Project – Evaluation Plan

- For the next lab, you should prepare an evaluation plan for your project.
- The following evaluation tests are expected:
  - Functional testing (using the NetFPGA test infrastructure)
  - Performance testing, using synthetic traffic (using OSNT or equivalent to measure latency at low-rate and maximum throughput)
  - Performance benchmarking, using known benchmarks
  - Performance comparison to other solutions (software, hardware)
Low Latency Switches
How to lower the latency of a switch?

- Obvious option 1: Increase clock frequency
  - E.g. change core clock frequency from 100MHz to 200MHz
  - Half the time through the pipeline
How to lower the latency of a switch?

- Obvious option 1: Increase clock frequency
- Limitations:
  - Frequency is often a property of manufacturing process
  - Some modules (e.g. PCS) must work at a specific frequency (multiplications)
How to lower the latency of a switch?

- Obvious option 2: Reduce the number of pipeline stages
  - Can you do the same in 150 pipeline stages instead of 200?
  - Limitation: hard to achieve.
How to lower the latency of a switch?

- Can we achieve ~0 latency switch?
  
  - Is there a lower bound on switch latency?
Cut Through Switching
Cut Through Switch

- Cut through switch ≠ Low latency switch
  - A cut through switch can implement a very long pipeline...
- But:
  - For the smallest packet, the latency is ~same as longest packet
  - As packet size grows, latency saving grows
What is a cut-through switch?


- “when a message arrives in an intermediate node and its selected outgoing channel is free (just after the reception of the header), then, in contrast to message switching, the message is sent out to the adjacent node towards its destination before it is received completely at the node; only if the message is blocked due to a busy output channel is a message buffered in an intermediate node.”
What is a cut-through switch?

- Past (far back):
  - Networks were slow
  - Memory was fast
  - Writing packets to the DRAM took “negligible” time

- With time:
  - Networks became faster
  - Memory access time is no longer “negligible”
What is a cut-through switch?

What is a cut-through switch?

- But what does a REAL silicon implementation looks like?
- Tip 1: search for patents on Google Scholar
- Tip 2: read *carefully* performance evaluation reports
  - We’ll discuss some examples in the next lecture
Latency considerations within modules
Network Interfaces

• Data arrives at (up to) \(~50\text{Gbps per link}\).  
• Let us ignore clock recovery, signal detection etc.  
• Feasible clock rate is \(~1\text{GHz}\)  
• But if data rate is \(\times 50\) times faster…  

• Observation: data bus width will be no less than incoming data rate and feasible clock rate
Network Interfaces

• Line coding often directs the bus widths:
  • E.g., 8b/10b coding led to bus widths of 16b (20b) or 64b (80b)

• A port is commonly an aggregation of multiple serial links
  • 10G XAUI = $4 \times 3.125\text{Gbps}$
  • 100G CAUI4 = $4 \times 25\text{Gbps}$
  • 400G PSM4 = $8 \times 50\text{Gbps}$

• Need to take care of aligning the data arriving from multiple links on the same port.
Network Interfaces

- Role: check the validity of the packet (e.g., FCS)
- What to do if an error is detected?
  - Forward an error using a “fast path”
  - Mark the last cycle of the packet
    - E.g., to cause drop in the next hop
- Other roles need to be maintained too
  - Frame delimiting and recognition, flow control, enforcing IFG, …
Packet Processing

• A likely flow:

- Header starts
- Header parsed
- Match (table look up)
- Action (set output port)
- Header sent

• Possible implementations:

- The entire packet goes through the header processing unit
- Just the header goes through the header processing unit
- “Better” depends on your performance profile (what are the bottlenecks? Resource limitations?)
Packet Processing

- A likely flow:
  - Header starts
  - Header parsed
  - Match (table look up)
  - Action (set output port)
  - Header sent

- Challenges:
  - A field may arrive over multiple clock cycles (e.g. 32b field, 16b on clock 2 and 16b on clock 3)
  - Memory access taking more than 1 clock cycle
    - E.g. request on clock 1, reply on clock 3
    - Some memories allow multiple concurrent accesses, some don’t
    - The bigger the memory, the more time it takes
Packet Processing

• A likely flow:

Header starts → Header parsed → Match (table look up) → Action (set output port) → Header sent

• Solutions:

• Pipelining!
  Don’t stall, add NOP stages in your pipe.

• Reorder operations (where possible)

  • E.g. Lookup 1 → Action 1 → Lookup 2 → Action 2 turns:
    Lookup 1 → Lookup 2 → Action 1 → Action 2

• Don’t create hazards!
Arbitration

- Simple example:
  - Packets arriving from 4 ports
  - (approximately) same arrival time
  - Arbiter uses Round Robin
- Problem: arbitration on packet boundaries?
  - No: interleaved packets within the pipeline
  - Need to track which cycle belongs to which packet
  - May require multiple concurrent header lookups
  - Order is not guaranteed (e.g. P1-P2-P3-P1-P2-P2-…), due to NIF timing
Arbitration

- Simple example:
  - Packets arriving from 4 ports
  - (approximately) same arrival time
  - Arbiter uses Round Robin
- Problem: arbitration on packet boundaries?
  - Yes: packets need to wait for previous packets to be handled before being admitted.
    Worst case waiting with \(<N>\) inputs is \(<N-1>\times\text{Packet time}\)
Arbitration

• Solutions to the previous problem:
  • Scheduled (or slotted) traffic
  • Multiple pipelines
  • ...


How do you build a bufferless network?

- **Statistical Multiplexing Switch**
  - Switch Scheduler
  - STOP
  - Unpredictable output
  - Time = 0.8μs to >1900μs

- **Network Scheduler**
  - Predictable output
  - Time = Fixed (0.8μs to <3.5μs)

- **Host's queues** → **On the wire**
- **Switch queues**

- **Passive Cross Point (PCP)**
  - No queues
  - Predictable output
Arbitration

- This example solves the arbitration problem entering the device

- Resource inefficient:
  - Pipeline overdesign
  - Inefficient use of memories
  - Concurrency issues

- One solution:
  - Shared memories / tables
  - Highly complex
• The previous arbitration solution “pushed” the problem to the switching unit

• But now the problem is only when multiple packets compete over the same output – that’s fine!

• Assuming your switch can handle multiple packets per cycle
  • E.g. crossbar
Switching
Switching

• ... This is also queueing

• Challenge: SCALE
  • So you can do it with 4 ports
  • Can you do it with 32? 128? 256?

• Not just resource / area
  • Computation time – being able to examine and choose between all available inputs

• Eventually:
  Packets must be sent out on packet boundaries
  Do not interleave packets!
Host architecture

Legacy vs. Recent (courtesy of Intel)
Interconnecting components

• Need interconnections between
  – CPU, memory, storage, network, I/O controllers

• Shared Bus: shared communication channel
  – A set of parallel wires for data and synchronization of data transfer
  – Can become a bottleneck

• Performance limited by physical factors
  – Wire length, number of connections

• More recent alternative: high-speed serial connections with switches
  – Like networks
I/O System Characteristics

• **Performance measures**
  – Latency (response time)
  – Throughput (bandwidth)
  – Desktops & embedded systems
    • Mainly interested in response time & diversity of devices
  – Servers
    • Mainly interested in throughput & expandability of devices

• **Reliability**
  – Particularly for storage devices (fault avoidance, fault tolerance, fault forecasting)
I/O Management and strategies

- **I/O is mediated by the OS**
  - Multiple programs share I/O resources
    - Need protection and scheduling
  - I/O causes asynchronous interrupts
    - Same mechanism as exceptions
  - I/O programming is fiddly
    - OS provides abstractions to programs

Strategies characterize the *amount of work* done by the CPU in the I/O operation:

- Polling
- Interrupt Driven
- Direct Memory Access
The I/O Access Problem

• Question: how to transfer data from I/O devices to memory (RAM)?

• Trivial solution:
  • Processor individually reads or writes every word
  • Transferred to/from I/O through an internal register to memory

• Problems:
  • Extremely inefficient – can occupy a processor for 1000’s of cycles
  • Pollute cache
DMA

• DMA – Direct Memory Access
• A modern solution to the I/O access problem
• The peripheral I/O can issue read/write commands directly to the memory
  • Through the main memory controller
  • The processor does not need to execute any operation
• Write: The processor is notified when a transaction is completed (interrupt)
• Read: The processor issues a signal to the I/O when the data is ready in memory
1. Message arrives on I/O interface. Message is decoded to Mem read/write. Address is converted to internal address.

2. Mem Read/Write command goes through the switch to the internal bus and memory controller.

3. Memory controller executes the command to the DRAM. Returns data if required in the same manner.
DMA

- DMA accesses are usually handled in **buffers**
  - Single word/block is typically inefficient
- The processors assigns the peripheral unit the buffers in advance
- The buffers are typically handled by **buffer descriptors**
  - Pointer to the buffer in the memory
  - May point to the next buffer as well
  - Indicates buffer status: owner, valid etc.
  - May include additional buffer properties as well
Transfers blocks of data between external interfaces and local address space

1. A transfer is started by SW writing to DMA engine configuration registers

2. SW Polls DMA channel state to idle and sets trigger

3. DMA engine fetches a descriptor from memory

4. DMA engine reads block of data from source

5. DMA engine writes data to destination
Intel Data Direct I/O (DDIO)

- Data is written and read directly to/from the last level cache
PCIe introduction

- PCIe is a *serial point-to-point interconnect* between two devices
- Implements *packet based protocol (TLPs)* for information transfer
- *Scalable performance* based on # of signal Lanes implemented on the PCIe interconnect
- Supports *credit-based* point-to-point flow control (not end-to-end)

Provides:
- Processor independence & buffered isolation
- Bus mastering
- Plug and Play operation
PCIe transaction types

- Memory Read or Memory Write. Used to transfer data from or to a memory mapped location.
- I/O Read or I/O Write. Used to transfer data from or to an I/O location.
- Configuration Read or Configuration Write. Used to discover device capabilities, program features, and check status in the 4KB PCI Express configuration space.
- Messages. Handled like posted writes. Used for event signaling and general purpose messaging.
PCIe architecture
Interrupt Model

PCI Express supports three interrupt reporting mechanisms:

1. Message Signaled Interrupts (MSI)
   - interrupt the CPU by writing to a specific address in memory with a payload of 1 DW

2. Message Signaled Interrupts - X (MSI-X)
   - MSI-X is an extension to MSI, allows targeting individual interrupts to different processors

3. INTx Emulation
   - four physical interrupt signals INTA-INTD are messages upstream
     - ultimately be routed to the system interrupt controller
NetFPGA Reference Projects

Host system

PCI endpoint
Direct Memory Access
Input Arbiter
Output Port Lookup
Output Queues

AXI Interconnect

10GE
10GE
10GE
10GE
Processing Overheads

- Processing in the kernel takes a lot of time…

<table>
<thead>
<tr>
<th>Component</th>
<th>Time [us]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver RX</td>
<td>0.60</td>
</tr>
<tr>
<td>Ethernet &amp; IPv4 RX</td>
<td>0.19</td>
</tr>
<tr>
<td>TCP RX</td>
<td>0.53</td>
</tr>
<tr>
<td>Socket Enqueue</td>
<td>0.06</td>
</tr>
<tr>
<td>TCP TX</td>
<td>0.70</td>
</tr>
<tr>
<td>IPv4 &amp; Ethernet TX</td>
<td>0.06</td>
</tr>
<tr>
<td>Driver TX</td>
<td>0.43</td>
</tr>
</tbody>
</table>

Source: Yasukata et al. “StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs”, Usenix ATC 2016
Processing Overheads

- Processing in the kernel takes a lot of time…
- Order of microseconds (~2-4us on Xeon E5-v4)
- $\times 10$ the time through a switch

- Solution: don’t go through the kernel!
Kernel Bypass

• The Kernel is slow – lets bypass the Kernel!

• There are many ways to achieve kernel bypass

• Some examples:

  • Device drivers:
    • Customized kernel device driver. E.g. Netmap forks standard Intel drivers with extensions to map I/O memory into userspace.
    • Custom hardware and use bespoke device drivers for the specialized hardware.

  • Userspace library: anything from basic I/O to the entire TCP/IP stack
Kernel Bypass - Examples

- No Bypass
- Partly within Kernel
- Completely in User Space
DPDK

• DPDK is a popular set of libraries and drivers for fast packet processing.
• Originally designed for Intel processors
  • Now running also on ARM and Power CPUs
• Runs mostly in Linux User space.
• Main libraries: multicore framework, huge page memory, ring buffers, poll-mode drivers (networking, crypto etc)
• It is not a networking stack
Usage examples:

- Send and receive packets within minimum number of CPU cycles
  - E.g. less than 80 cycles
- Fast packet capture algorithms
- Running third-party stacks
- Some projects demonstrated 100’s of millions packets per seconds
  - But with limited functionality
  - E.g. as a software switch / router