CO405H

Computing in Space with OpenSPL

Topic 14: Networking DFEs

Oskar Mencer **Georgi Gaydadjiev**

Department of Computing
Imperial College London

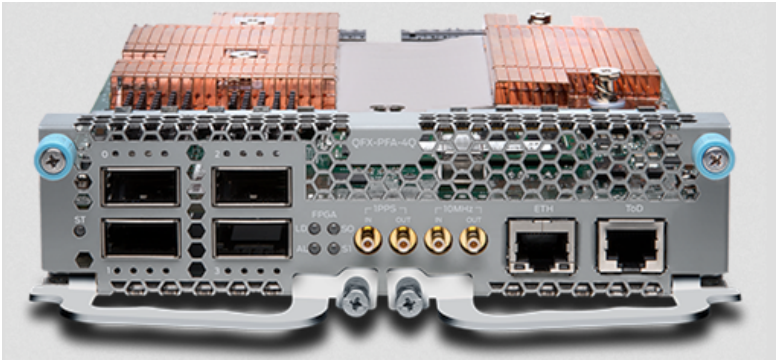
http://www.doc.ic.ac.uk/~oskar/
http://www.doc.ic.ac.uk/~georgig/

CO405H course page: http://cc.doc.ic.ac.uk/openspl16/
WebIDE: http://openspl.doc.ic.ac.uk
OpenSPL consortium page: http://www.openspl.org

o.mencer@imperial.ac.uk g.gaydadjiev@imperial.ac.uk

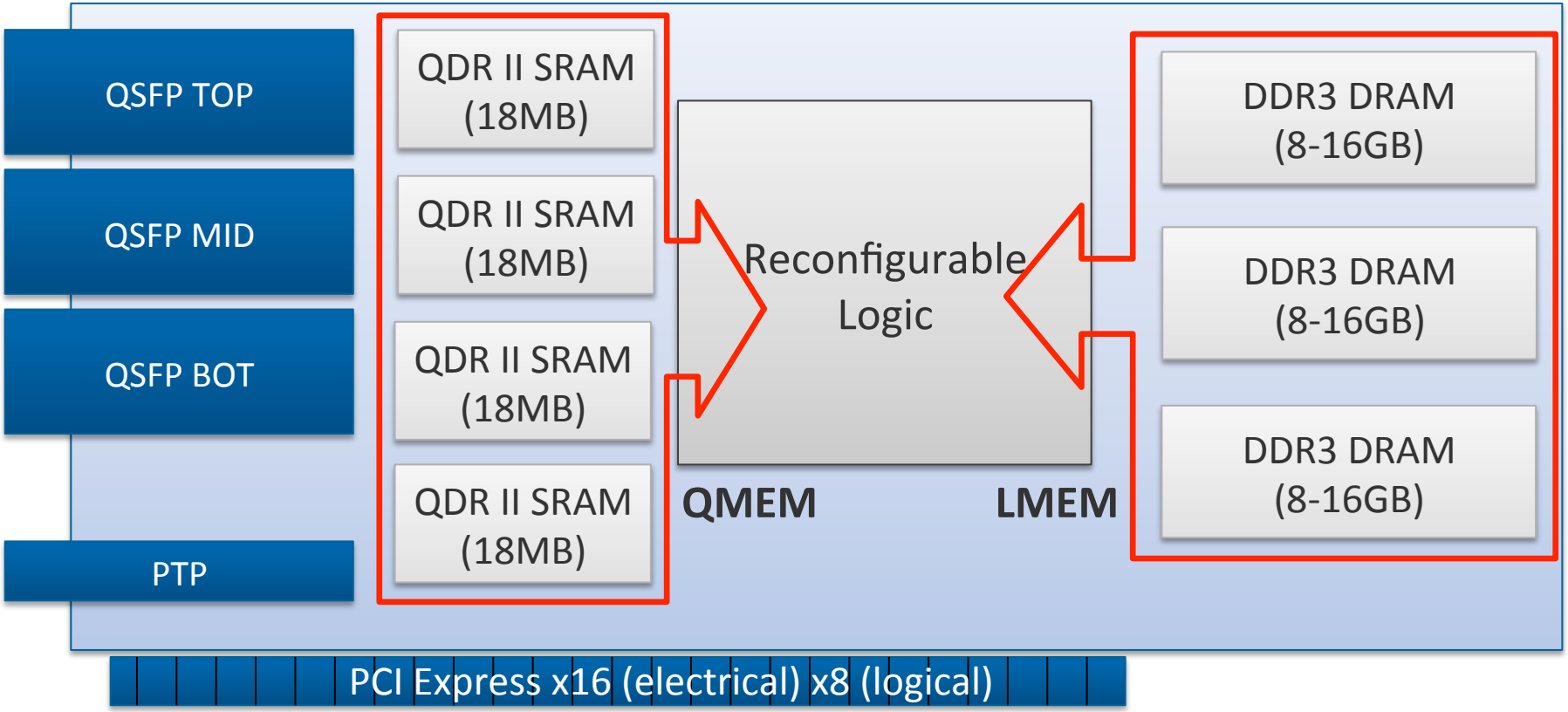
DFEs in the network

- Networks operate on data streams
- This seems like natural use scenario for DFEs
- Some network processing specific problems
- Possible solutions



The networking DFE Card

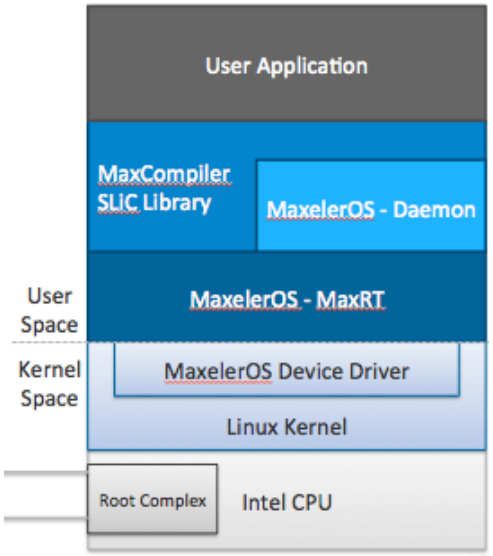
{TOP, MID, BOT}: 4 x 10G Serial Links



JDFE is pin compatible but has 8 serial links:

- 4 to the switch fabric (ports selectable via JunOS)
- 4 directly connected to the reconfigurable chip

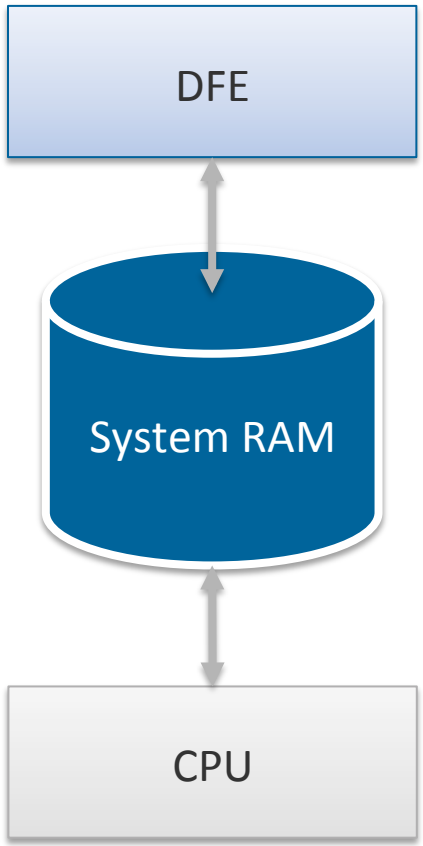
The Software Stack



- The user application is written using the SLiC API
  - Statically linked to `libslic.a`
- SLiC relies on MaxelerOS's MaxRT
  - Linked via `libmaxeleros.so` – a shared library
- MaxRT communicates with the driver
  - Driver exposes functionality through the file-system:
    - `/dev/maxeler0`
    - `/proc/maxeler/…`
  - MaxRT performs `ioctls` on the device file
  - MaxRT also uses file-operations on the `/proc/maxeler/dev0/…` files
- The driver communicates with the Hardware via Slave IO
  - Effectively Mapped Memory IO – Driver writes to a special memory address and the Hardware receives the data
- The Hardware communicates
  - Directly with the User application using DMA
    - This means the hardware accesses the System RAM directly
    - User app reads/writes from/to System RAM

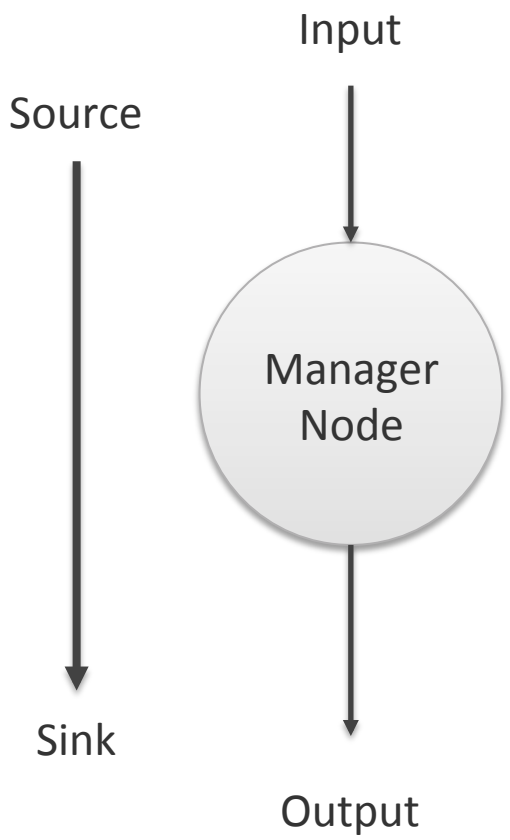
Links to/from CPU

- The manager can create a special entity for exchanging data with the CPU
- All communications with the CPU is done over PCI Express
- Generally, there is only one way to exchange data with the CPU: Memory Access
- From the CPU point of view:
  - To send data to the DFE → Write it to a special memory address (pointer)
  - To read data from the DFE → The DFE writes the data directly in to a given buffer, so the CPU simply polls that memory region and the data will appear there at some point
- From the DFE point of view:
  - Receiving data from the CPU → Data will appear at the output of a special manager block, and go over a link in to a kernel
  - Sending data to the CPU → Send the data out on a link that is connected to the special manager block



Link Interfaces (flow control)

- Links have 2 types of interfaces:
  1. PUSH
  2. PULL
- PUSH
  - Valid / Stall Semantic
- Pull
  - Read / (empty/almost empty) Semantic
- Direction of the Data determines the arrow direction:
  - Input = Data coming into the block
  - Output = Data going out

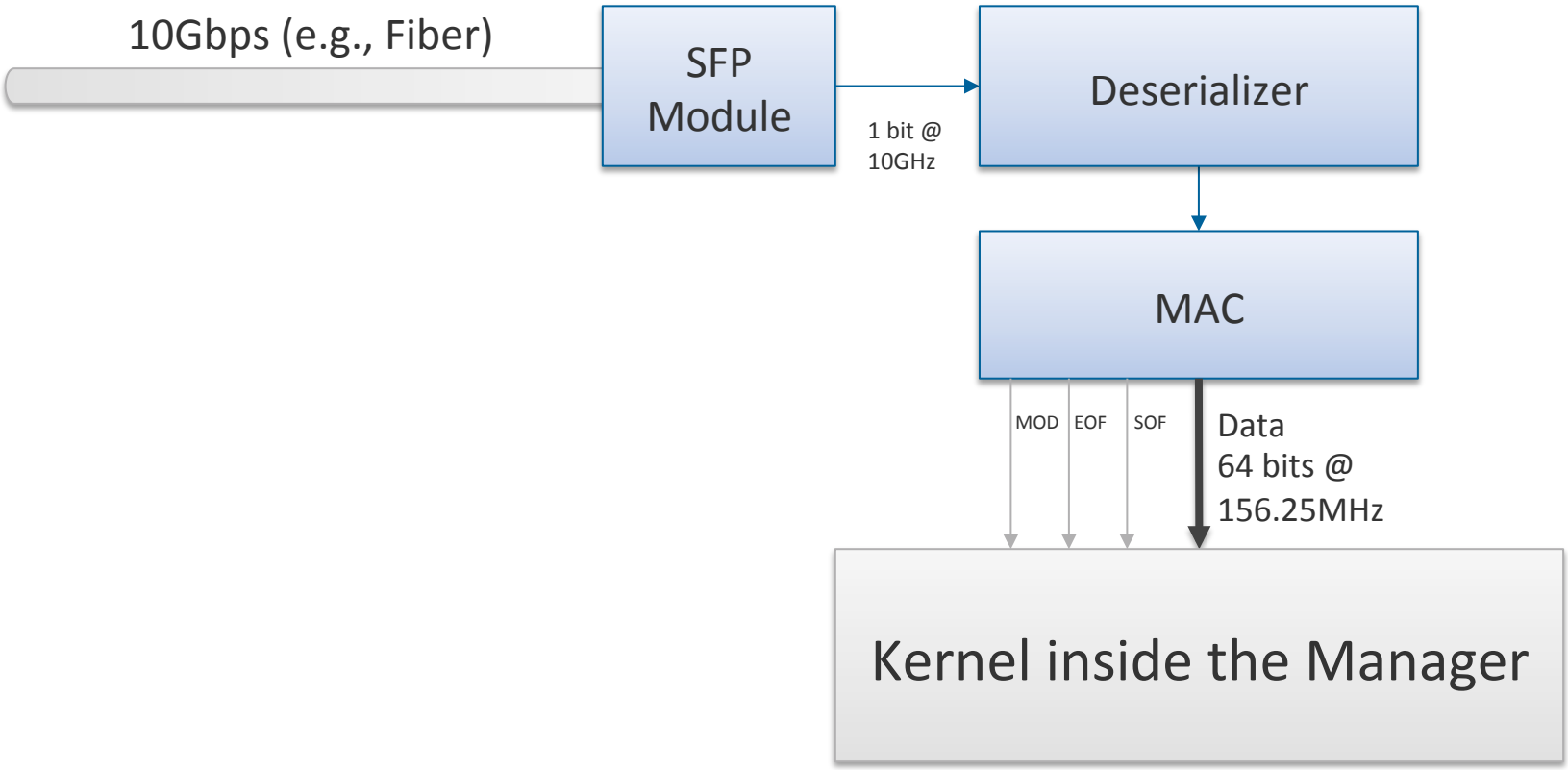


Computer Networks (Packets and Frames)

- IO connected in the Manager code
- `addEthernetStream(…)` will create all the necessary components for you to be able to receive packets from the network
- The terms Packets and Frames are used interchangeably – but we prefer the term Frame
- A frame is any data that is presented to the user along side the following metadata:
  - SOF – Start Of Frame indicator
  - EOF – End of Frame indicator
  - MOD – Number of valid bytes on the End Of Frame word

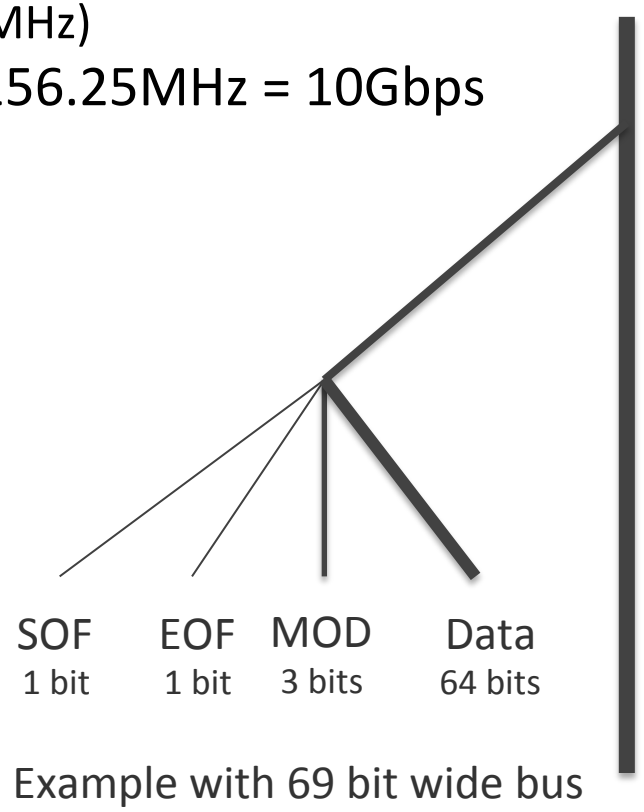
10G Network Traffic Through a Serial Link

- 10Gbps → 1 bit every 0.1ns → 64bits every 6.4ns
- 6.4ns period → 156.25MHz



Standard Ethernet Interfaces

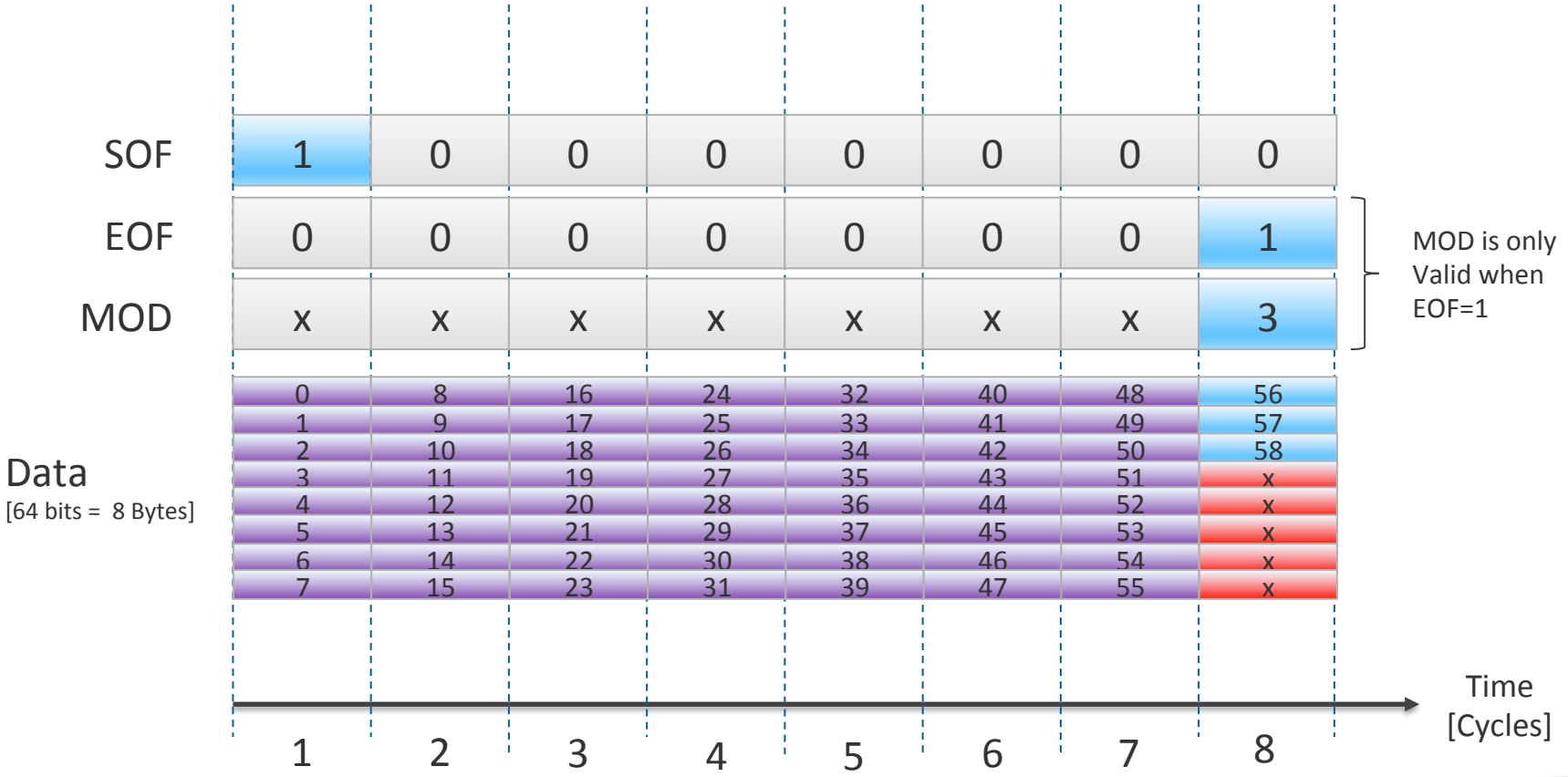
- The most commonly used 10G Ethernet interfaces vary simply by data width
  - 64 bits @ 156.25 MHz
  - 32 bits @ 312.5 MHz (It's really 322.265625MHz)
- This is because: 32b * 312.5MHz = 64b * 156.25MHz = 10Gbps
- 64-bit interface:
  - Data = 64 bits
  - SOF=1 bit
  - EOF=1 bit
  - MOD = BitsToAddress(64/8) = 3 bits
- 32-bit interface:
  - Data = 32 bits
  - SOF=1 bit
  - EOF=1 bit
  - MOD = BitsToAddress(32/8) = 2 bits



Example with 69 bit wide bus

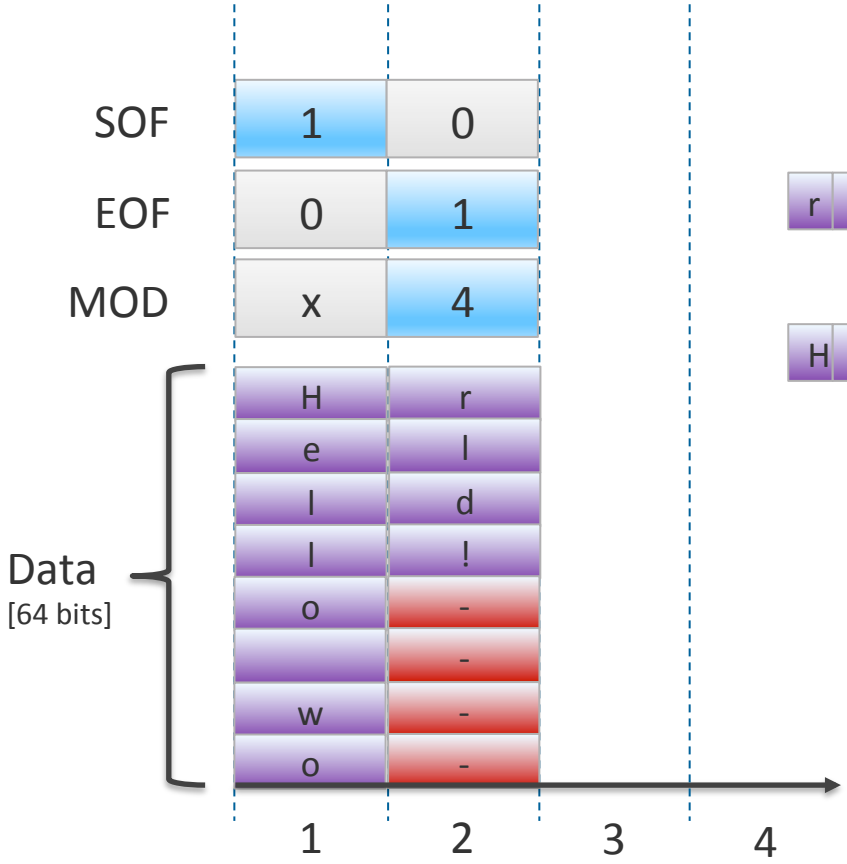
Network Traffic: User Point of View

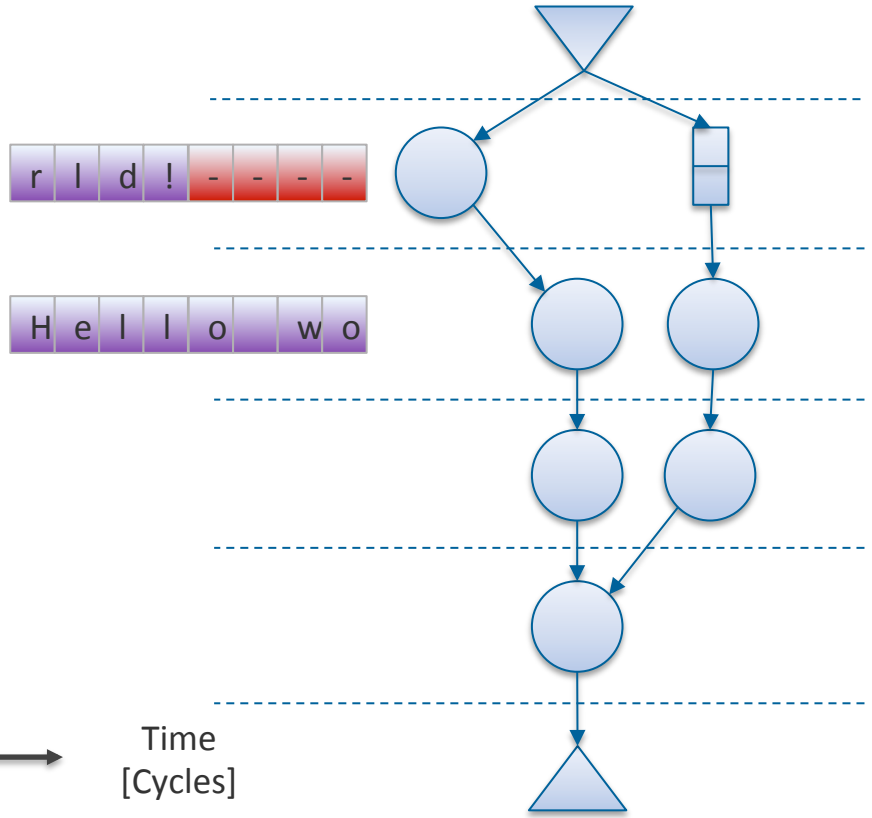
- Example of a 59 byte frame over a 64 bit link running at 156.25MHz
  - MOD indicates how many bytes are valid at the last word of the frame



“Hello World!”

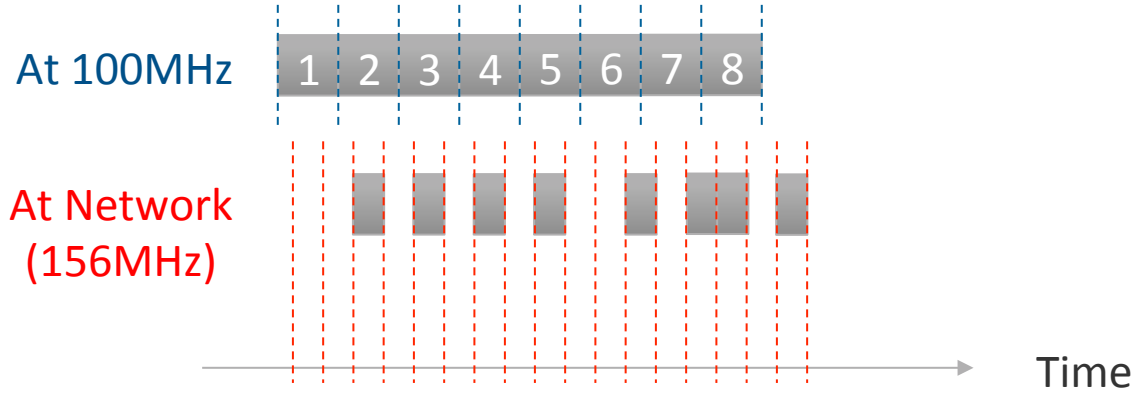
1	2	3	4	5	6	7	8	9	10	11	12
H	e	l	l	o		W	o	r	l	d	!





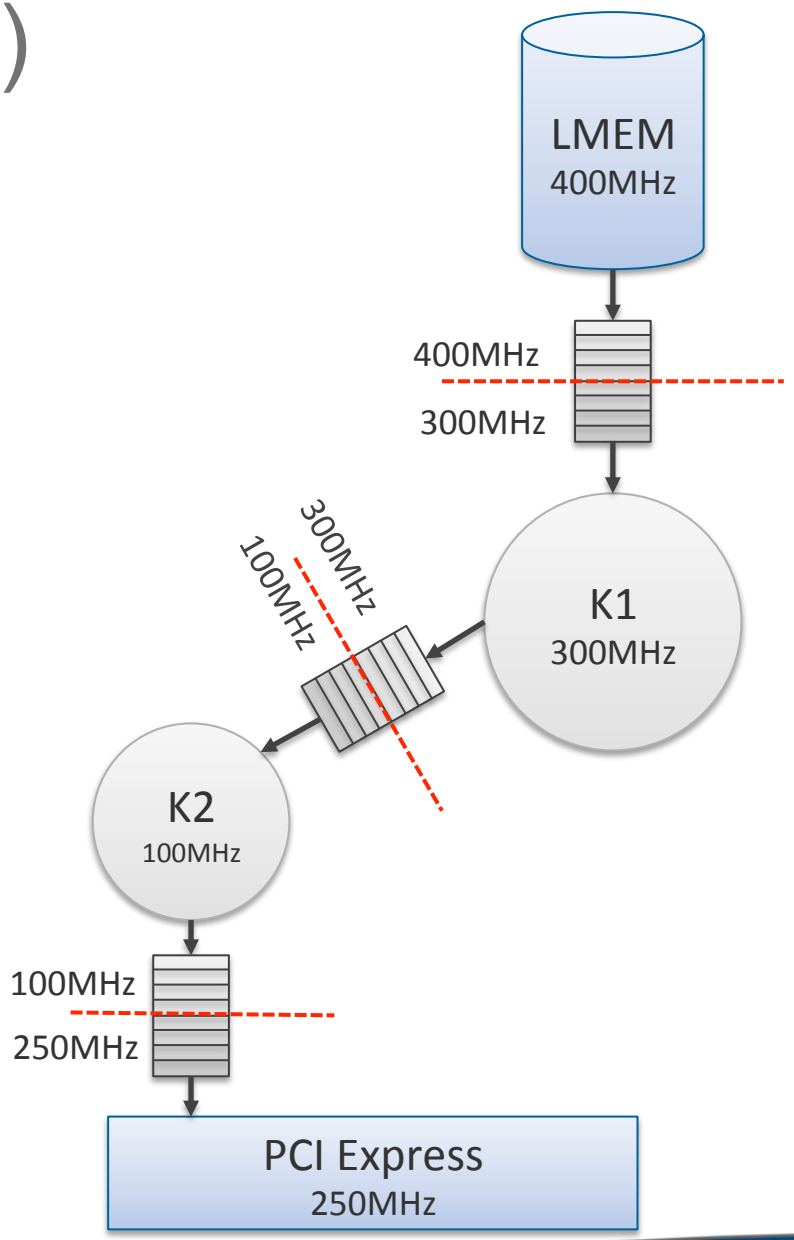
Clock Domains

- Simply put: A set of blocks in which data can move at a certain rate
- Typical Clock Domains:
  - PCI Express (Gen 2.0 x8) – 250MHz
  - Network (64 bits) – 156MHz
  - LMEM – 400MHz
  - QMEM – 550MHz
  - Stream Clock (default) – 100MHz
- The higher the clock frequency, the faster data can move
- Data can move between clock domains
  - it might not appear contiguous at the destination clock domain



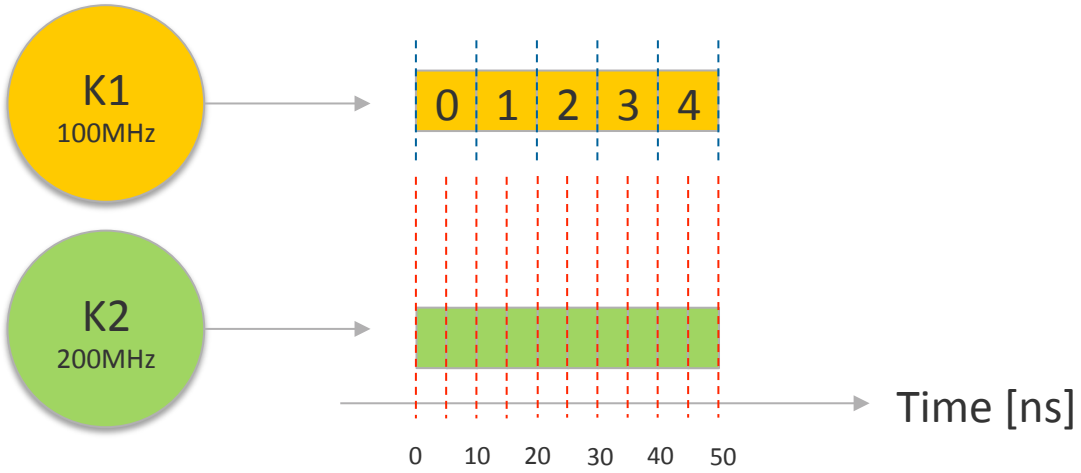
Clock Domains (cont)

- Every Manager Block is associated with a Clock Domain
- Some blocks, like Kernels and State Machines are flexible and can be assigned to a specific clock domain by the User
- When different blocks that belong to different Clock Domains are connected – the Manager automatically inserts a Dual-Clock FIFO to help with the domain transition



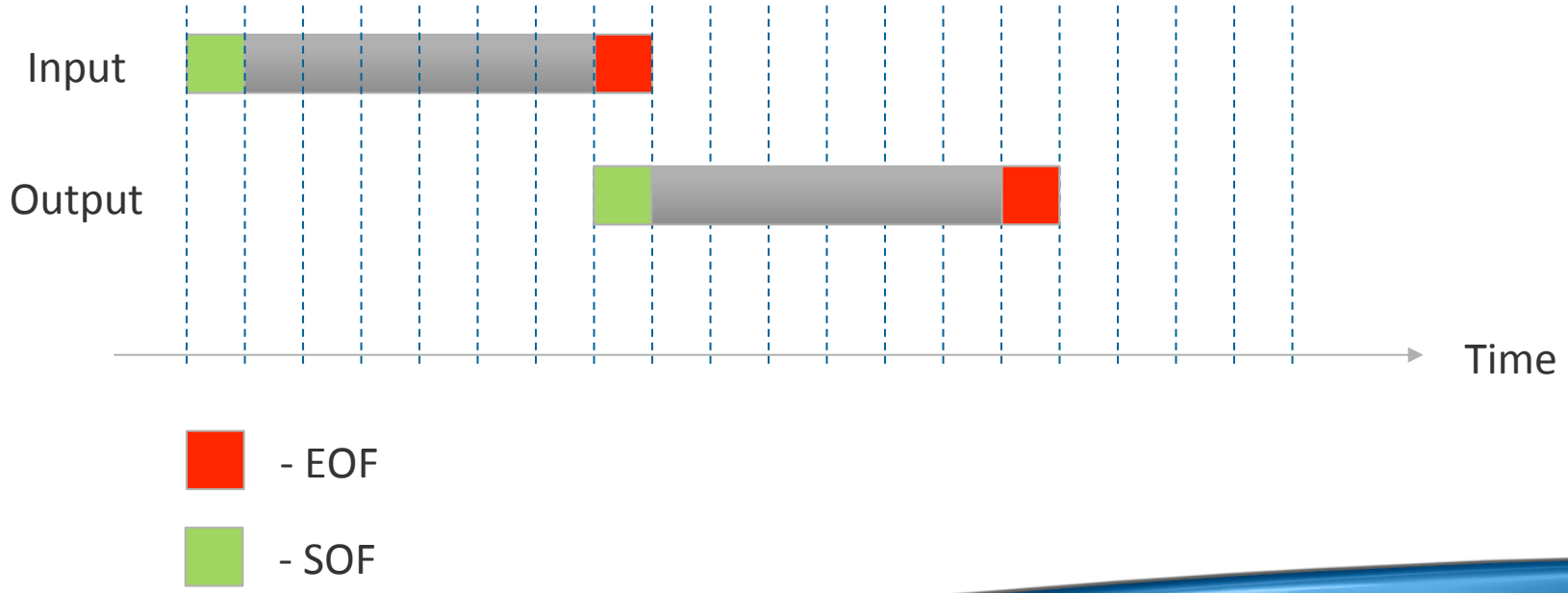
Clock Domains and Throughput

- A Kernel that generates data on every clock cycle
- For a fixed length of time: The higher the clock frequency, the more data the kernel will generate



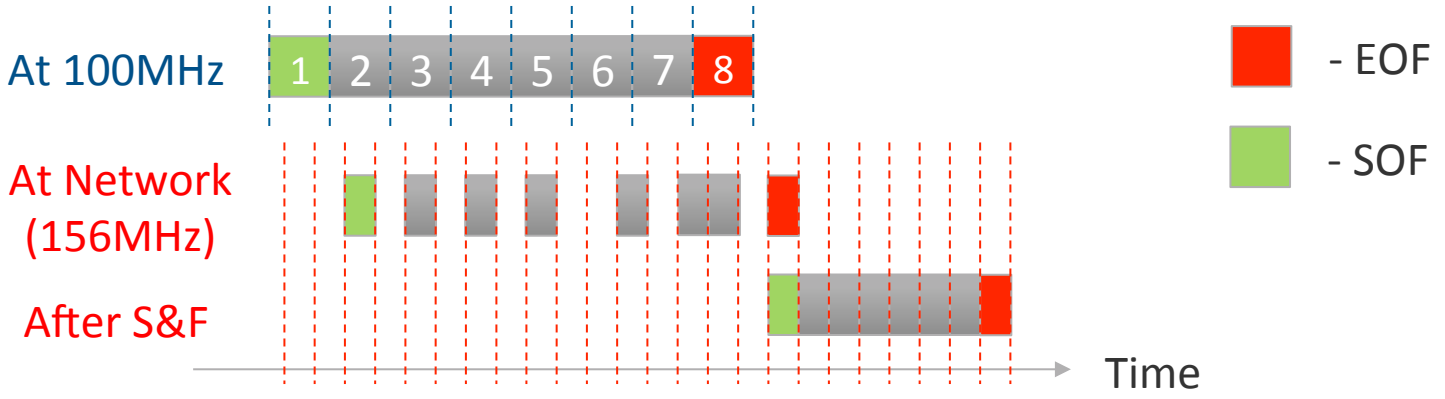
Store and Forward

- The act of storing a complete frame before forwarding it downstream for further processing
  - Common for Checksum verification
  - Less common: Clock-domain transitioning



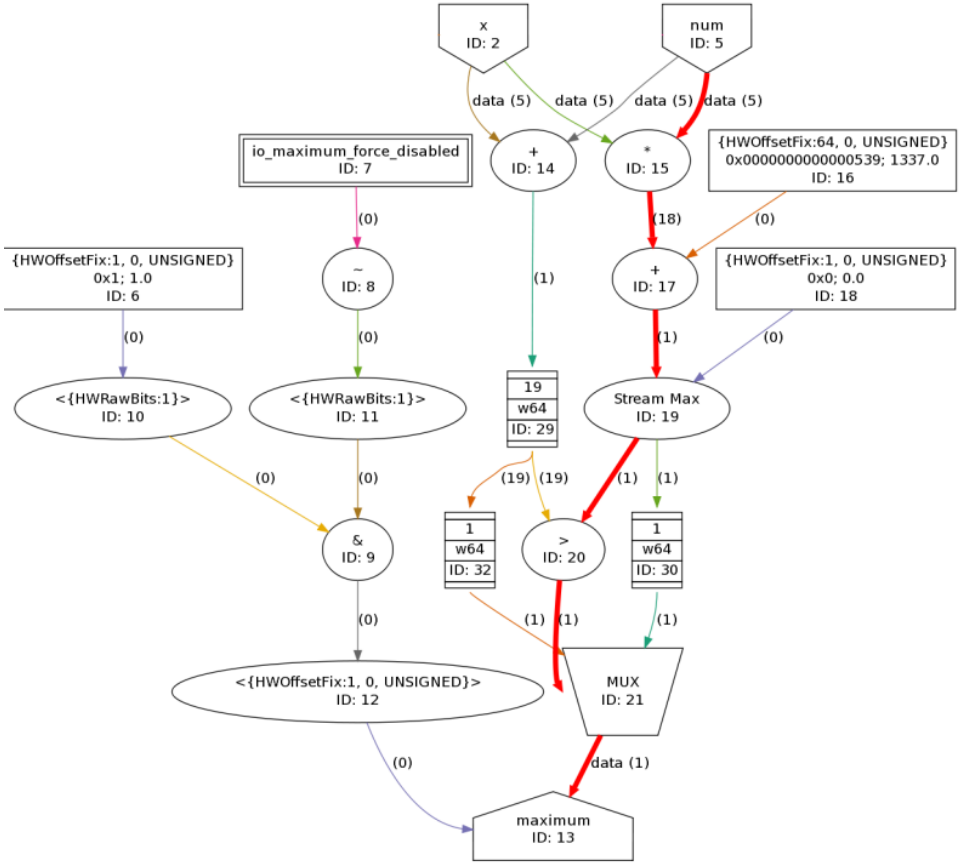
Store and Forward – Frame Continuity

- A store and forward block can convert a non-continuous frame in to a continuous one
- This property is important when connecting directly to Ethernet MACs – since those can only work with continuous frames



Kernel Latency

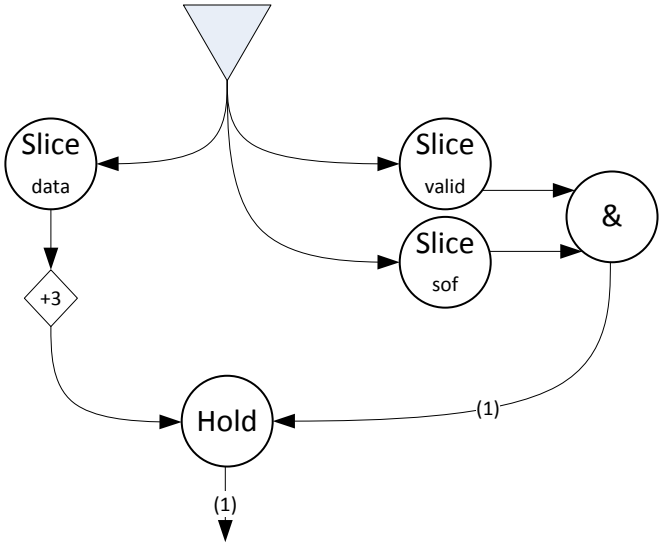
- The pipeline depth from a specific input to a specific output
- For HPC applications - Typically in the 1,000s
- For Networking applications – Typically in the 100s
- For Ultra-Low-Latency applications – Typically in the 10s



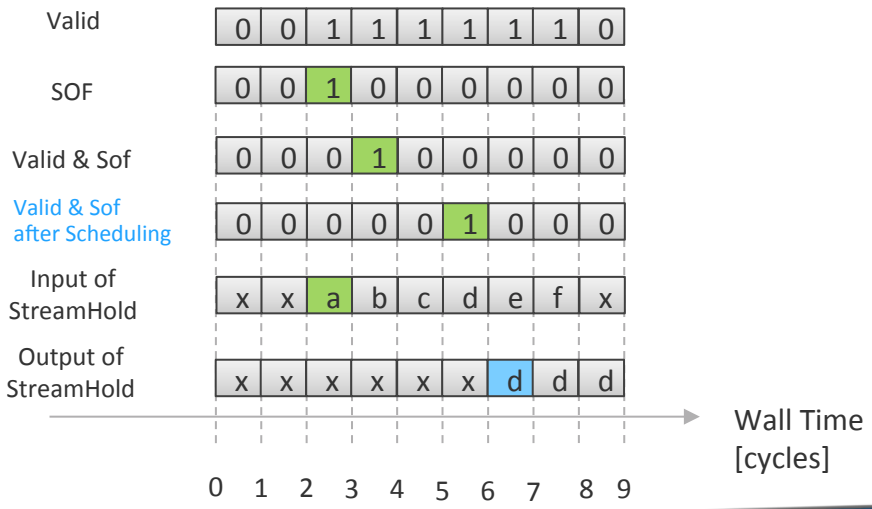
The numbers in (brackets) are the node latencies

Stream Hold

- Data flowing through a graph will only be valid at certain times
- If we're interested in the 4th data item relative to the start of frame, we can store it for later use by using a stream hold
- The streamHold will only remember the value that was at its input when valid & sof = true
- Only when valid&sof = true, the stream.offset(data, 3) was interesting

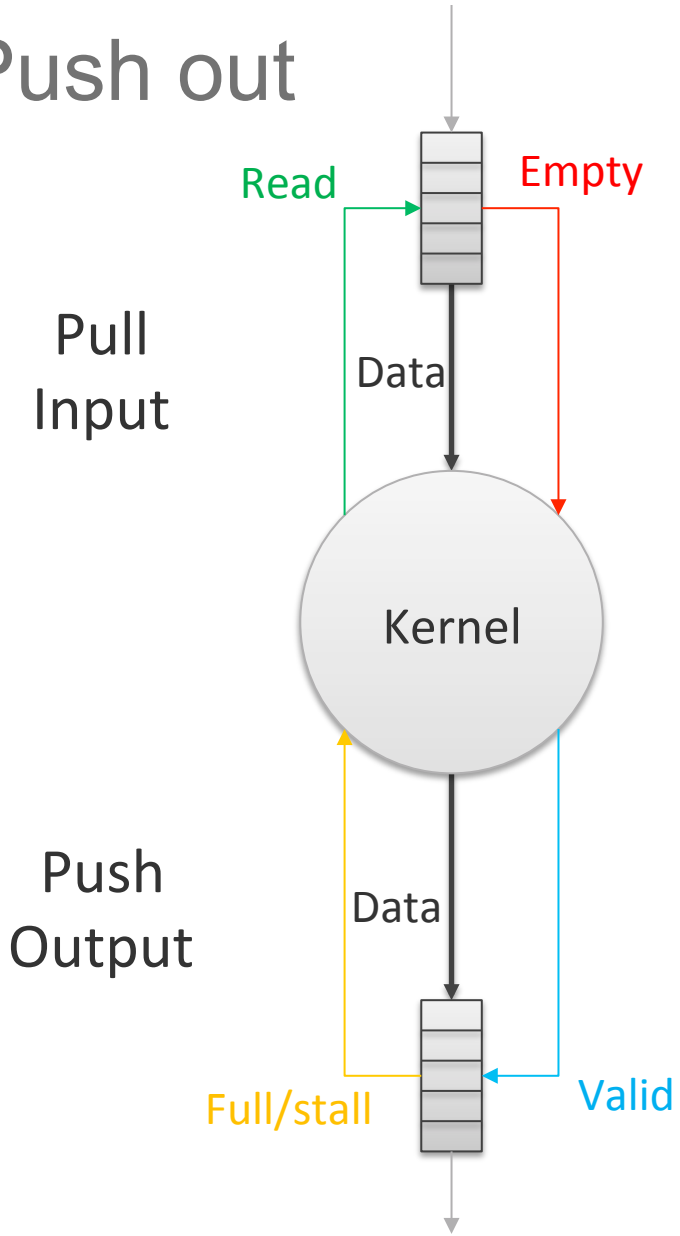



```
streamHold(stream.offset(data, 3), valid & sof)
```



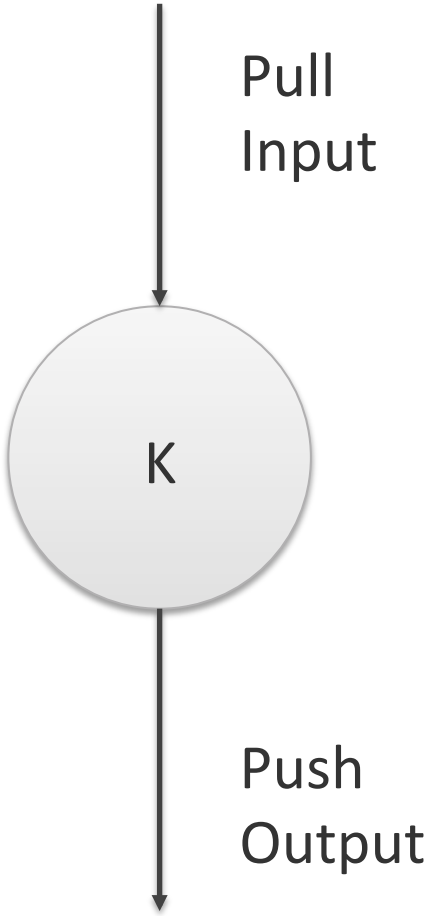
Kernels behavior: Pull in, Push out

- PULL type inputs
- PUSH type outputs
- FIFO queues at each input and output of a kernel
- FIFOs can normally hold up to 512 data words before being “full”



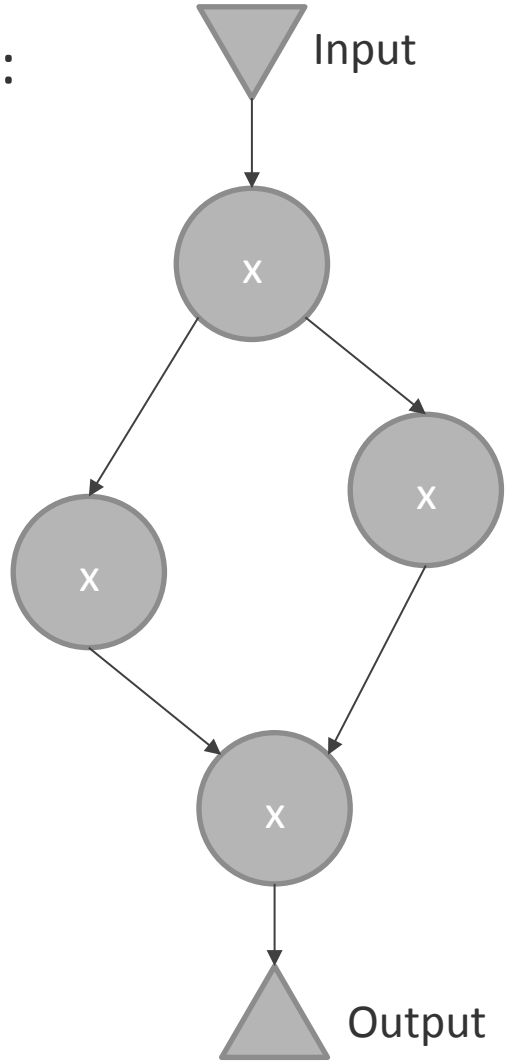
Kernel Flow Control - Stalling

- Input: Read attempt but there is no data available
  - Kernel will stop everything until there is more data!
- Output: Write attempt but the output buffer is full
  - Kernel will stop everything until there is more space!



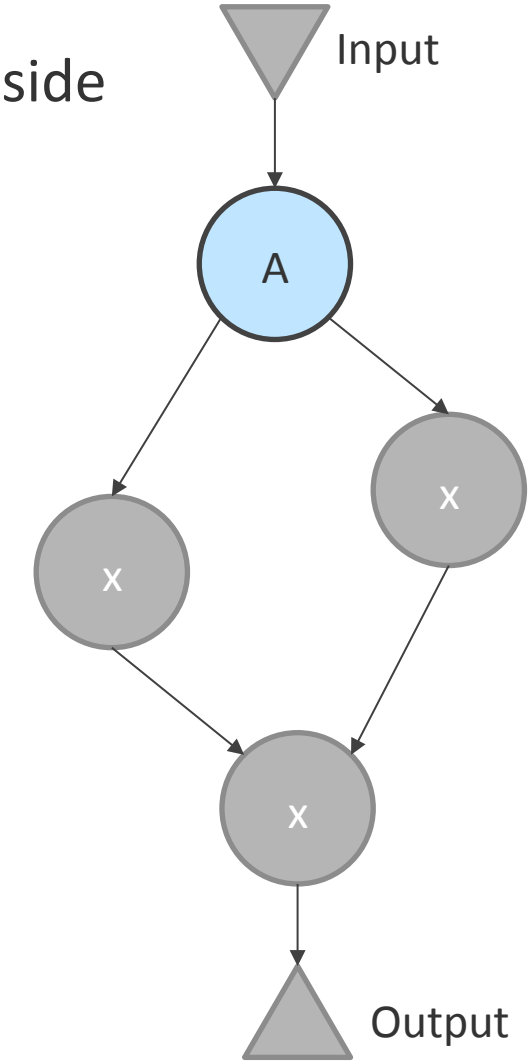
Kernel Flushing

I have only 2 data Items:
A and B



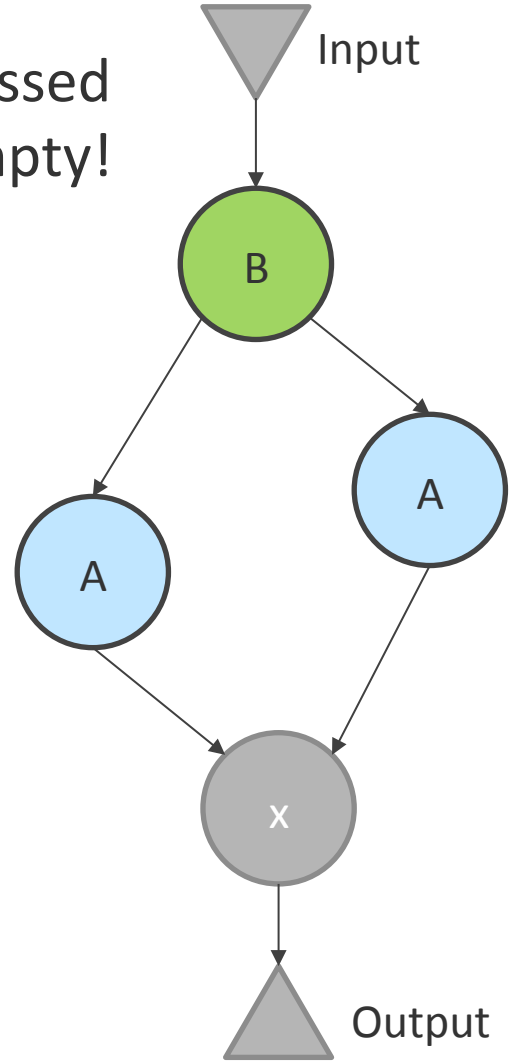
Kernel Flushing

A goes in, B is still outside



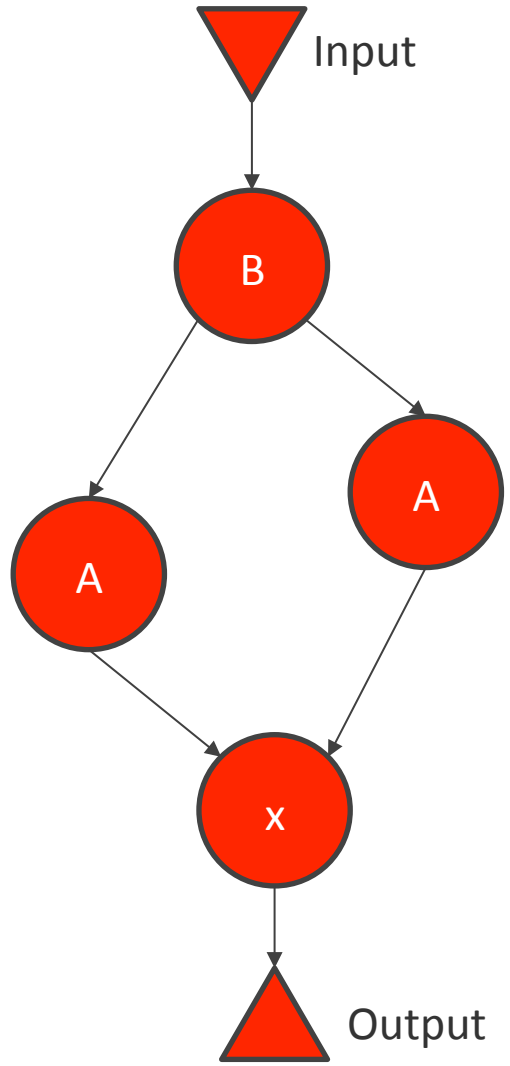
Kernel Flushing

Both A and B are processed
Input has now gone empty!



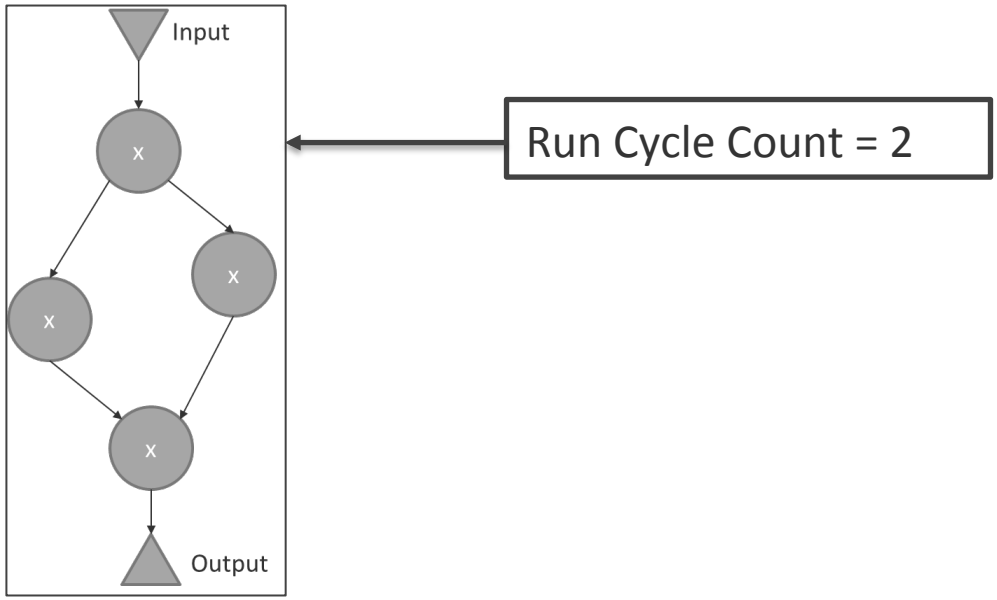
Kernel Flushing

A problem!
No more inputs –
Kernel is stalling!



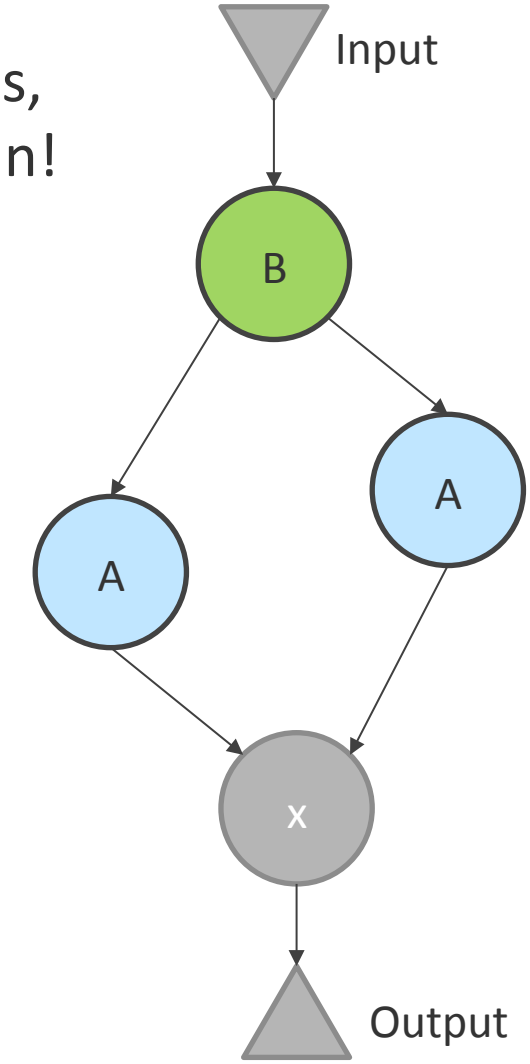
How long will a kernel run for?

- We normally specify the number of expected data items
- This is called the RunCycleCounter and it's part of the kernel's Flushing Logic



Kernel Flushing – With Cycle Count = 2

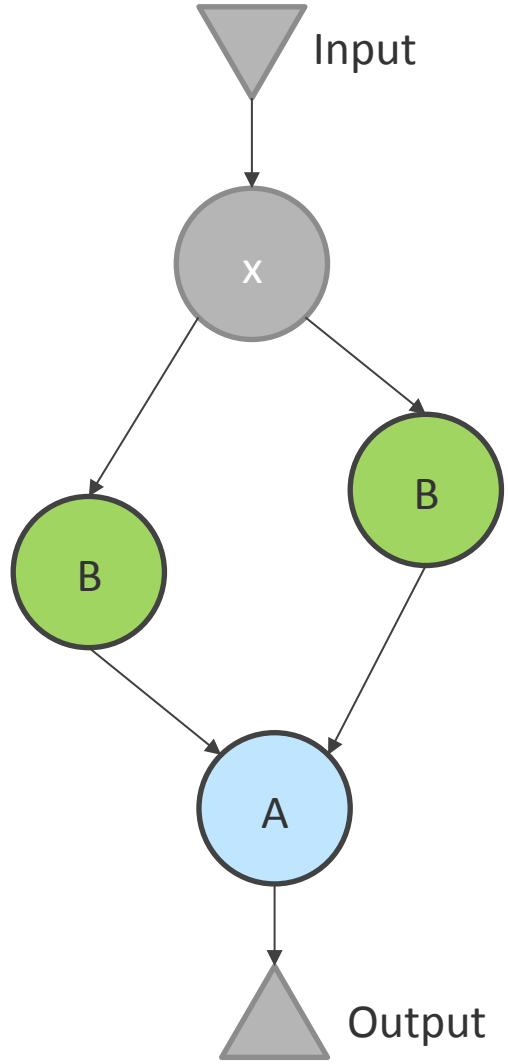
We got 2 data items,
Flushing logic kicks in!



Run Cycle Count = 2

Kernel Flushing – With Cycle Count = 2

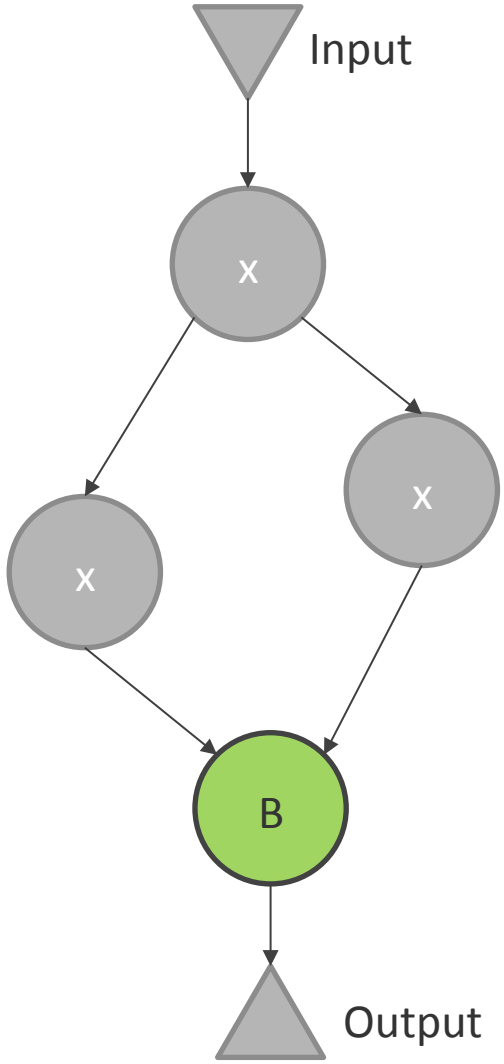
Flushing….



Run Cycle Count = 2

Kernel Flushing – With Cycle Count = 2

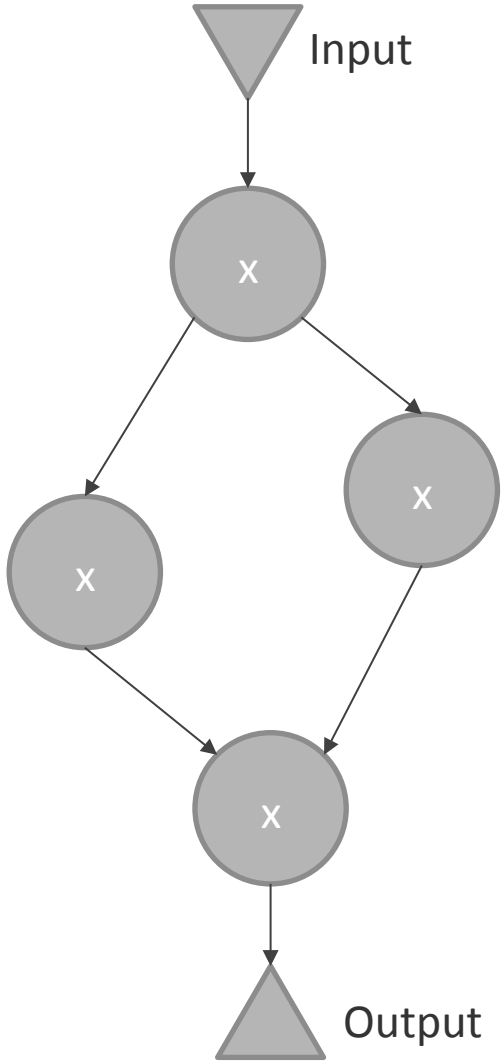
Still Flushing….



Run Cycle Count = 2

Kernel Flushing – With Cycle Count = 2

And we're done!



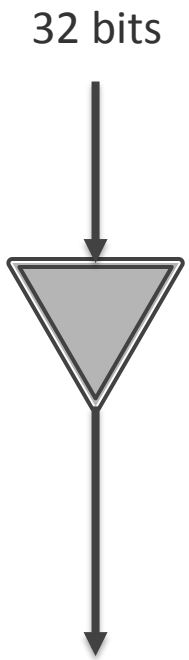
Run Cycle Count = 2

There is, however, a problem …

- This doesn't work with networking
- How many data items will the kernel expect?
  - Unknown
  - We might receive 1. We might receive 2. An infinite amount or nothing at all!
- What do we do?

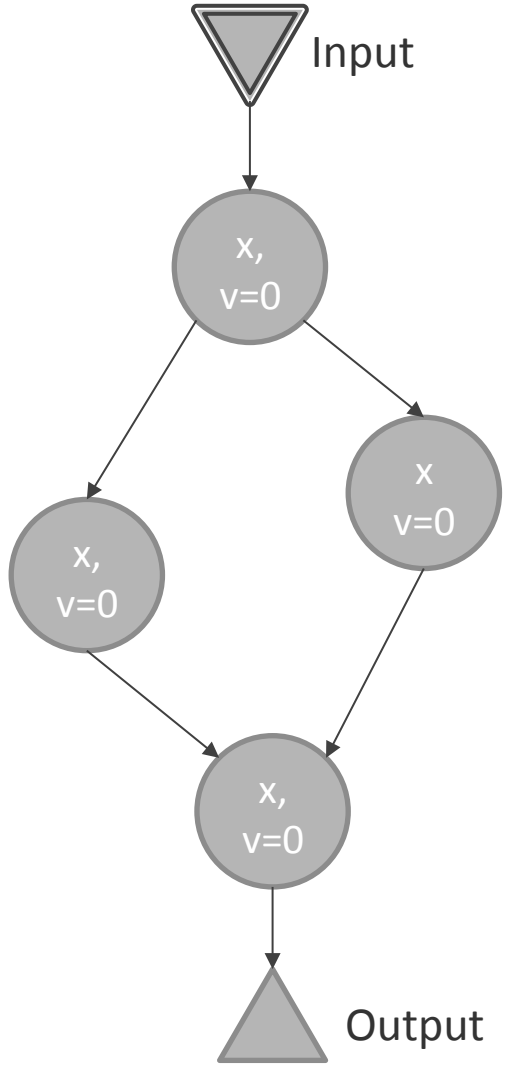
Non-Blocking Inputs

- Non-blocking inputs solve this problem
- They never stall the kernel – even when there's no data available!
- They do this by adding a Valid bit to every incoming word

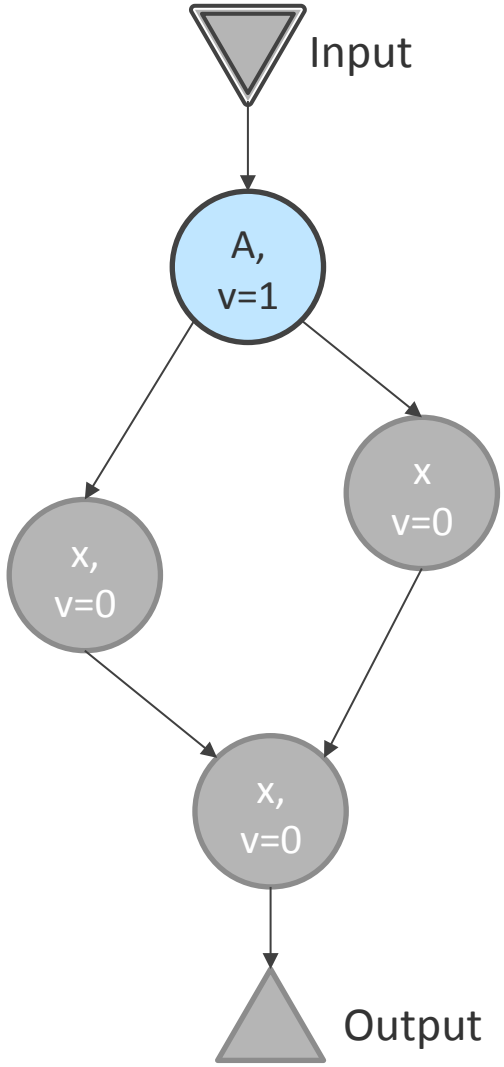


33 bits =
32 bits + 1 valid bit

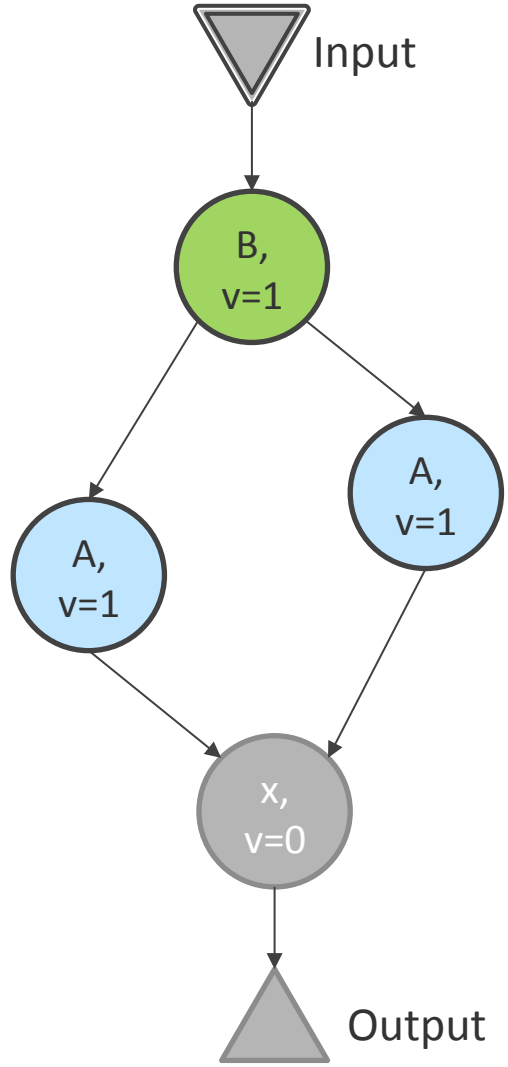
Kernel Flushing – Non-Blocking Input



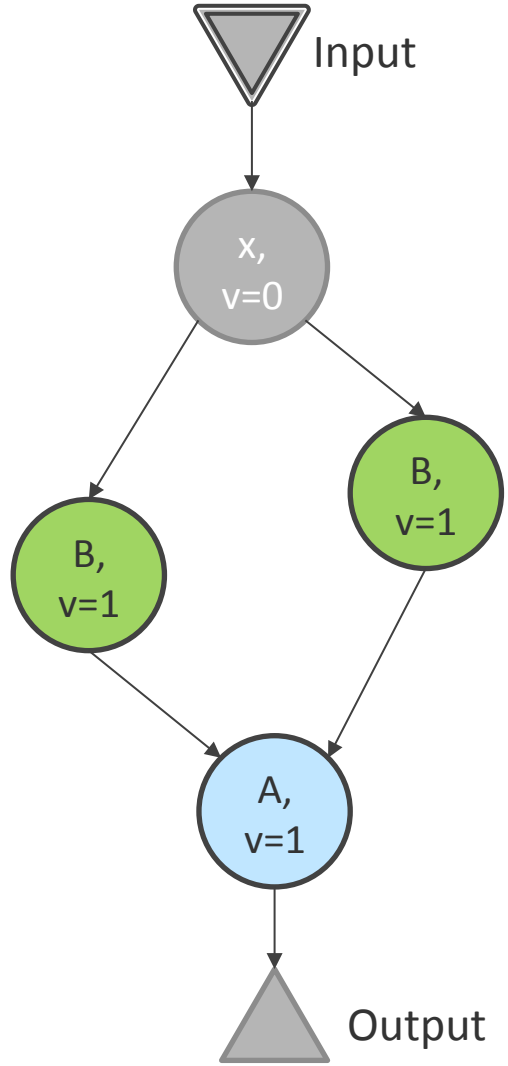
Kernel Flushing – Non-Blocking Input



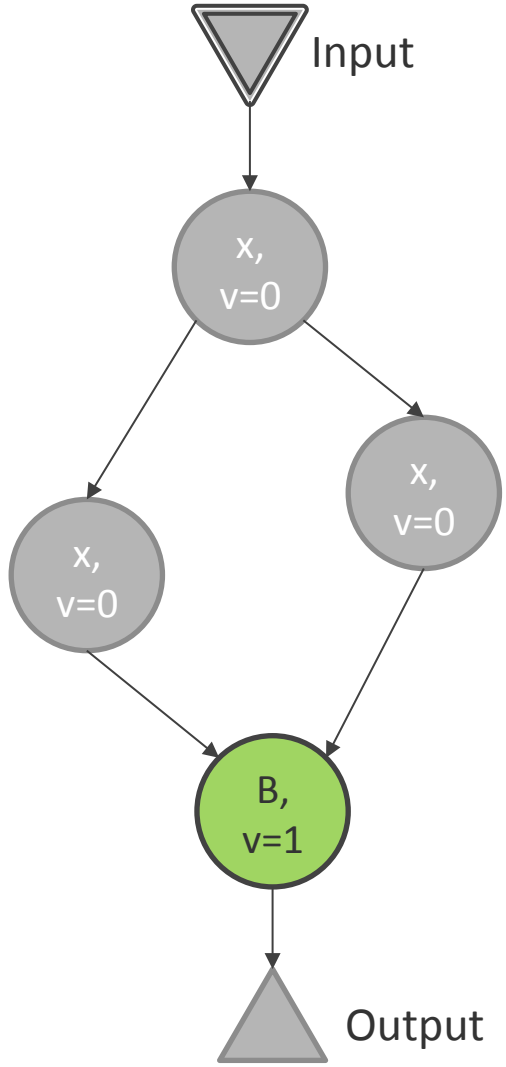
Kernel Flushing – Non-Blocking Input



Kernel Flushing – Non-Blocking Input

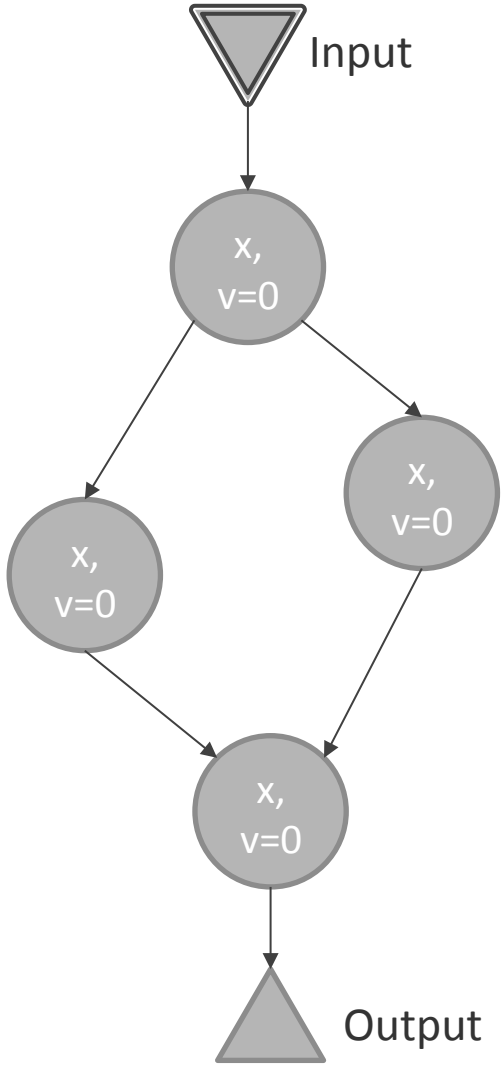


Kernel Flushing – Non-Blocking Input



Kernel Flushing – Non-Blocking Input

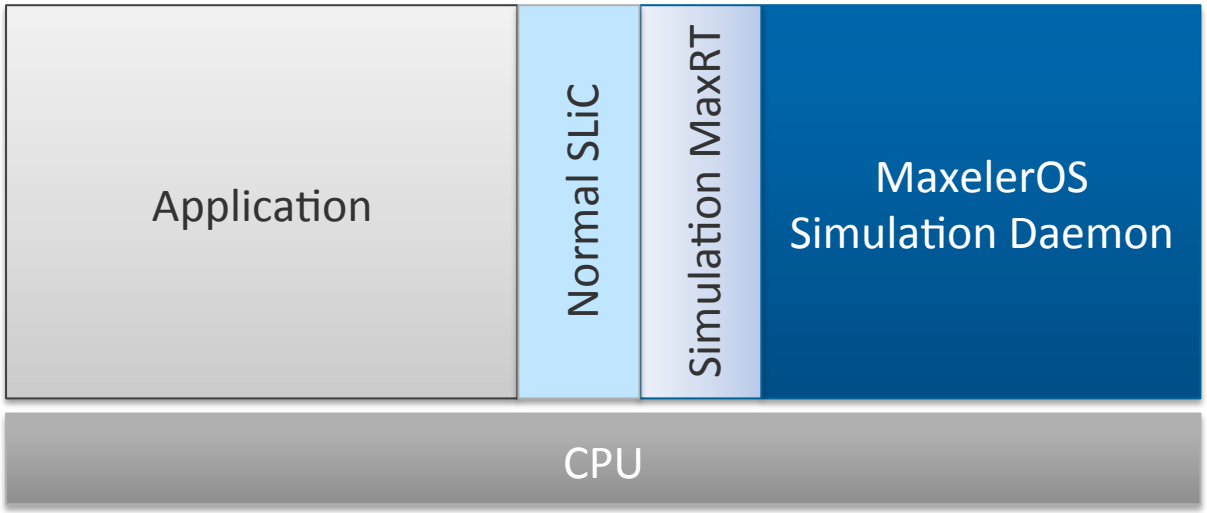
The kernel keeps running forever with valid=0 until more real data arrives!



Things will furfure improve with Custom Kernels, a new class currently under development

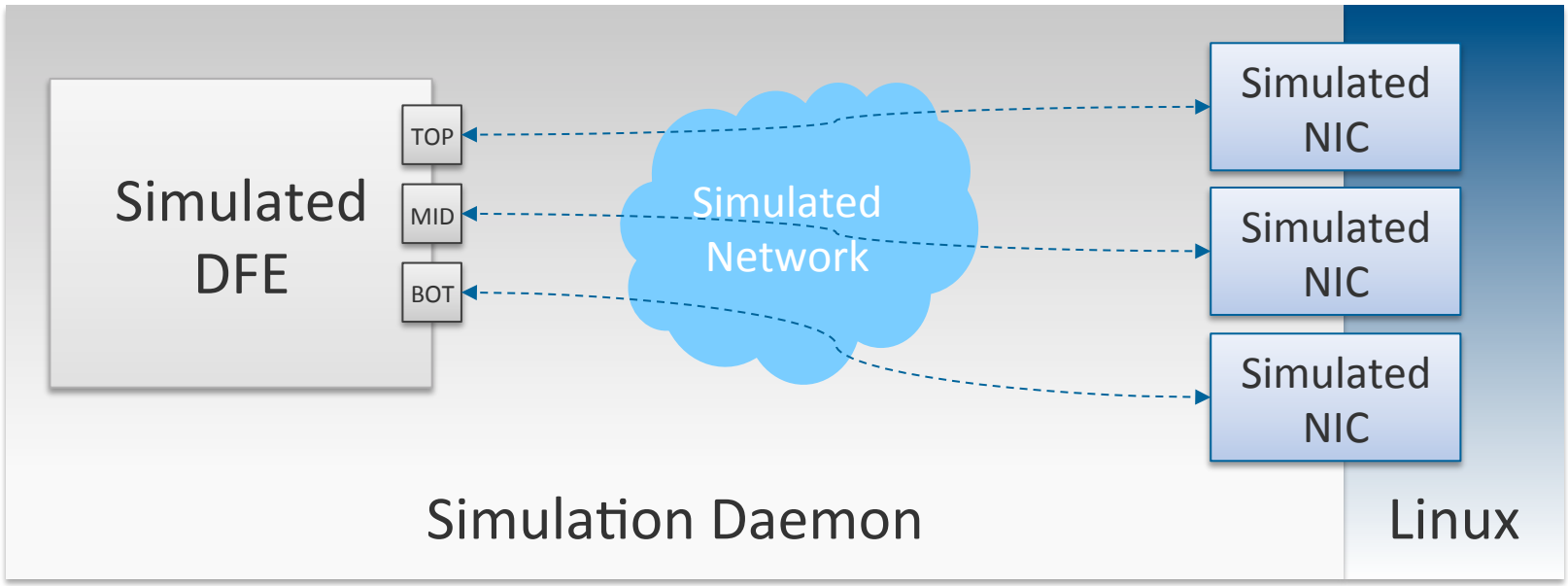
Network DFE Simulated System

- MaxelerOS comes with a Simulation environment
- It aims to be cycle-accurate when compared to the hardware environment
  - In reality, timing of dynamic events are completely different, but the in-kernel simulation is very accurate
- SLiC is always the same – so it's trivial for an application to switch between hardware and simulation
- The hardware is simulated inside the MaxelerOS Sim Daemon



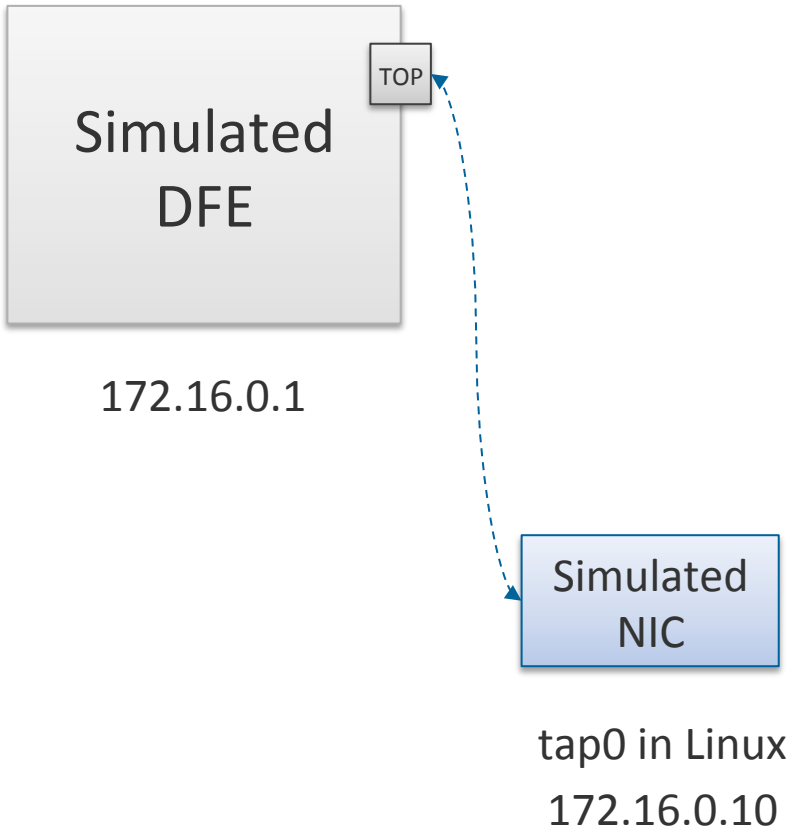
Simulated Networks

- Simulation uses TUN/TAP devices to create virtual NICs in Linux
- These NICs simulate a network device that has a direct connection to a physical port on the Simulated DFE
- Linux can send and receive packets through the simulated NIC and those would go to/from the simulated DFE's port



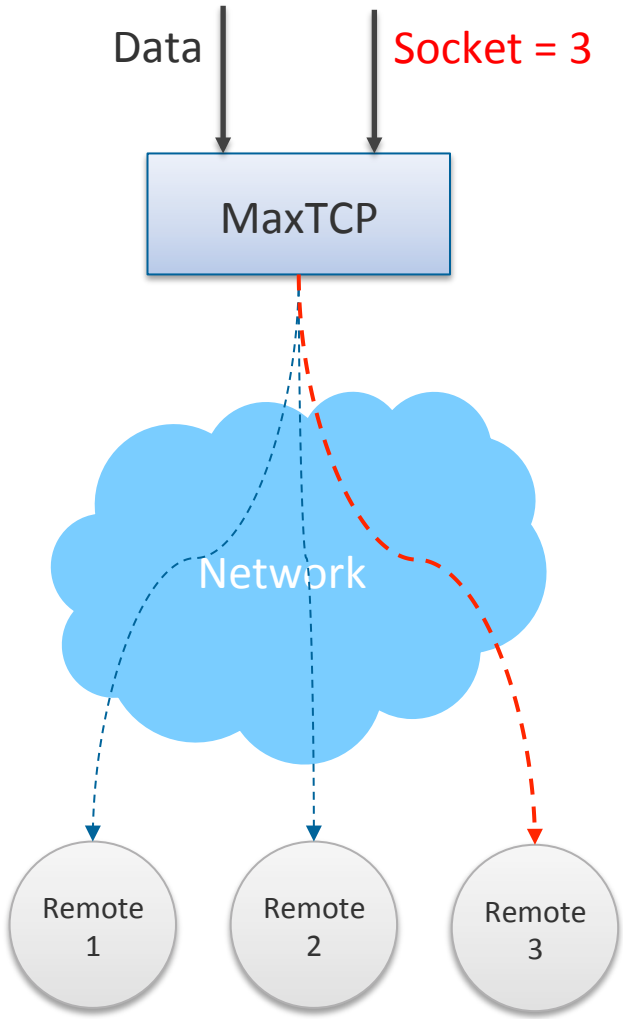
Simulation in Practice

- The simulated DFE is completely invisible from Linux's point of view
- The best way to think about it, is that the DFE is a different computer entirely – that happens to be connected to the same network that the Simulated NIC is connected to.
- This means that a Linux program that uses standard sockets, can send data back and forth to the Simulated DFE using standard network protocols – TCP/UDP/ ICMP etc
- You need to make sure:
  - To assign the Simulated NIC an IP address
  - To assign the Simulated DFE an IP address which is in the same network as the Simulated NIC



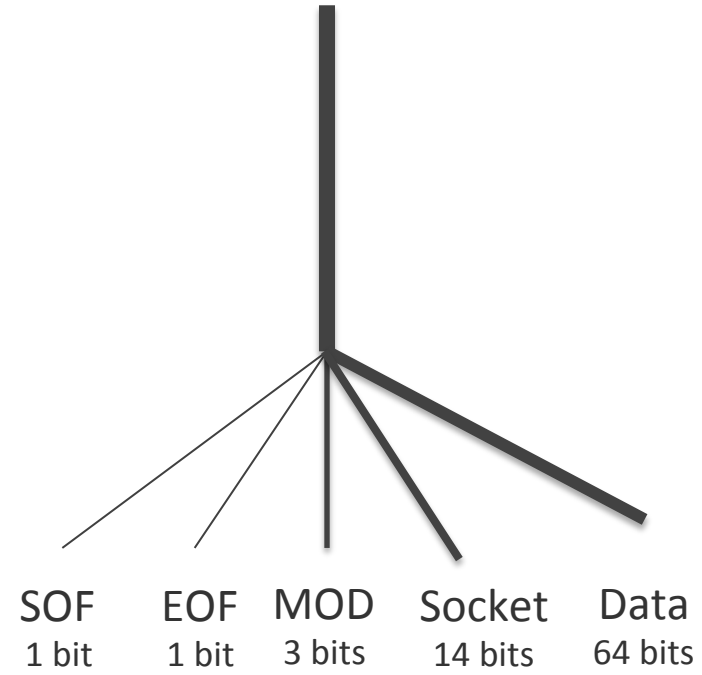
Metadata

- Metadata – Data about data
- Links in the Manager are designed to have data stream from one manager entity to the other, metadata is essential for contextualizing the data
- Most common examples:
  - SOF, EOF, MOD
    - Indicates how to interpret the data as a frame
  - UDP/TCP Socket
    - Tells which remote connection the data belongs to



Framed Link Fields with Metadata

- Ultimately, a link is just a collection of wires: data is indistinguishable from metadata from the Hardware's point of view
- When the link connects to the destination block, the link fields, it is viewed as one wide bus – in our example 83 wires.
- The individual fields are sliced out of this bus
  - First 64 wires are the data
  - Next 14 wires are the socket number etc.



The link width is 83 bits – includes both data and metadata. That's 83 wires going in to the manager block.

Summary

- Network streams can be handled by DFE kernels
- Networking DFEs have specific requirements
- There are challenges with flow control
- Flushing kernels and non-blocking inputs can help
- Networking DFE kernels care about latency