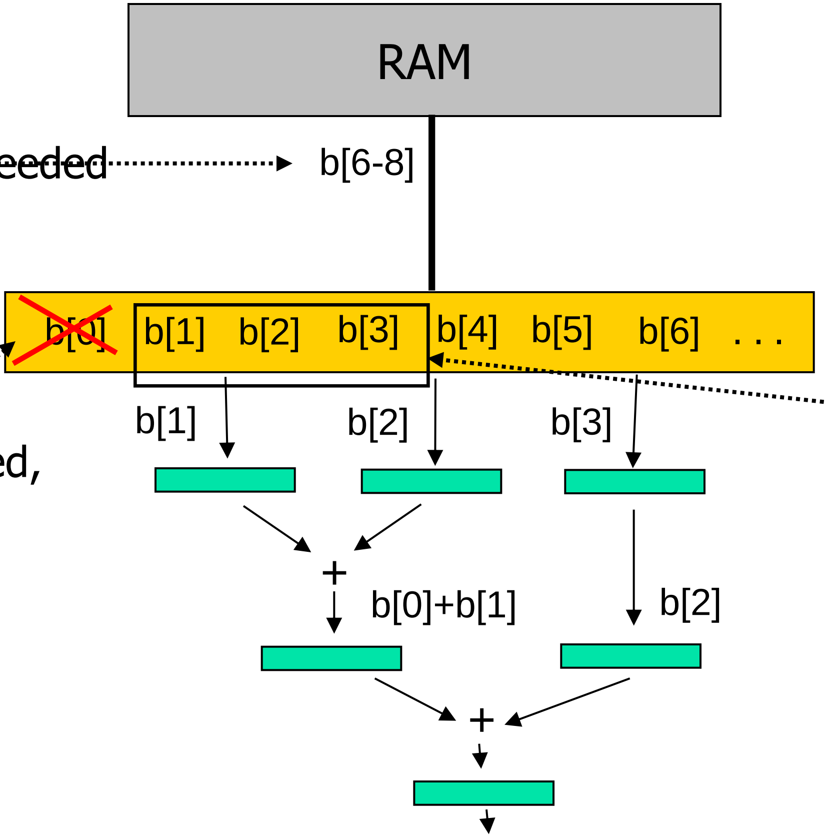


1st window pushed to datapath

```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

Smart Buffer

Continues fetching needed data (as opposed to windows)

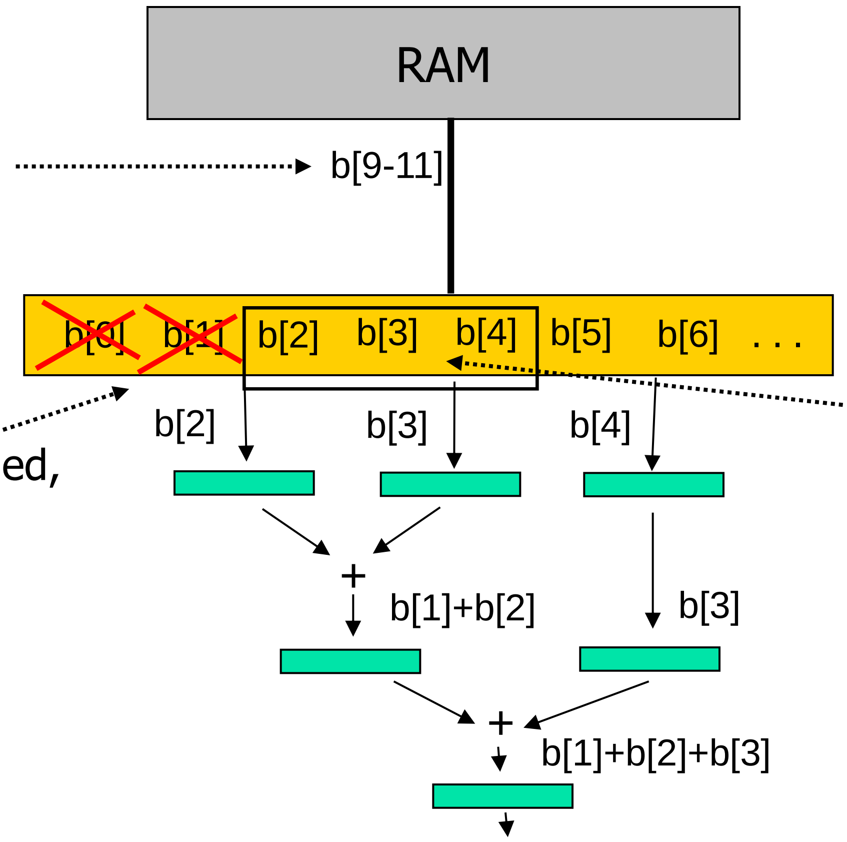


```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

2nd window pushed to datapath

b[0] no longer needed, deleted from buffer

Smart Buffer



And so on

b[1] no longer needed, deleted from buffer

3rd window pushed to datapath

```
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

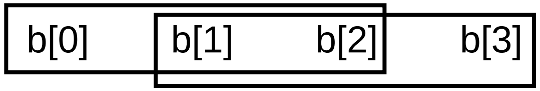
Comparison with FIFO

- FIFO - *fetches a window for each iteration*
  - Reads 3 elements every iteration
    - 100 iterations * 3 accesses/iteration = 300 memory accesses
- Smart Buffer - *fetches as much data as possible each cycle, buffer assembles data into windows*
  - Ideally, reads each element once
    - # accesses = array size
    - Circuit performance is equal to latency plus time to read array from memory
      - **No matter how much computation, execution time is approximately equal to time to stream in data!**
        - Note: Only true for streaming examples
  - 102 memory accesses
    - Essentially improves memory bandwidth by 3x
    - How does this help?
      - **Smart buffers enable more unrolling**

Unrolling with Smart Buffers

- Assume bandwidth = 128 bits
    - We can read 128/32 = 4 array elements each cycle
    - First access doesn’t save time
        - No data in buffer
        - Same as FIFO, can unroll once

```
long b[102];
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

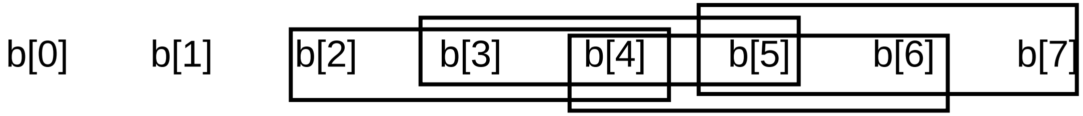


First memory access allows only 2 parallel iterations

Unrolling with Smart Buffers

- After first window is in buffer
    - Smart Buffer Bandwidth = Memory Bandwidth + Reused Data
        - 4 elements + 2 reused elements *(b[2],b[3])*
        - Essentially, provides bandwidth of 6 elements per cycle
    - Can perform 4 iterations in parallel
        - **2x speedup compared to FIFO**

```
long b[102];
for (i=0; i < 100; I++)
   a[i] = b[i] + b[i+1] + b[i+2];
```



However, every subsequent access enables 4 parallel iterations (b[2] and b[3] already in buffer)

Datapath Design w/ Smart Buffers

- Datapath based on unrolling enabled by smart buffer bandwidth (not memory bandwidth)
    - Don’t be confused by first memory access
- Smart buffer waits until initial windows in buffer before passing any data to datapath
    - Adds a little latency
        - But, avoids 1st iteration requiring different control due to less unrolling

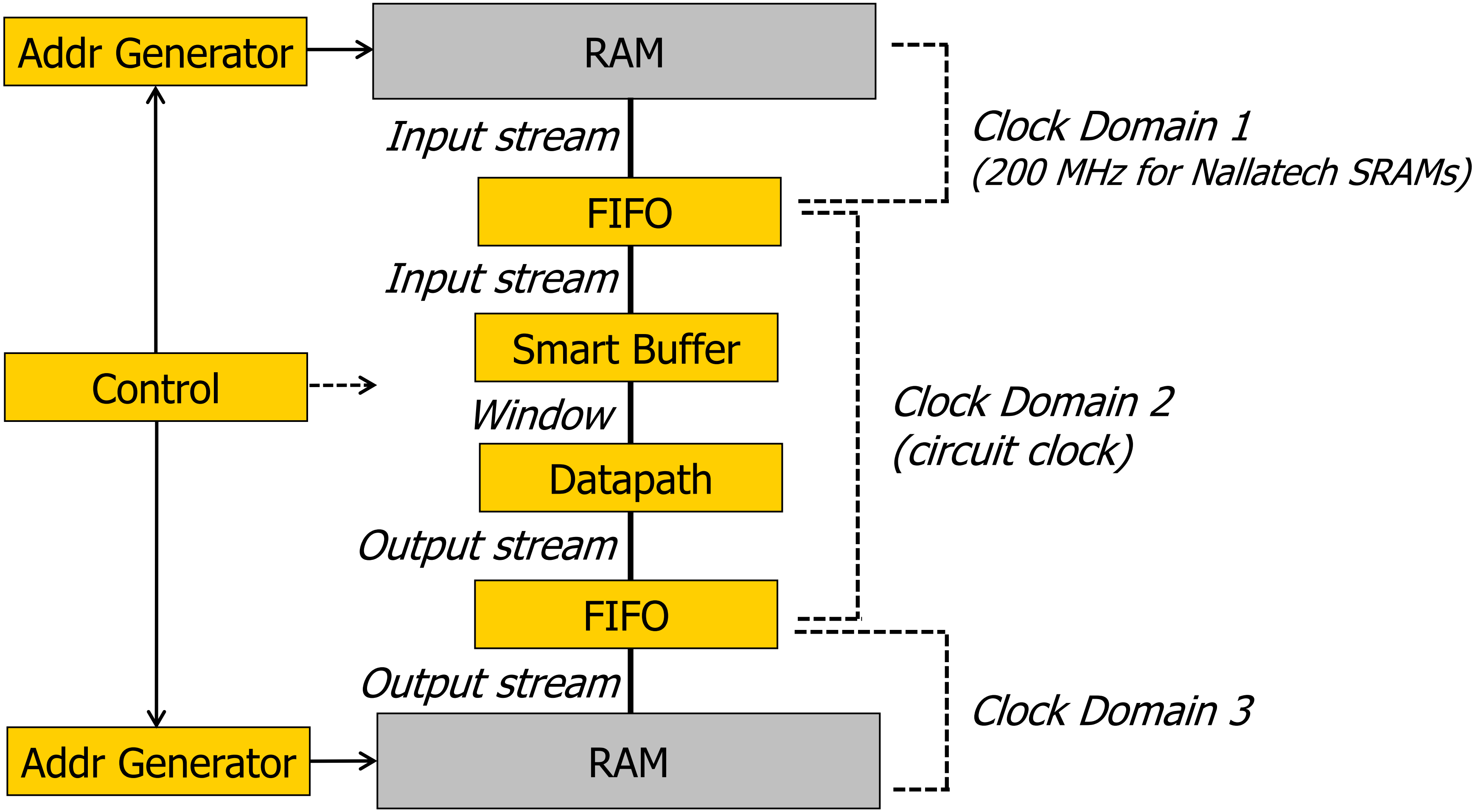
Another Example

- Your turn

```
short b[1004], a[1000];
for (i=0; i < 1000; i++)
   a[i] = avg( b[i], b[i+1], b[i+2], b[i+3], b[i+4] );
```

- Analyze memory access patterns
  - Determine window overlap
- Determine smart buffer bandwidth
  - Assume memory bandwidth = 128 bits/cycle
- Determine maximum unrolling with and without smart buffer
- Determine total cycles with and without smart buffer
  - Use previous systolic array analysis (latency, bandwidth, etc).

Complete Architecture



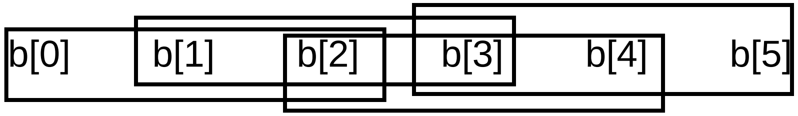
Smart Buffer Implementation

- Several possibilities (see papers on website)
  - 1) Register smart buffer
  - 2) BRAM smart buffer
- Recommended methodology
  - 1) Determine unrolling amount, which defines window size
  - 2) Determine buffer size and allocate resources
  - 3) Add steering logic to update buffer for each consecutive window

Register Smart Buffers

```
for (i=0; i < 100; i++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

- 1) Determine unrolling amount
  - Assume FIFO delivers 4 elements/cycle
  - 2 elements reused in each iteration
  - Smart buffer b.w. = 4+2 = 6 elements/cycles
- Can execute 4 iterations in parallel
  - Window size = 6 elements
  - 1st window = b[0-5], 2nd window = b[4-9], etc.



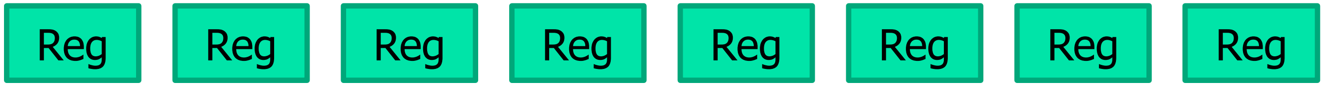
- Important: remember that unrolling also limited by output bandwidth
  - Assume output FIFO can write 4 elements/cycles

Register Smart Buffers

```
for (i=0; i < 100; i++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

- 2) Determine buffer size and allocate resources
  - Buffer size = # of elements read from FIFO to get first window
  - Assume FIFO provides 4 elements
  - Reading one window (6 elements) requires 2 FIFO accesses
    - 8 total elements read from FIFO
    - Therefore, buffer consists of 8 registers

Smart buffer:

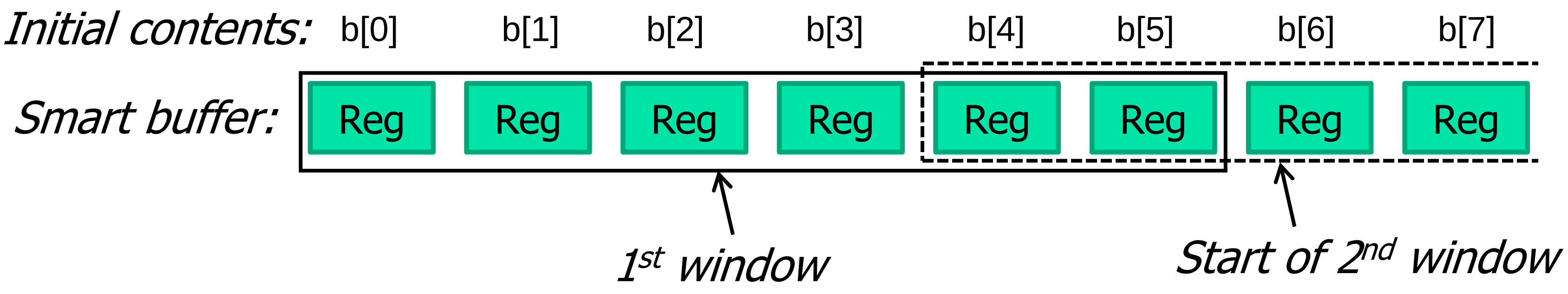


Reg Reg Reg Reg Reg Reg Reg Reg

Register Smart Buffers

```
for (i=0; i < 100; i++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

- 3) Add steering logic
  - Determine by analyzing access patterns



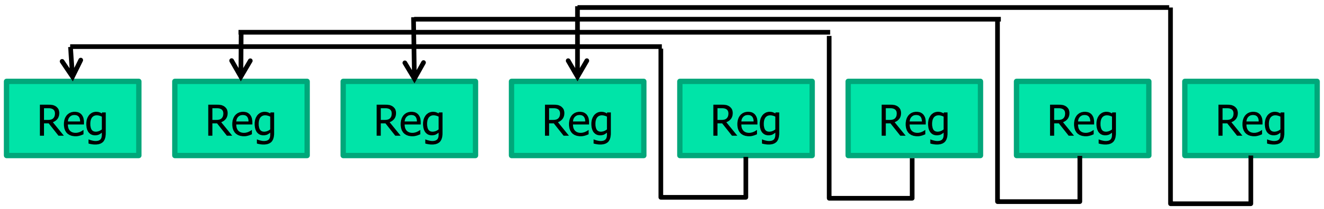
Initial contents: b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7]

Smart buffer: Reg Reg Reg Reg Reg Reg Reg Reg

1st window

Start of 2nd window

- Need to shift left by 4



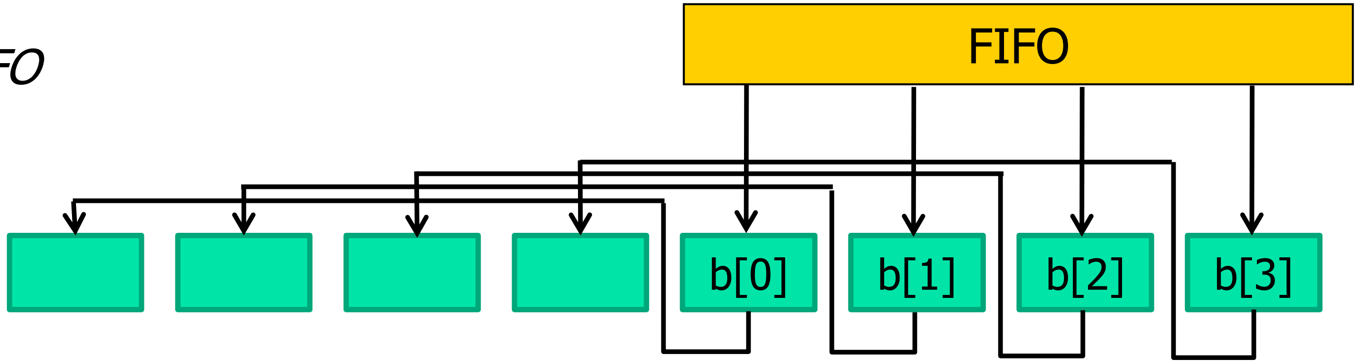
Reg Reg Reg Reg Reg Reg Reg Reg

Register Smart Buffers

```
for (i=0; i < 100; i++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

- Steering logic, cont.
  - But, how does initial window get to the appropriate location?

1st read from FIFO

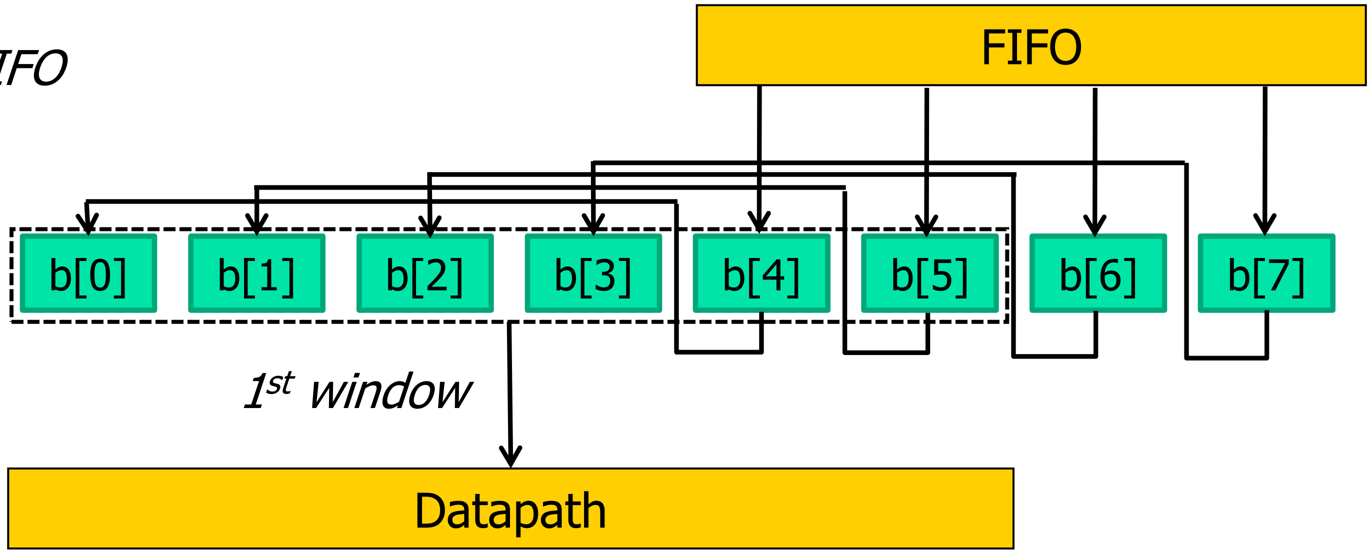


Register Smart Buffers

```
for (i=0; i < 100; i++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

- Steering logic, cont.
  - But, how does initial window get to the appropriate location?

2nd read from FIFO

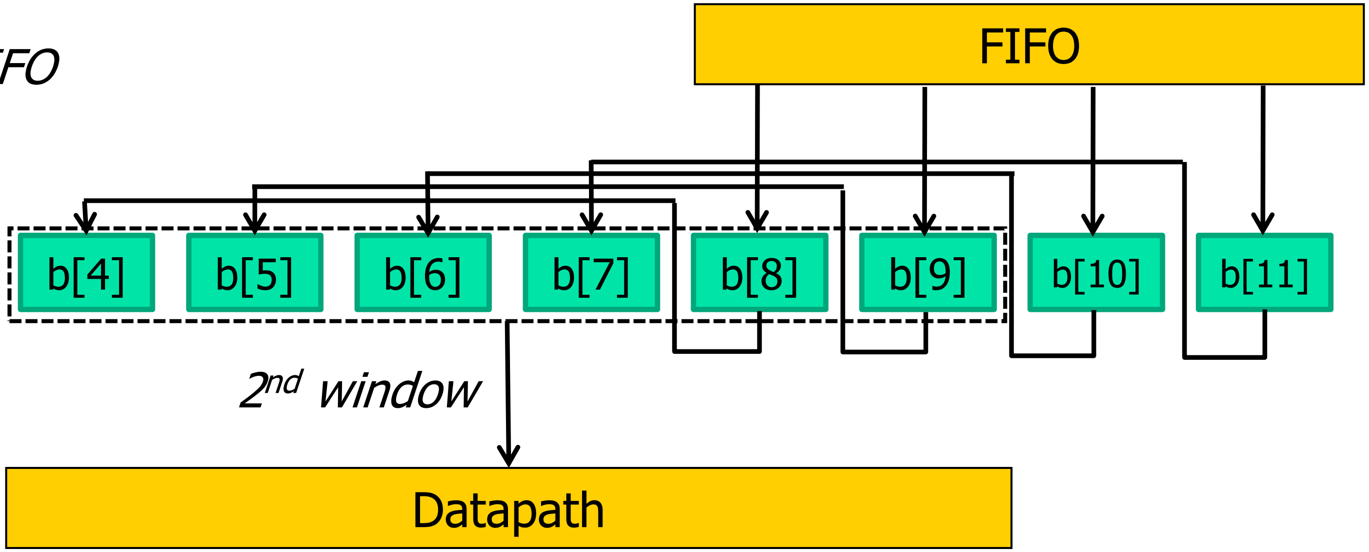


Register Smart Buffers

```
for (i=0; i < 100; i++)
   a[i] = b[i] + b[i+1] + b[i+2];
```

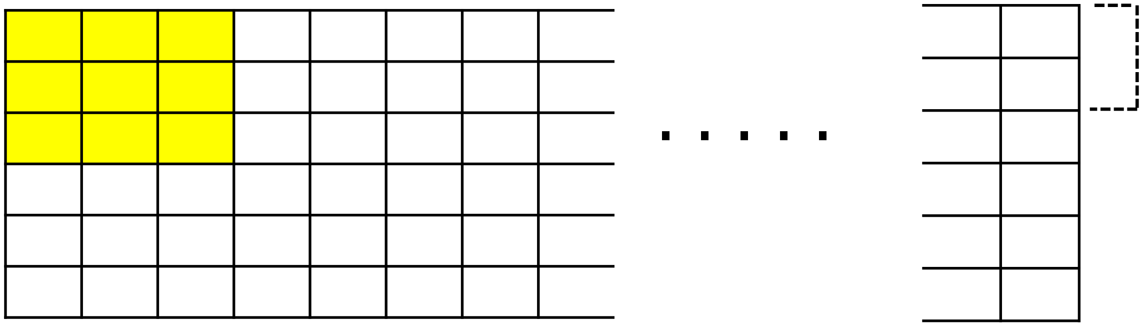
- Steering logic, cont.
  - But, how does initial window get to the appropriate location?

3rd read from FIFO



Block RAM Smart Buffers

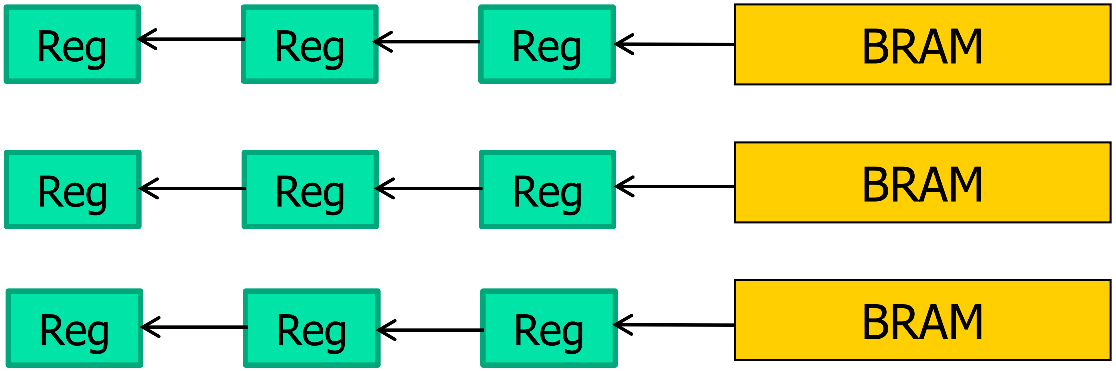
- Register smart buffers always a possibility
  - But, may have a huge area overhead
- Remember:
  - Buffer size = # of elements read from memory to get first window
- For 1-D examples, usually not much overhead
- Not the case for 2-D examples
- Example:
  - For a 3x3 window, buffer must read first 2 rows of input
  - For an image 1024 pixels wide, smart buffer would require more than 2024 registers



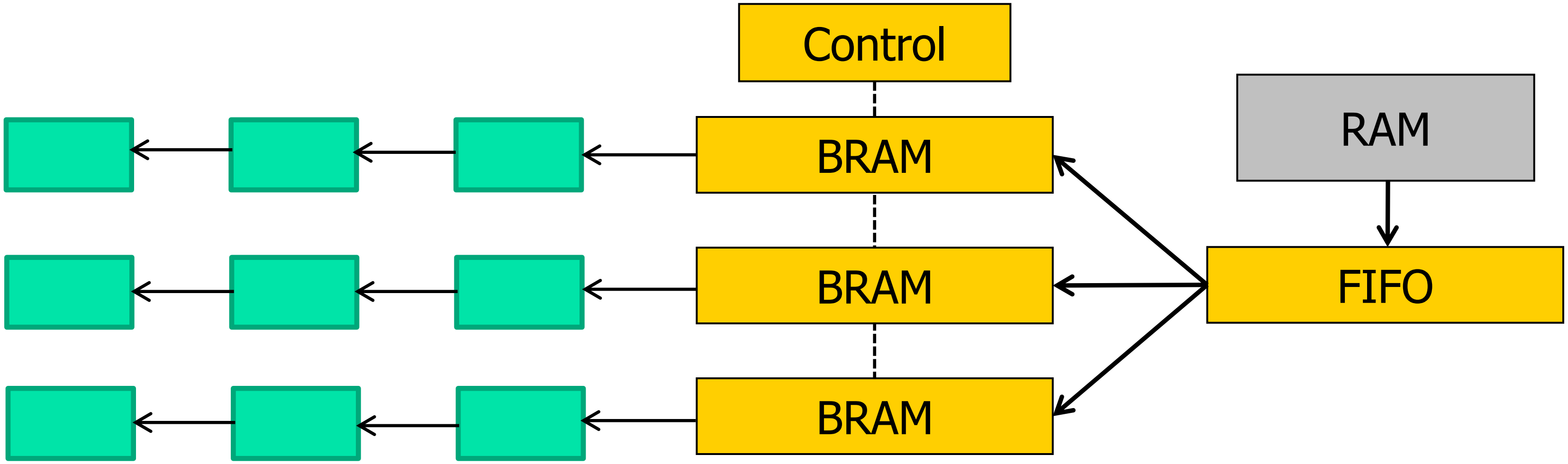
1st 2 rows of input must be read before 1st window is available

Block RAM Smart Buffers

- Instead, store each row in block RAM
  - # BRAMs = # of rows in window
  - Size of BRAM = # of columns
- Shift data into a 2-D register array equal to the size of the window
- Example for 3x3 window:

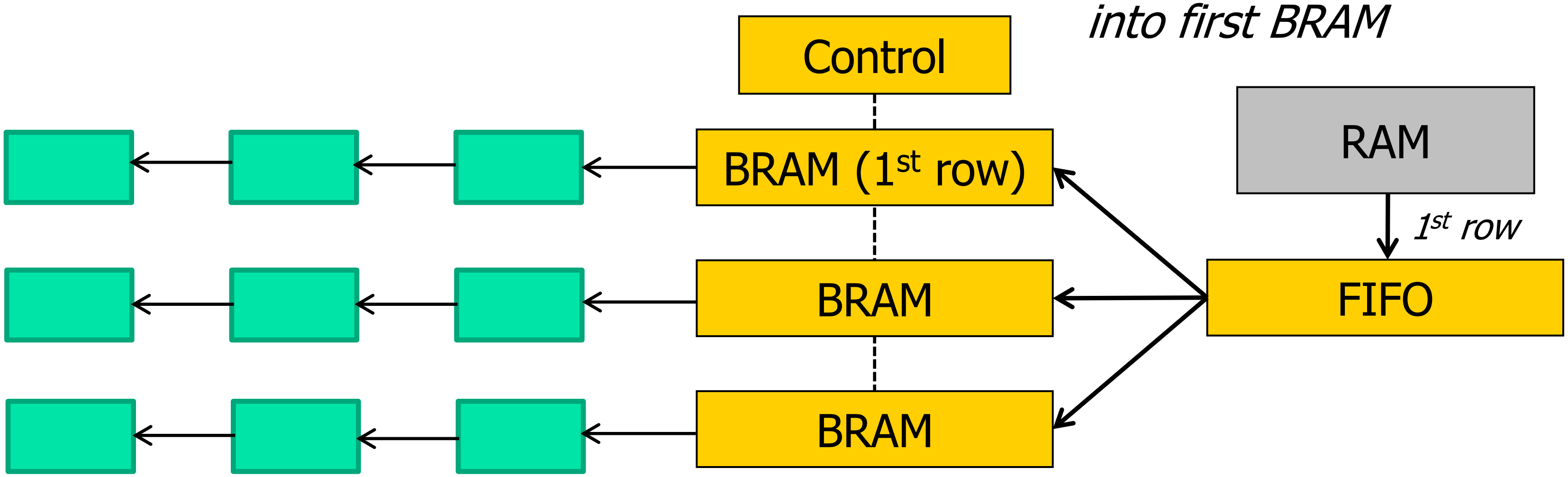


Block RAM Smart Buffers

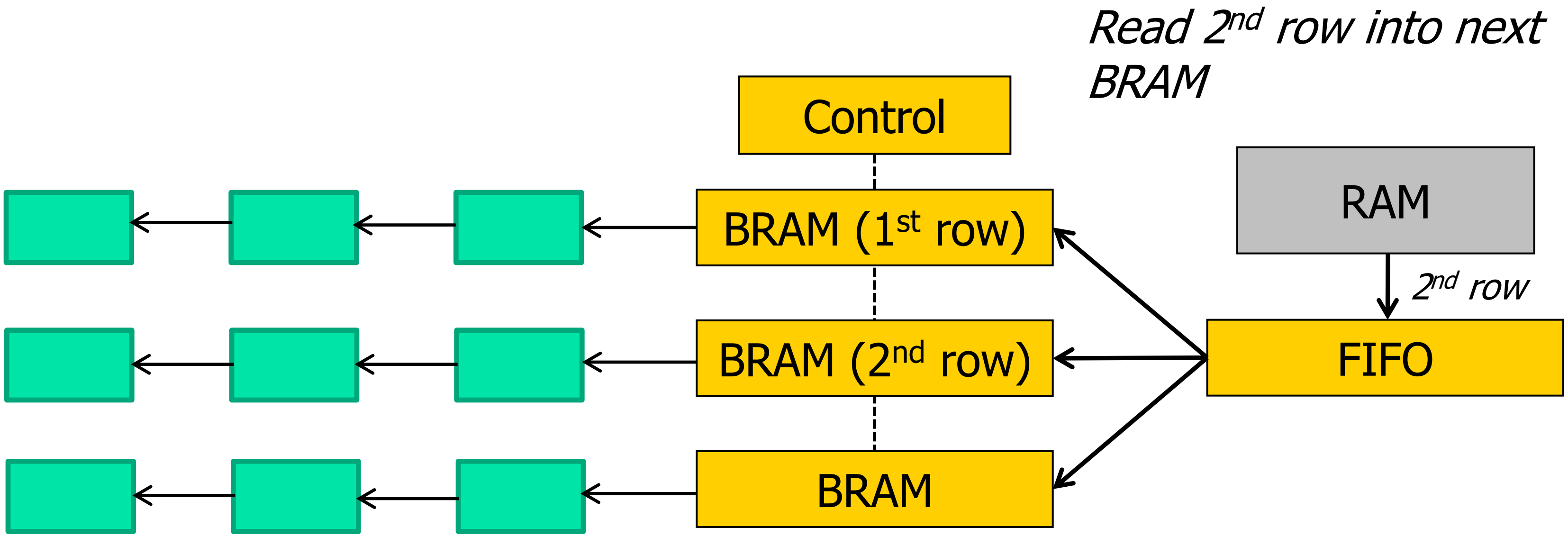


Block RAM Smart Buffers

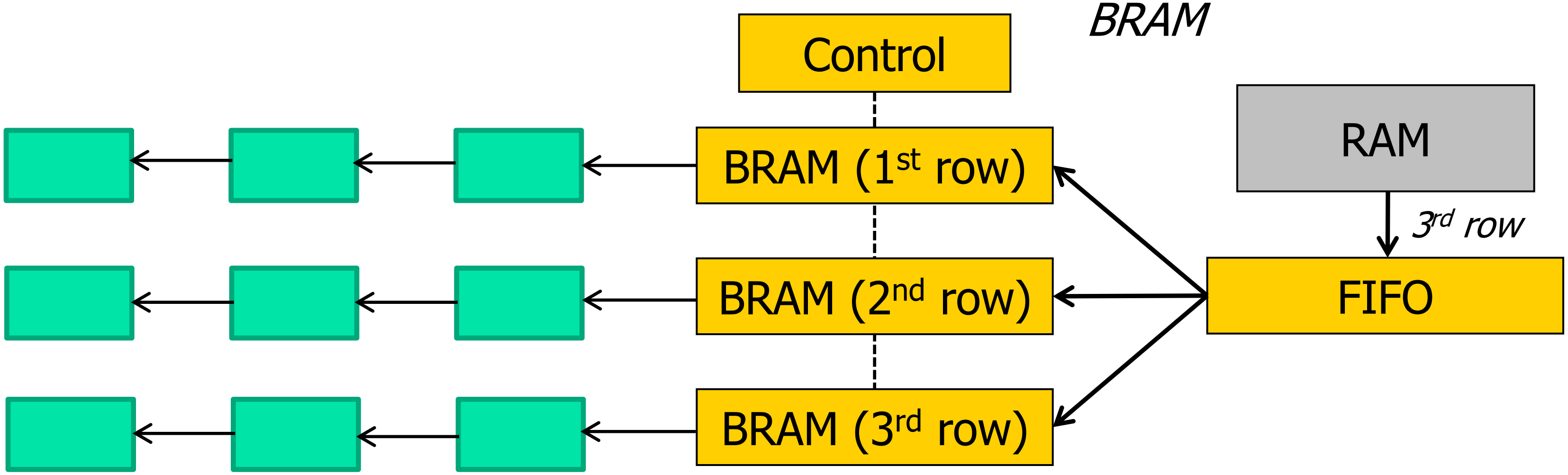
Initially, read first row into first BRAM



Block RAM Smart Buffers


Read 2nd row into next BRAM

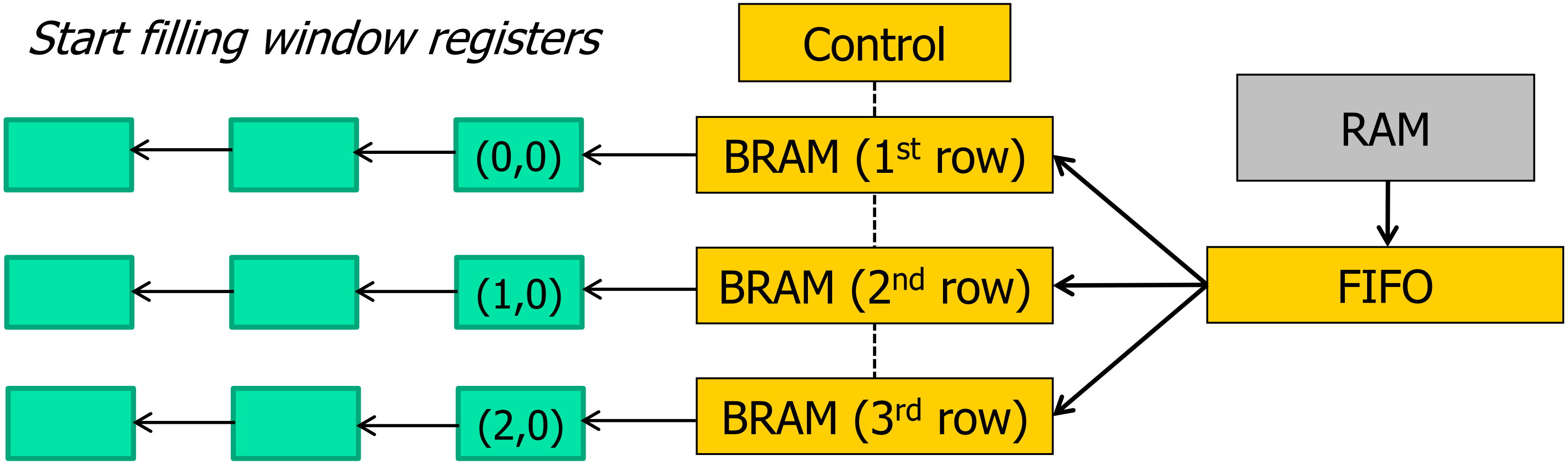
Block RAM Smart Buffers



Read 3rd row into next BRAM

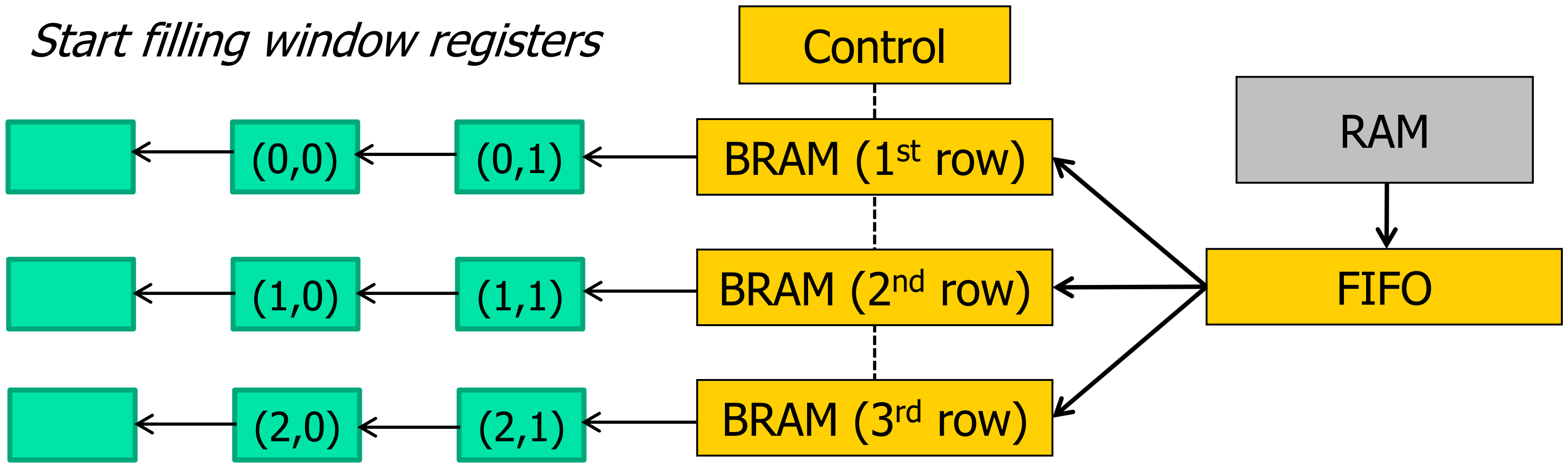
Block RAM Smart Buffers

Start filling window registers

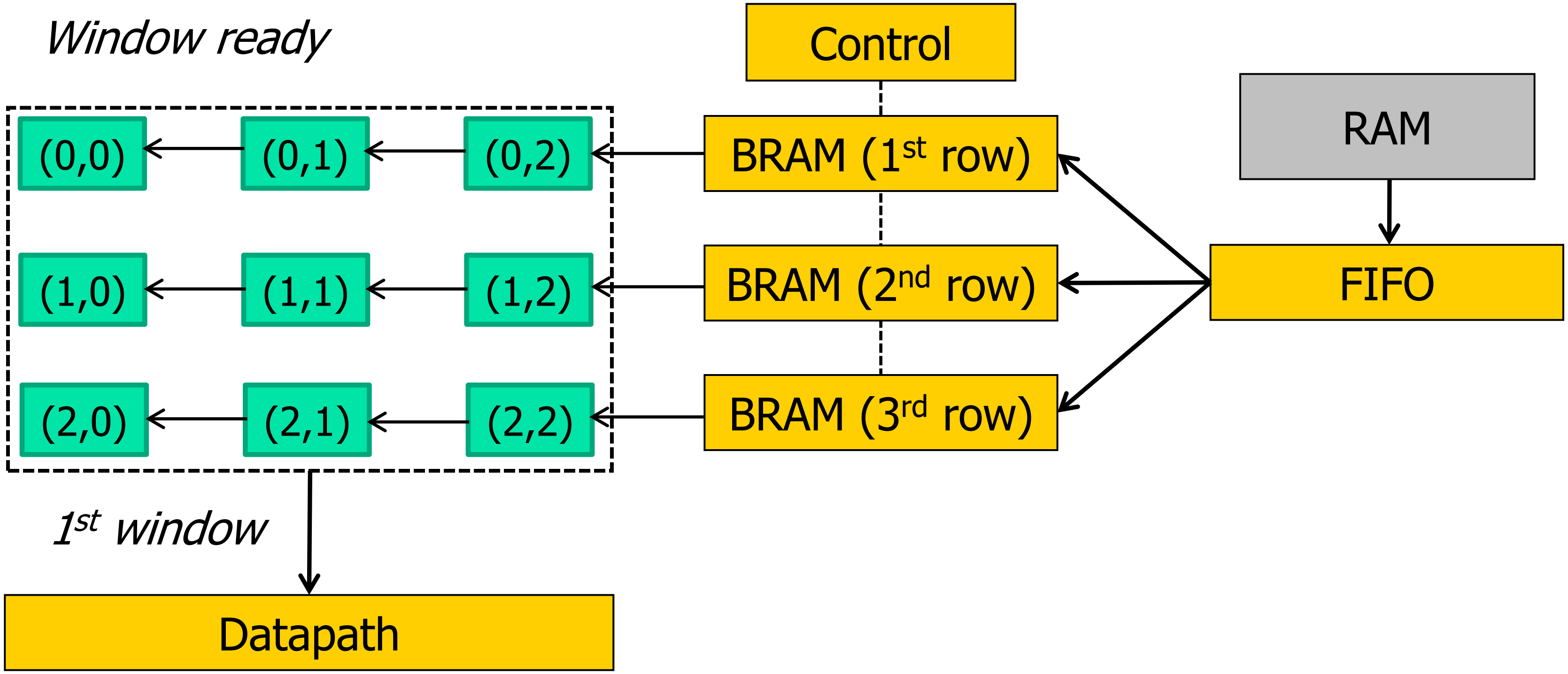


Block RAM Smart Buffers

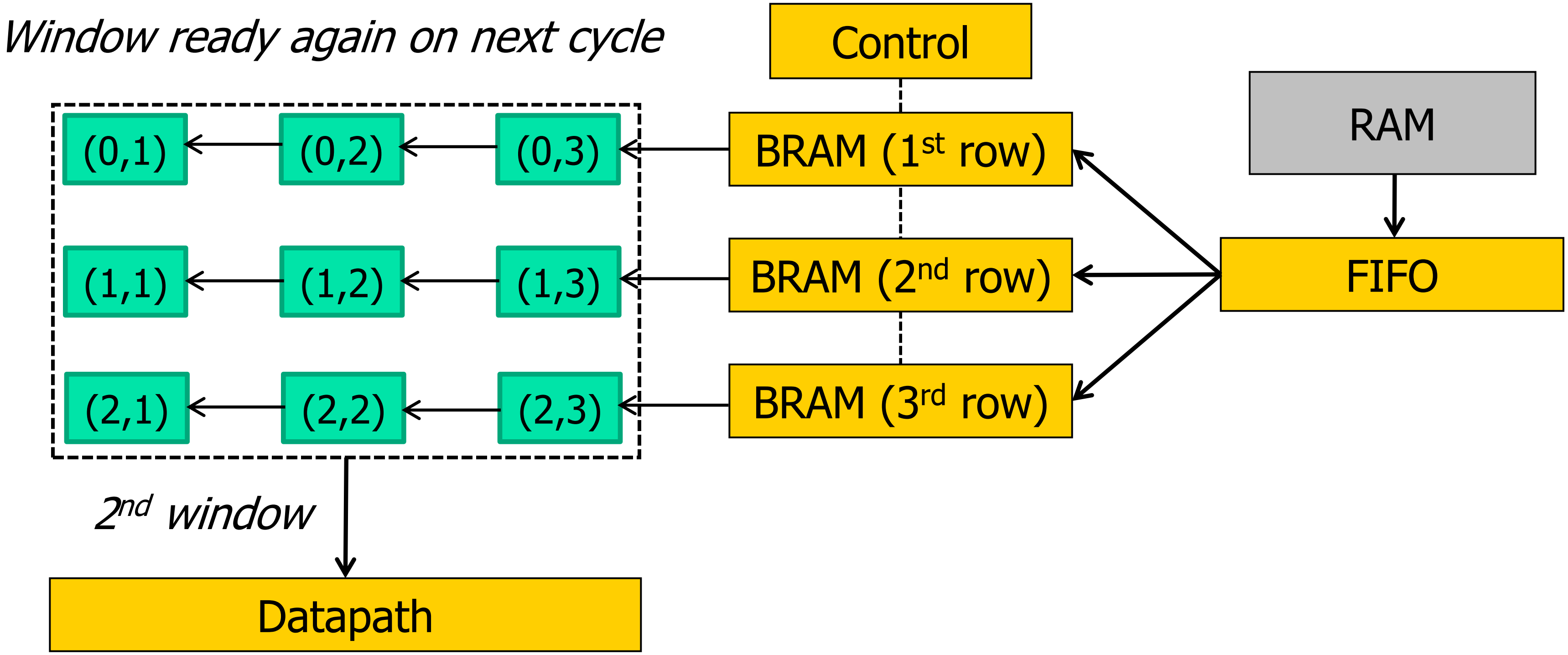
Start filling window registers



Block RAM Smart Buffers

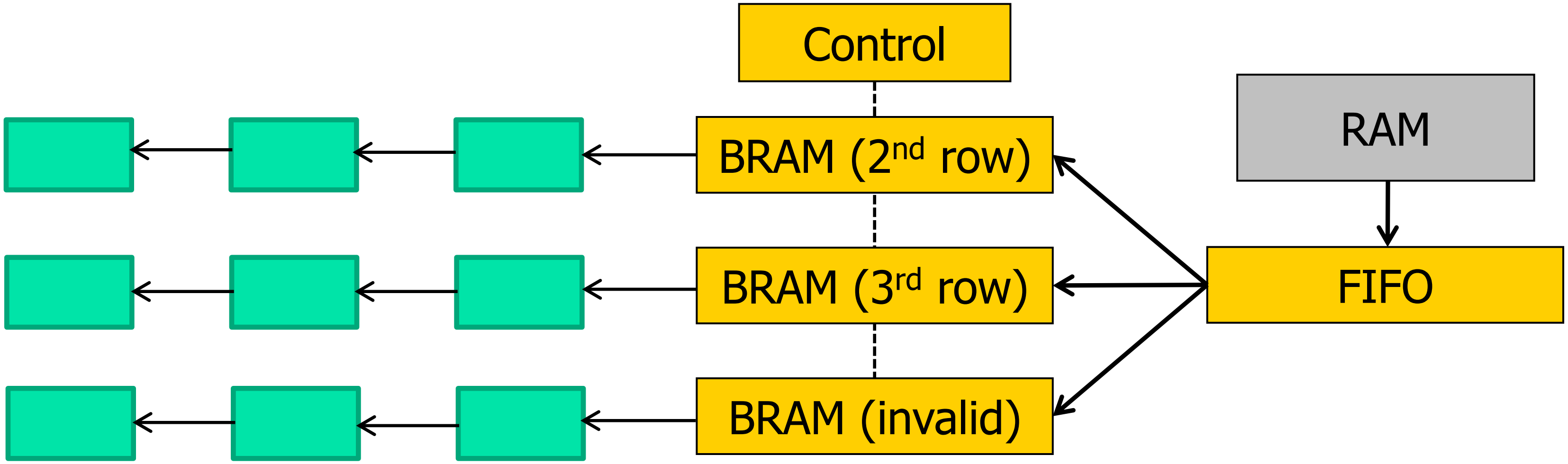


Block RAM Smart Buffers



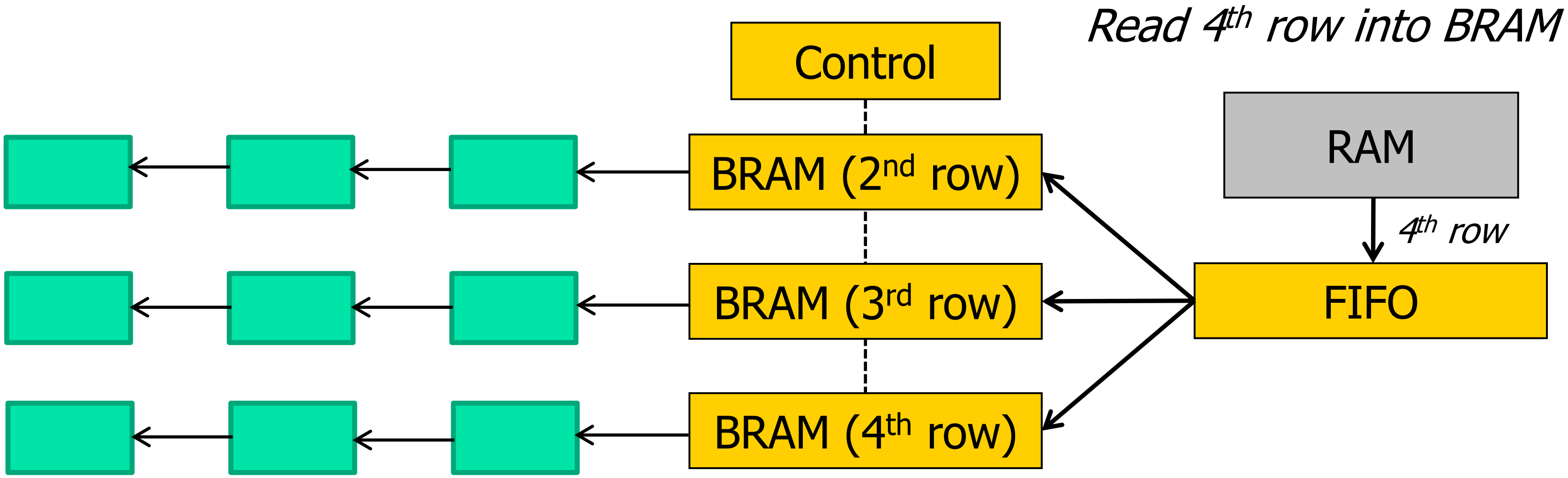
Block RAM Smart Buffers

After finishing one row, realign BRAMs with next row of windows

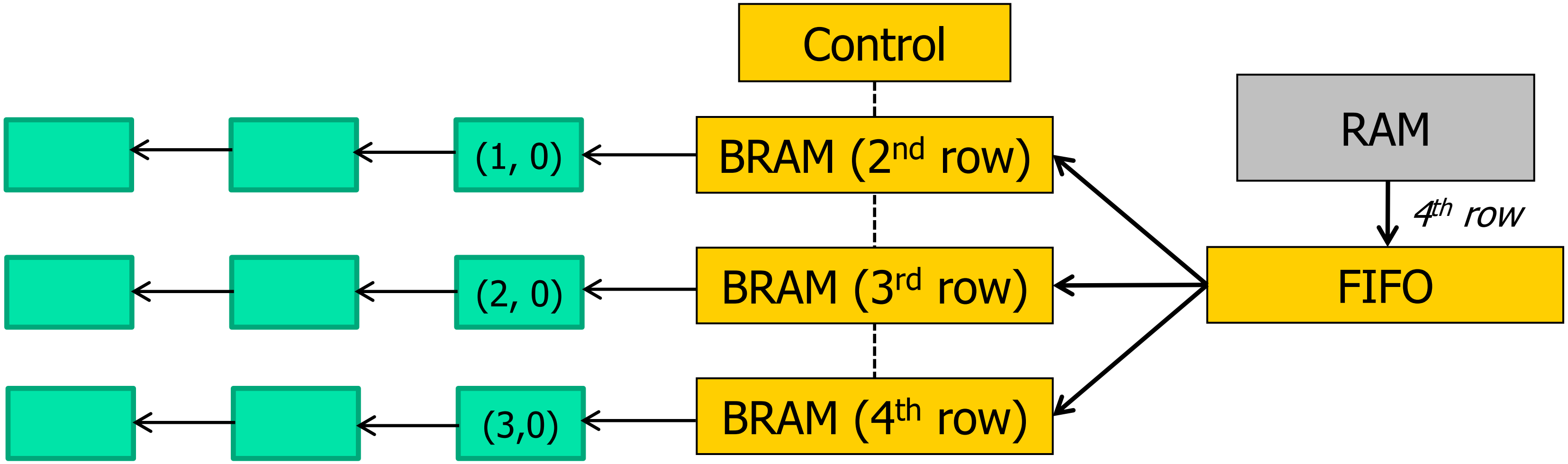


NOTE: This BRAM alignment is not done by copying data. Instead, the BRAMs are treated as a circular queue, where the front changes after each row

Block RAM Smart Buffers



Block RAM Smart Buffers



Repeat until entire input has been processed

Summary

- Smart buffers assemble an input stream into windows
  - Avoids memory accesses for reused data
  - Improves memory bandwidth, enables more unrolling
- Some final thoughts:
  - Smart buffers add latency, but greatly improve throughput
  - For 2-D smart buffers, couldn't you avoid buffering rows by reading elements from the same column of different rows?
    - Yes, but memory bandwidth would be terrible
    - Remember, many memories are efficient for sequential accesses