Embedded Systems Design
with Platform FPGAs

Chapter -6 Managing Bandwidth

Dr. Bassam Jamil
Adopted and updated from
Ron Sass and Andrew G. Schmidt

Chapter 6 — Managing Bandwidth

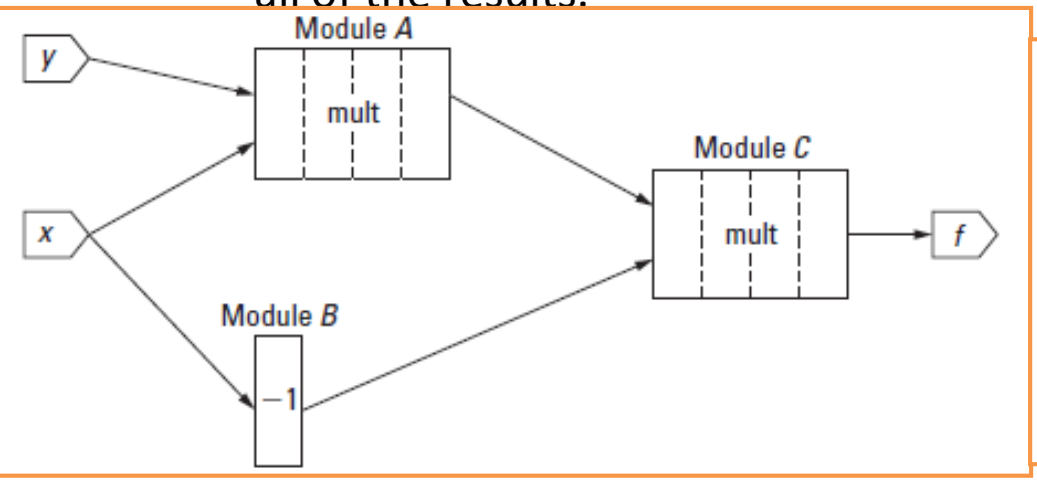
Managing Bandwidth

- The most fundamental issue in building a computing system is managing **the flow of data through the system**.
- Without considering the **rates** at which various function units consume and produce data, it is very likely that any potential performance gains (in function units) may be lost because the function units are idle, waiting to receive their inputs or transmit their results.
- Design Issues:
  - Performance issues include both rate of computation and power because an idle function unit is still using static power.
  - There is also a correctness issue for some real-time systems as well. It is often the case that data are arriving from an instrument at a fixed rate — failing to process data in time is frequently considered a fault.
- In this chapter we will discuss:
  - balancing bandwidth in parallel pipelines.
  - methods of managing bandwidth on- and off-chip.
  - scaling bandwidth with number of function units.

Outline of This Chapter

6.1 Balancing Bandwidth

- Suppose:
  - f = xy (x−1)
  - the computation is applied repeatedly to a large number of sequential inputs.
  - Lets discuss the options of implement the function
- **(1) Simple Network**
  - Multiplier is four-stage pipeline. Adder completed in one cycle.
  - A stall occurs in two cases:
    - (1)whenever a computation unit has some, but not all, of its inputs **or**
    - (2) when it does not have the ability to store its output,
  - Module B is stalled because it has to hold its result until module C uses it.
  - If we have n number of x and y inputs, the network will take 4n cycles to compute all of the results.



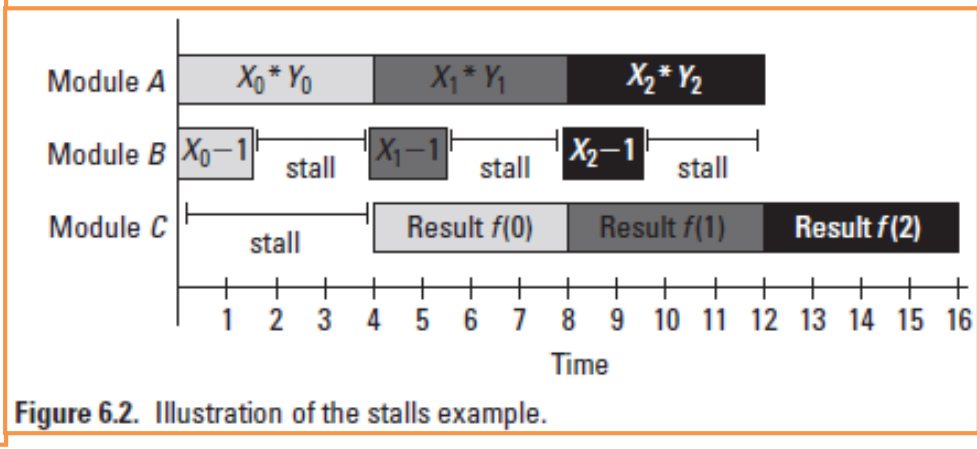
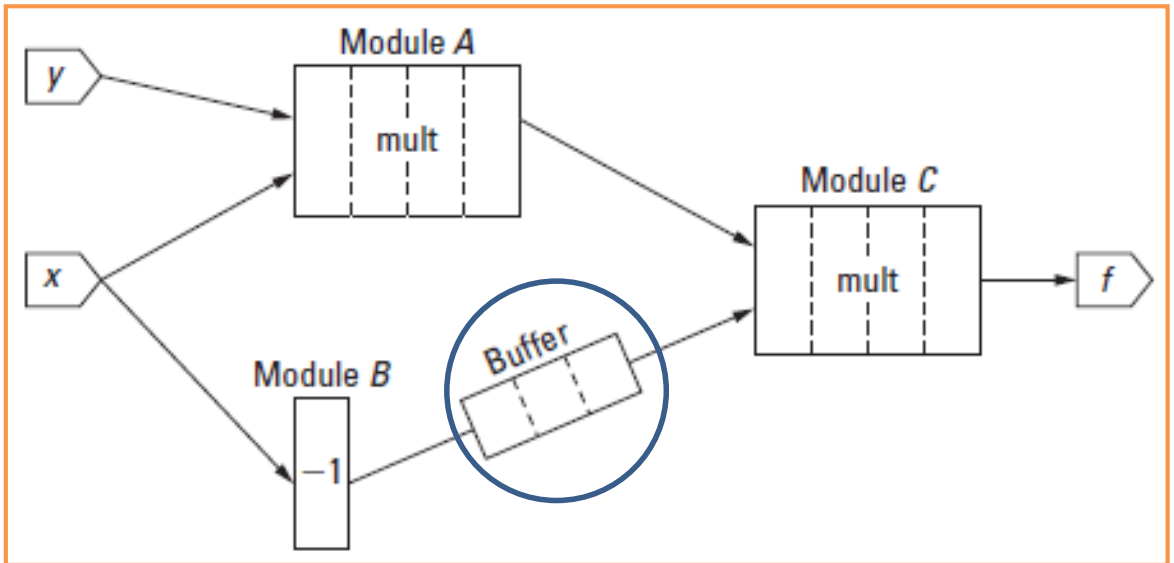


Figure 6.2. Illustration of the stalls example.

6.1 Balancing Bandwidth, continue

- **(2) High-throughput Network: Pipeline balancing**
  - Add a four-stage FIFO or buffer into the network.
  - We can make this computation produce a new result every cycle after t > 4.
  - Pipeline balancing: the introduction of buffers so that a network of computations does not stall.



6.1.1. Kahn Process Network (KPN)

- In a **KPN**, one or more processes are **communicating through FIFOs**, with blocking reads and non-blocking writes.
- Below figure shows KPN
  - each **node** (circle) represents a process
  - each **edge** (A, B, F) between the nodes is a unidirectional communication channel.
- KPN is also called **data flow graph** with the following **firing rule**:
  - the operation only begins when all of its inputs are available and there is a place for it to write its output.

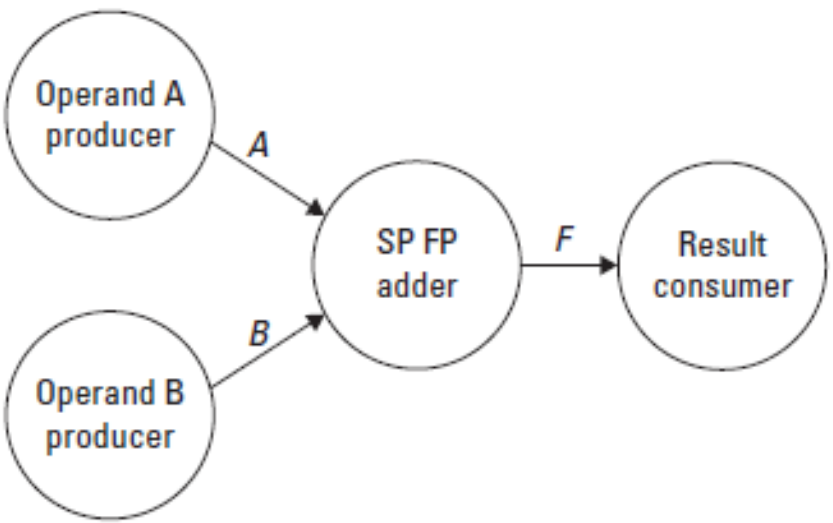


Figure 6.4. Kahn process network diagram of a simple single precision floating-point system.

KPN Features

- Kahn process network supports the **unidirectional flow of data** from a source to a final destination.
- KPN **does not offer** any mechanism to provide **feedback** between compute processes.
- The requirement of “unbound” FIFOs between processes makes for challenging designs in systems
  - FIFOs are a fixed resource that cannot be reallocated,
  - when the Block RAM has been allocated as a FIFO and the design is synthesized and mapped to the device, that memory is fixed to that process.
- Platform FPGA designs can quickly incorporate the Kahn process network with fixed on-chip memory resources by including feedback between the compute cores.

Synchronous and Asynchronous Designs

- Synchronous Design
  - Operations share the same clock and data are injected into the network every clock cycle.
  - Because all data will arrive at the inputs at the right time ,
    - We can avoid checking the data-flow firing rules and we get maximum throughput.
    - the FIFOs used in the example can be replaced with a smaller chain of buffers.
- Asynchronous design strategy
  - Uses finite state machines to read inputs, drive the computation, and write the outputs.
  - No global clock, so the design can be partitioned across different clock domains.
  - If the buffers are added as in the second implementation example, then it too has the benefit of getting maximum throughput.

Synchronous and Asynchronous Designs

- In both synchronous and asynchronous cases,
  - we have assumed that each **operation takes a fixed number of cycles.**
  - However, the model can be extended to also include **variable length latency operations** by using the maximum latency.
- The key to both of these design strategies is that FIFOs have the correct minimum depth.
  - However, we do not want to waste resources—especially memory resources
  - So the key to this problem is to find the minimum number of buffers needed to maximize throughput

Outline of This Chapter

6.2. Platform FPGA Bandwidth Techniques

- In this section, we will:
  - –look at **integrating designs with on-chip and off-chip memory**.
    - A variety of memory types and interfaces exist and understanding when each is applicable is important to Platform FPGA designs
  - –consider **data streaming** into the FPGA from some instrument.
    - The “instrument” may be a sensor , some digitally converted signal , or even low-speed devices such as keyboards and mice.
    - In some cases the instrument may need to use off-chip memory as a buffer or as intermediate storage

6.2.1. On-Chip and Off-Chip Memory

- The designer has to pay attention to the memory subsystem.
- The FPGA fabric does not include embedded caches found with modern processors.
  - Note: Some hard processors may include cache, such as the Xilinx PowerPC 440, although this cache is not accessible by the rest of the FPGA.
  - FPGA designers must either construct custom caches or rely upon a modified memory hierarchy.
  - From a custom compute core's perspective, we modify the drawing of the memory hierarchy for Platform FPGAs by removing the cache and replacing it with user-controlled local and remote on-chip memory (storage),

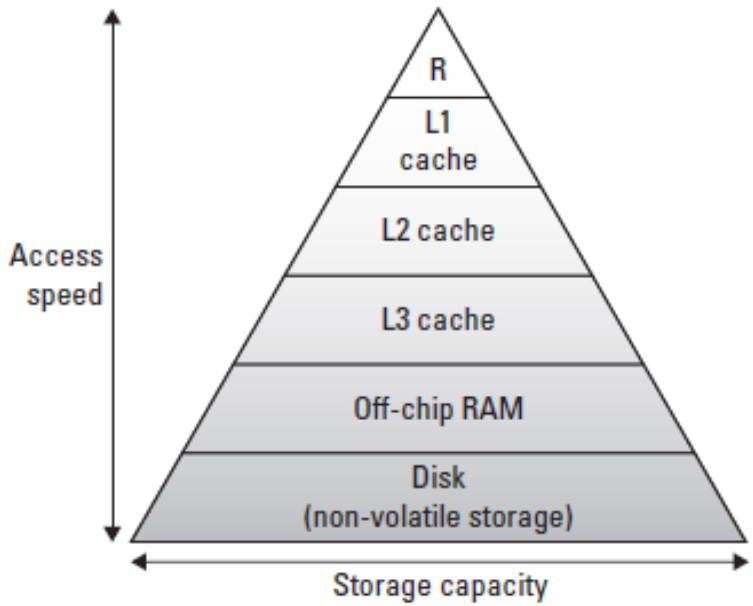


Figure 6.6. Traditional memory hierarchy comparing storage capacity to access times.

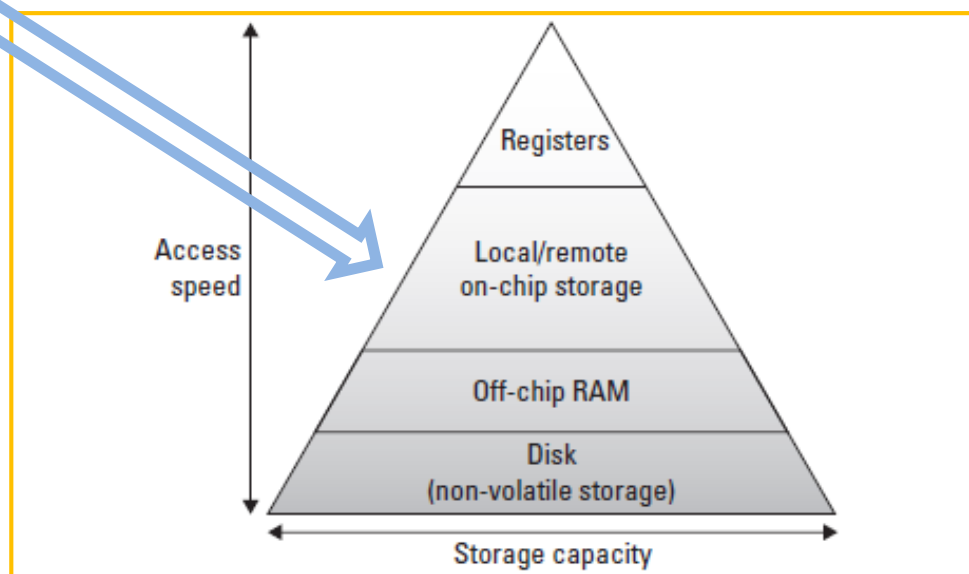


Figure 6.7. FPGA compute core's memory hierarchy with local/remote storage in place of cache.

Memory Locations with Respect to FPGA Compute Core

- Local on-chip vs. remove on-chip memory
  - On-chip memory is considered local storage when the memory resides within a custom compute core, maybe as Block RAM.
  - Remote storage is still on-chip memory, except that it does not reside within the compute core.
    - An example of this is on-chip memory that is connected to a bus, accessible to any core through a bus transaction.
    - The locality is relative to the accessing core. If compute core A needed to read data from compute core B's local memory, we would say that A's remote memory is B's local memory.

The figure shows various memory locations with respect to an FPGA's compute core with access times increasing from :
• registers,
• to local on-chip storage,
• to remote on-chip storage,
• finally to off-chip memory

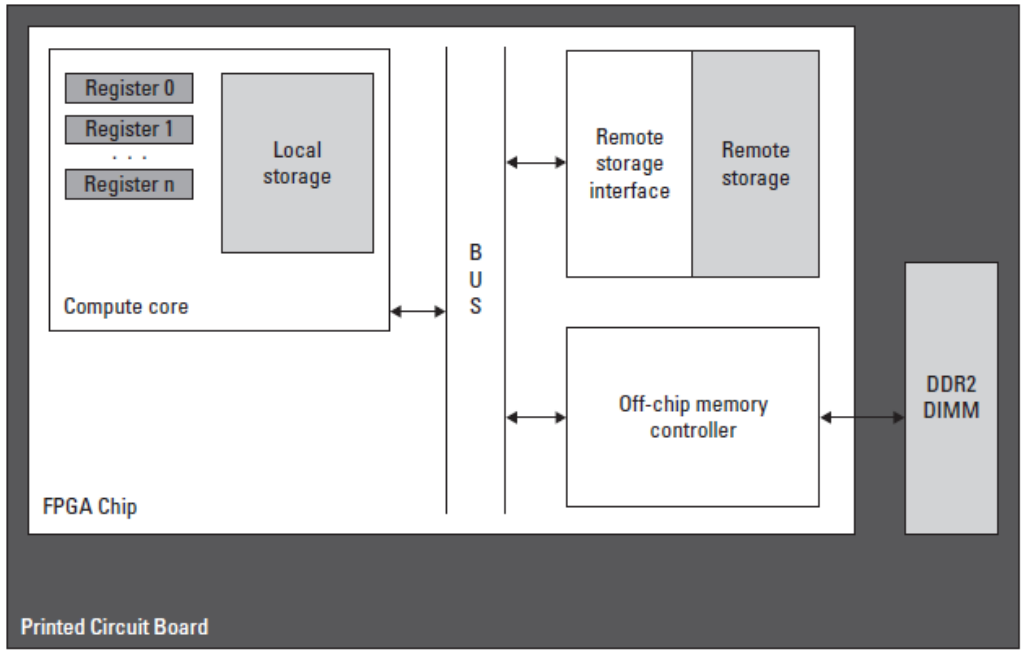


Figure 6.8. Various memory locations with respect to an FPGA's compute core with access times increasing from registers, to local storage, to remote storage, and finally to off-chip memory.

Memory Hierarchy: Access Types

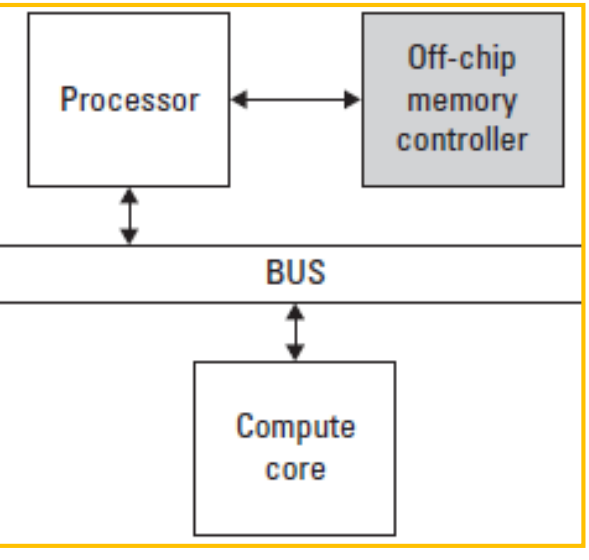
Memory location	**Access Type**	**Access speed**
Registers	Data are referenced by **register by name** rather than by address	**1 cycle**
Local On-chip (e.g. Block RAM)	Access is based **handshaking signals**, which may include: address, data in, data out, read enable, and write enable.	**1-2 cycles**
Remote on -chip	• Access the memory using a **bus request**. • Requires **bus interface** to translate bus requests into Block RAM requests.	**Few tens of cycles**
Off chip memory	• Same as on-chip remote storage, • But additionally requires off-chip memory controller.	**Tents to hundreds of cycles**

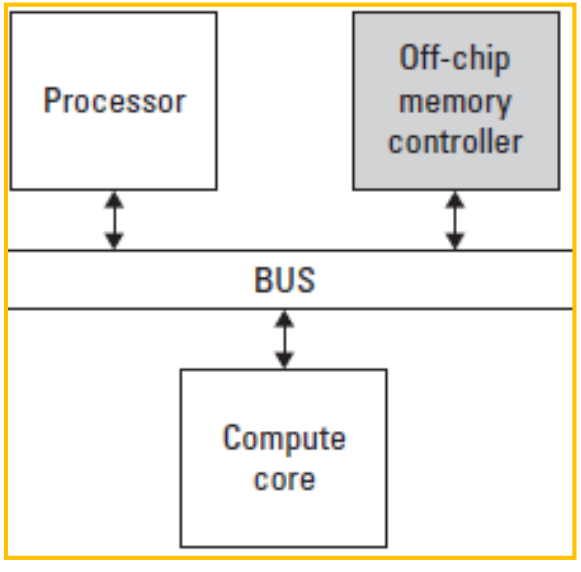
cache vs. on-chip local/remote storage

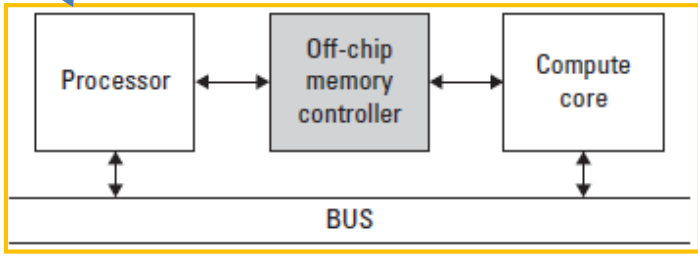
- Both share the same concept **of moving frequently used data closer** to the computation unit.
- The difference is the **controlling mechanism**.
  - For caches, sophisticated controllers support various data replacement policies, such as direct-mapped, set associative, and least recently used.
  - In Platform FPGA designs we are left to implement our own controller.
    - This may seem like a lot of additional work , but keep in mind many custom compute cores most likely do not follow conventional memory access patterns or protocols, so creating custom controllers may actually be necessary to achieve higher computation rates

Off-Chip Memory Controller

- Off-chip memory controller
  - is responsible for turning **memory requests into signals sent off-chip**
  - reside on-chip as a soft core that interfaces between off-chip memory and the compute cores.
- Interfacing between off-chip memory controller and on-chip compute cores:
  - The memory controller is connected directly to the processor.
  - The memory controller is connected to a shared bus to support requests by any core on the bus
  - The processor and the compute core interfacing to the memory controller directly





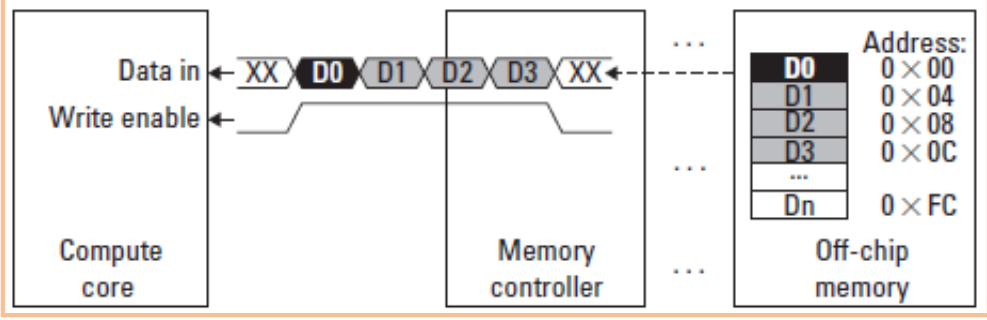


Memory Bandwidth

- Two important parameters for memory transfer
  - Bandwidth : # bits/sec
  - Latency: initial latency time for the first word
- **Bandwidth** = Operating_Frequency × Data_Width
- Example
  - a bus operating at 100MHz over 64-bit data words
  - bandwidth = $(100\times10^6/\text{sec})\times64$ bits = 6400 Mbits/sec
- **Latency**:

  1. initiates a transfer,
  2. the request may need to wait to be granted access to master the bus,
  3. wait for the memory controller to complete its current transaction,
  4. wait for data to be returned.

Burst Transfer



- A burst transfer
  - a single request to the memory controller to read or write multiple sequential data
- Burst transfer latency:
  - for the first datum is still the same,
  - but each datum after the first arrives in a pipelined fashion, one datum per clock cycle
- Example:
  - A burst transfer of four 32-bit sequential data words would take the initial latency time for the first word, denoted as $t_{latency}$, plus three clock cycles for the remaining three words:
  - $t_{total} = t_{latency} + 3t$
- 2 limitations to burst transfers.
  - Data must be in a contiguous memory region.
  - Length may be limited by an upper bound, requiring multiple burst transactions for large request lengths

6.2.2. Streaming Instrument Data

- Streaming Instrument Data:
  - Data generated (by an instrument) and arrive at some **fixed rate and must be processed**, otherwise they may be lost.
  - Examples: **video cameras and sensors**
- System Designer must:
  - How to plan for the data, and how to use it.
  - Consider using on-chip and off-chip memory intelligently for short-term storage when the need arises to buffer data.
- For instruments with a fixed sample rate:
  - The designer can calculate the exact amount of time allotted to processing data and work to design a compute core accordingly.
  - As the sample rate increases, the amount of time to process data decreases unless the designer prepares the system to be pipelined.
- **Pipeline implementations**: The requirement on pipelining is that the sampling rate (arrival rate of new data) be less than or equal to the slowest stage in the pipeline. So, pipeline:
  - Achieve high throughput,
  - Alleviate the tight timing constraints that may be placed on a designer with a high sampling rate.

Streaming Instrument Data: Memory Issues

- If compute unit cannot process data fast enough and **on-chip memory** does not provide sufficient storage space, then
  - it is necessary to use **off-chip memory** as a larger intermediate buffer.
- A compute core requiring memory as a **buffer** may need to:
  - **store input** data arriving faster than can be computed
  - **retrieve data** when it can be computed
  - **store** the computed **results**

6.3. Scalable Designs

- When going from one FPGA generation to the next, there will be additional resources:
  - more capacity,
  - more capability,
  - more flexibility.
- Scalability: “How do we modify our design to take advantage of these additional resources?”
- Simply increasing the number of compute cores may have adverse effects on the system unless careful consideration is made.

6.3.1. Scalability Constraints

- **Constraint (1):** Number of multiple compute core that can be included in the design

  Number of Cores = Available Resources/(Resources/Core)

  - Formula sets the upper bound on the actual number of compute cores obtainable on the device.
  - Example: A designer who builds a compute core that utilizes 20% of the FPGA resources. If the design allows for multiple instances of that compute core to be included, we could conceivably instantiate five cores and use 100% of the resources.

- **Constraint (2): bandwidth**
  - "As we increase number of cores, can we sustain the necessary bandwidth
    - to/from a central processor,
    - between each compute core,
    - or to/from memory

To see any significant performance gains when scaling the design, we must be considerate of their bandwidth's impact on the system.

Bandwidth: Processor's Perspective

- We consider the "bandwidth" from the processor's perspective as: the time the processor spends to **control the core**.
  - Recall : processor that can only communicate with one core at a time,
- In scalable designs we want to identify the amount of attention a compute core needs and scale the design (by adding more cores)to not exceed the processor's capacity:

Processor to Core Bandwidth =

(Processor-to-Core Max Bandwidth)/ (Number of Cores)

Bandwidth: Compute Core's Perspective

- Compute Cores can be connected by shared bus or cross bar

(1) Connection by Shared Bus:

- As we add more compute cores to the bus, we run the risk of saturating that shared resource.
- The bus has a fixed upper bound on its bandwidth:

  Bus Bandwidth= Bus Frequency × Bus Data Width

- core's bus bandwidth is calculated as:

  Core Bus Bandwidth = Core Frequency × Bus Data Width × (% of Core Bus Utilization)

- To calculate the required bus bandwidth we can sum up each core's bus bandwidth. The bus saturates when the required bus bandwidth is greater than the available bus bandwidth.

Bandwidth: Compute Core's Perspective

(2) Connection by Crossbar:

- A crossbar switch allows more than one core to communicate with another core in parallel.
- Each output port is connected to a four-input multiplexer, totaling in 4 4-port multiplexers.
- As we increase the size, we increase the number of cores that can connect and communicate in parallel.
- So scaling the design increases number and size of multiplexers

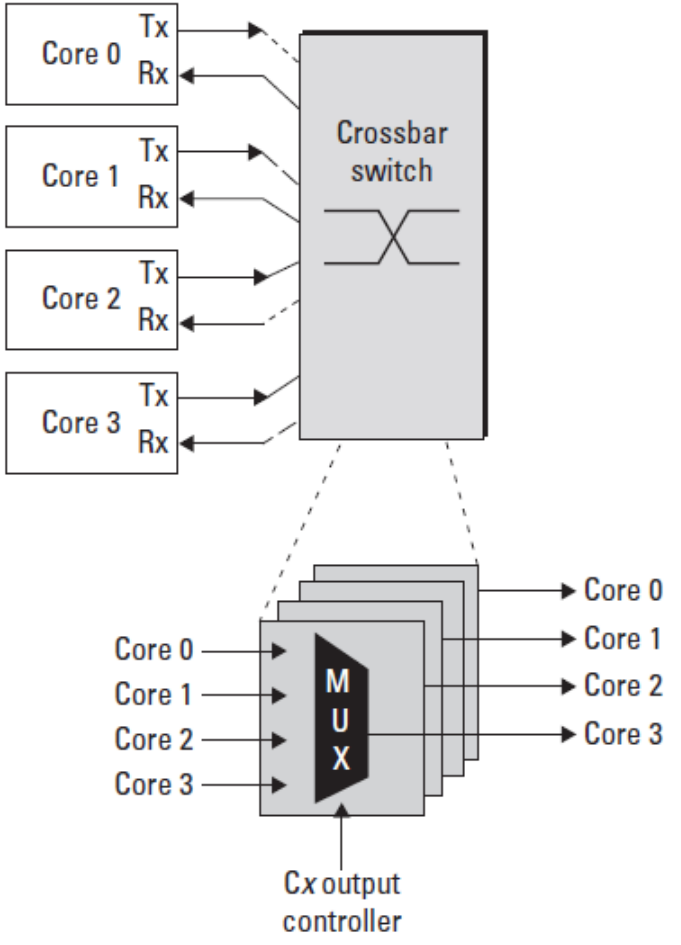


Figure 6.15. A four-port crossbar switch and its internal representation based on multiplexers.

Bandwidth: Memory Perspective

- Memory bandwidth is defined as

  Memory Bandwidth= Frequency × DataWidth

- Off-chip: the bottleneck is at the memory controller where data are then distributed across the FPGA

6.3.2. Scalability Solutions

Limitation	Solution
Overcoming Processor Limitations	• move more of the application's computation into hardware • allow hardware cores to independently access memory through DMA • use interrupts instead of polling to communicate with the processor
Overcoming Bus Bandwidth Limitations	• increasing operating frequency of the bus • increasing data width of the bus • segmenting the bus into smaller, more localized buses • using burst transfers to communicate more efficiently
Overcoming Memory Bandwidth Limitation	• provide support for direct memory access • separate interfaces for on-chip and off-chip communication • use double buffering of requests whenever possible

Chapter 6 Terms

stall occurs in two cases, whenever a computation unit has some, but not all of its inputs, or when it does not have the ability to store its output

pipeline balancing refers to the introduction of buffers so that a network of computations does not stall

data-flow graph the network represented as a directed graph

data-flow firing rule the operation only begins when all of its inputs are available and there is a place for it to write its output

Chapter 6 Terms

programmable I/O the processor performs all requests on behalf of the compute core; specifically related to transfers between memory and the compute core

direct memory access (DMA) alleviating the processor from the task of transferring memory to or from a compute core through the use of a intermediate DMA controller or by allowing the compute core to directly access memory

double buffering refers to requests that are made while the previous request is still in transit

burst transfer consists of issuing a single request to read or write multiple sequential data elements; typically used by compute cores during DMA transactions to access on-chip/off-chip memory