Embedded multiprocessing in data network applications

Course SYSC 5906 Directed Studies, Summer 2010. Co-supervised by professors Miodrag Bolic, School of Information Technology and Engineering (SITE), University of Ottawa and Ioannis Lambadaris Department of Systems and Computer Engineering (SCE), Carleton University

Submitted by: Josip Popovic, josip_popovich@yahoo.ca, MASc Candidate

Table of Contents

Abstract

In this paper we look at reasons why ASIC data network solutions are being gradually replaced with more suitable technologies. An overview of NP architecture is given followed by a chapter on Tensilica processors and how they can be used as NP building blocks. Tensilica processors with their TIE queues are well equipped for the producer/consumer based designs.

Introduction

Conventional wisdom of using exclusively Application Specific Integrated Circuits (ASIC) as the high end system building blocks is crumbling [1]. ASIC development costs and timelines are progressively becoming unacceptable so they are being replaced or augmented by a number of programmable technologies. Attempts to raise hierarchical level of design moving from tedious RTL coding towards behavioral and system level programming languages proved to be wrong direction. Synopsys Behavioral Compiler and SystemC Compiler are both not accepted by the semiconductor industry [21].

Processors at one point in time were optimized for a particular language [2]. However the real processor improvements are achieved by implementing algorithms in the processor hardware. Therefore processor customization and extensions become features of the paramount importance, they allow adding critical algorithm speedups. A completely new family of soft processor cores and tools for their customization and extension were launched. These new processors are being used as building blocks in SoC designs replacing custom RTL components.

New multimedia applications are addressed by stream multicore processors [2]. Streaming processors do not fit the von Neumann architecture. Data is streamed between individual threads using producer/consumer architecture (FIFO) avoiding or minimizing a need for shared address space, shared buses and memories. Pipelined stream processor threads/cores execute their respective tasks forming instruction, data and thread level of parallelism.

Networking Processors (NP) are programmable devices with packet processing hardware support. Due to the fast path packet processing speed requirements NPs employ a number of processing units. In addition to the fast path packet processing NPs need to process control and management packets. These packets do not have high speed requirements but level of processing is higher than for the fast path packets. Therefore NPs take advantage of control processor and packet processors – asymmetric multiprocessing. Control processors have less processing power but they require a large code space. Packet processor are opposite, lots of processing power, less code space. Packet processors tend to be multithreaded in order to hide IO latencies. If packet processor cores were not multithreaded, they would be most of time idle waiting for an IO transaction to complete – multithreading hides IO (for example memory) latencies. One of important questions to answer while designing HW or writing SW for NPs is how to use their multicore/multithreaded architecture: process a packet to completion on one the same core/thread or to do pipeline processing while forwarding packet data.

The latest networking processors are designed as asymmetric multicore and multithreaded architectures, with a choice of pipeline producer/consumer packet processing or processing to the completion. Due to their massive parallel structure shared memories where applicable are avoided and are replaced by the producer/consumer interconnects.

Ever Changing Design Landscape

Semiconductor Industry Push and Application Pull

Quest for semiconductor development environment changes are driven by the semiconductor industry itself (push to new design methodologies) and by new emerging applications (pull to new methodologies) [1].

The semiconductor push to the new methodologies can be categorized as:

Increasing Design Costs

- Deep-Submicron (DSM) Effects: geometry of wires used to connect gates in DSM (height versus width) imposed longer propagation delays as well as parasitics. On-chip signal integrity becomes part of the regular design flow. Wire effects amplified by a number of other DSM effects force a multiple of back end iteration before successfully meeting tape-out requirements (static timing analysis being one of them). New, cutting edge design tools are required driving development costs and project schedules to new levels.
- Increased Complexity: shrinking process technologies allow integration of millions of transistors. This puts more pressure on the exiting back-end tools. Designs are divided into smaller hierarchical blocks allowing tools to cope with complexity. Each of these sub-chips need separate backend and verification environments.
- Mixed Level Designs: Increased system level integration requires analog, RF and digital circuitry on the same die. A variety of design tools as well as expertise is required.
- Time to Market: time to market is getting shorter. This is driven by the customer requirements, new and emerging applications as well as the latest advances and research.

The following quote comes from the International Technology Roadmap for Semiconductors (ITRS) 2001 summarized it as: *"The main message in 2001 is this: Cost of design is the greatest threat to continuation of the semiconductor roadmap.*

Increasing Manufacturing Costs:

- Mask Costs
- Packaging Costs: days of two layer packages are gone. A number of high speed interfaces, their width or data rate (cross talk, power noise, isolated power planes, number of power domains, clock distribution etc) dictate package development. Semiconductor package design tools for mechanical package design, electrical, signal integrity, heat distribution etc are all required
- Testing: wafer level testing, manufacturing ATE, production test and development testing are all growing in complexity. The following quote from the ITRS 2001 states: "Test costs have grown exponentially compared to the manufacturing costs."

The above design methodology constraints suggest that minimizing non-recurring engineering costs (NRE) is imperative. One way of minimizing NRE effects is to spread these costs over:

- A number of products being supported by the same chip/design (different applications)
- Standard/specification updates that would normally require a new chip spin-off
- Delivering features in steps (getting to market faster)
- Bug fixes without a need for respin
- Starting design cycle before actual standards/specs are completed

The application pull to the new methodologies come from high volume, low power, wireless and portable devices. Field programmability and high performance computing are the main design requirements. ASICs inherently do not offer programmability while general purpose processors do not offer enough performance, energy efficiency or low costs.

The following 5 technologies are the main technologies being used in today's contemporary electronic devices: ASIC, ASIP, FPGA, ASSP, DSP and GPP.

ASIC (Application Specific Integrated Circuit) is the highest performance technology in building Systems On the Chip (SOC). ASIC is the most complex and expensive to develop. At the same time it allows very low programmability (behavior of state machines changed via changing register settings).

FPGA (Field Programmable Gate Array) offer field programmability (configuration change), low development costs (automatized-IDE tool used for back end) very good clock speeds, somehow limited silicon utilization and high part costs.

ASIP (Application Specific Instruction Processors) are based on the instruction set architecture where the instruction set (and underlying hardware) is adapted to a specific application. Some examples are DSPs, microcontrollers, network or other domain-specific processors. Here we define an ASIP as an instruction set oriented processor with application-specific optimizations (optimized just for one application) including optional tightly-coupled hardware accelerators [3].

ASSP (Application Specific Standard Products) offer a standard set of features that may be applicable to a certain market segment, for example video encoding. ASSP design methodology is similar to ASIC while development costs are amortized over a number of applications.

DSP (Digital Signal Processors) are processors specialized for a variety of digital processing applications.

GPP (General Purpose Processor) are off-the-shelf multi-purpose processors. GPP offer the highest flexibility, lowest part and application development costs. However on the flip side they have the highest power consumption and the lowest performance.

The following figure graphically portraits relative relations of the 5 main technologies [3].

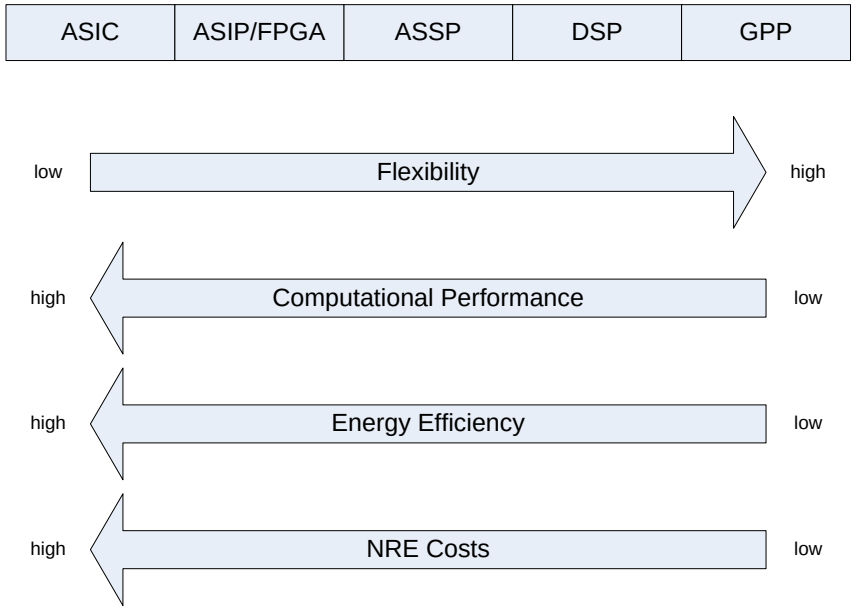


Figure 1 - Relative relations of the 5 main technologies (based on [3] and [4] with addition of NRE)

Figure 1 displays relative direction (high/low) of a number of methodology constraints (Flexibility, Computational Performance, Energy Efficiency and NRE) versus selected technology. These constraints are defined by the semiconductor industry push and by the application requirements pull.

Figure 1 and the current industry trends point to ASIP as an optimum design methodology that actually is increasingly replacing ASIC.

Design Discontinuity

In [1] authors show a chart that correlates design productivity and investment in EDA tools (Figure 1). The chart is characterized by a 4 S-curves. Each S-curve indicates periods of significant EDA investments with mediocre increase followed by a periods of significant increase in the design productivity. A good example is introduction of RTL (Register Transfer Level) design entry replacing much more labor intensive gate level design.
Design discontinuities take place when a contemporary design methodology allow a design teams made of a few designers outperform a design team of thirty engineers using an old methodology.

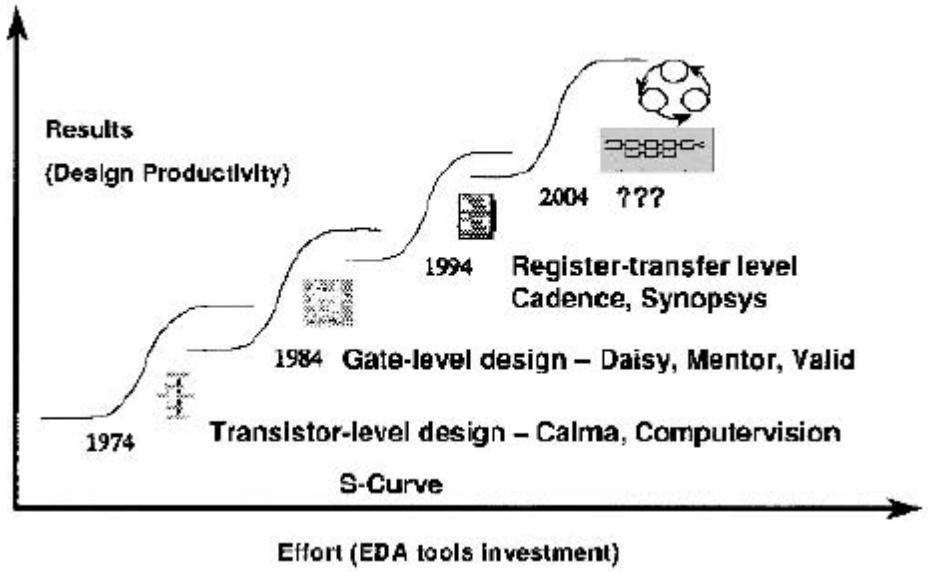


Figure 2 - EDA Design Discontinuity [1]

If the past history is indication of the future the semiconductor industry is due for another increase in productivity. There is significant commercial push for better design productivity as to take advantage of the contemporary semiconductor technology.

The question is what directions EDA industry need to follow. The recent trends point to:

- System Level Design
- IP Block Assembly
- Programmable Platforms

A number of EDA vendors have made significant investments in system level design tools. Some examples are Cadence and the Virtual Component Co-design (VCC) tool or Synopsys and the Behavioral Compiler, SystemC and System Studio. To this point in time it appears that system level design tools are not the next EDA direction simply because they rely on the underlining decaying RTL model. From [1]:

"As tool builders focused on raising the level of abstraction they failed to notice that the foundation at the register-transfer level was crumbling beneath them. In fact, even the foundation of the gate-level netlist, upon which register transfer level design had been built, was eroding."

IP Block Assembly is a design approach where a number of 3rd party IP blocks are interconnected via a system bus. Design reuse is very relevant and cost effective. However IPs are still built as RTL blocks and any costumatization (a frequent requirement) still requires modifying RTL.

Programmable Platforms offer a design productivity increase (programmability) with expense of significant decrease in performance comparing to more aggressive design methodologies. These platforms cannot be made only by assembling SoC from RISC and CISC architectures or they will not be able to compete with ASICs.

In the last few years (after the Internet bust 2001/02) ASIP have penetrated network and communication markets – traditionally ASIC arena.

Makimoto's Wave

Dr. Makimoto developed a "wave-theory" of a continuous exchange of domination between customization (ASIC) and standardization (ASIP, programmable platforms), Figure 3.

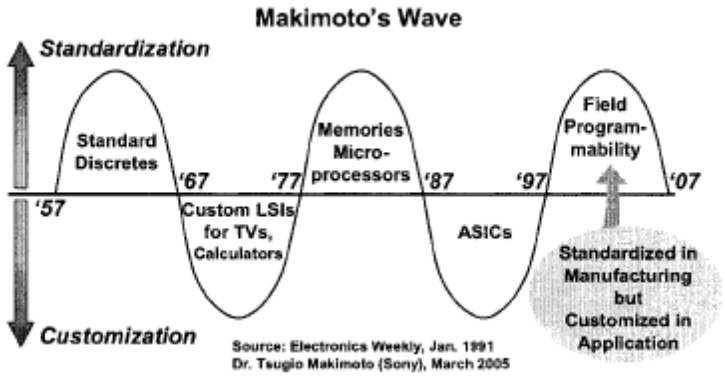


Figure 3 - Makimoto's Wave [1]

The standardization forces are:
- New device structures,
- New architectures
- Software development innovations

The customization forces are:
- Improved design automation
- Improved manufacturing process and testing

A methodology shift from ASICs to programmable devices was predicted to take place in 1997. Although no chart can take into consideration complete world economy (Internet bust in 2001, credit crunch 2008 etc.) we could conclude that the switch from ASICs to programmable devices is taking place, albeit with some time offset.

ASIP - Industry Adaptation

ASIP have made a noticeable penetration in traditionally ASIC dominated networking and communication space. Networking space is dominated by ISA (instruction-set architecture) based ASIPs while communication processor follows a variety of styles including VLIW processors.

The following text surveys Tensilica and Arc processor core design houses and companies that licensed their technology.

Tensilica

Tensilica Co. is one of the most successful providers of configurable embedded processors. Tensilica licensed their cores to a number of major electronics industry players mostly in DSP, data networking and communication space [6]. If we focus on the data networking space only, the following companies believe programmable platforms can deliver sufficient performance:

Astute Networks
Astute Networks employs ten Xtensa processors in its 10-Gbit Pericles storage network processor, capable of handling a variety of protocols, including TCP, Fibre Channel and iSCSI, for a range of systems from storage switches, gateways and servers to converged Fibre Channel-Ethernet storage systems.

Bay Microsystems
Bay Microsystems has used the Xtensa processor to develop the Montego InterNetworking Processor, a programmable 10Gig network processor (NPU).

Broadcom

Broadcom uses the Xtensa processor in the CALISTO line of Voice over IP communications processors.

Brocade
Brocade is a licensee of the Xtensa processor

Cisco
Cisco is both a repeat customer of the Xtensa processor and an investor in Tensilica. Tensilica's Xtensa processors are used in the innovative 40 Gigabit per second (Gbps) Cisco Silicon Packet Processor used in the Cisco CRS-1. Tensilica's Xtensa processors are also used in the QuantumFlow Processor in Cisco's ASR1000 router. Quantum Flow Processor 40-core processor stateful traffic processor. CRS-1 40-Gbit line rate – 192 processors cores.

HISilicon Division of Huawei
The HiSilicon Division of Huawei licensed Tensilica's Xtensa processors and ConnX DSP cores for network equipment chip design (**February 9, 2010**)

Juniper Networks
Juniper Networks (previously known as NetScreen) uses two Xtensa processors in its NetScreen-ISG 1000 and 2000 security gateways. These are high-performance integrated security gateways that deliver scalable network access for enterprise, carrier and data center networks.

Marvell
Marvell uses the Xtensa processor in a range of products, including the Yukon family of Ethernet controllers, the LinkStreet family of SOHO router chips, the Horizon family of communications controllers, and printer engines.

NEC
NEC is both a licensee and an investor in Tensilica. NEC Laboratories America and NEC Corporation have Xtensa licenses. NEC's iStorage NV8200 network attached storage (NAS) appliance for enterprise data processing uses Xtensa.

NetEffect's 10Gbps iWARP Ethernet Channel Adapters
NetEffect's 10Gb iWARP Ethernet Channel Adapter (ECA) is the first adapter that fully implements the iWARP Ethernet standard, allowing data center managers to realize more than 10Gbps throughput using existing Ethernet hardware and software, while radically improving processor utilization for applications that use networking.

Neterion
Neterion is the market leader in the 10-gigabit Ethernet adapter market, and is using the Xtensa LX processor for next-generation designs. "Tensilica's automated configurable processor design approach gives us the speed we need with lower power and smaller die size," stated Dennis Shwed, Neterion's vice president of hardware

engineering. “Tensilica’s Xtensa LX processor delivers the performance levels required for demanding 10-gigabit Ethernet in high-speed server and storage networking applications."

QLogic
QLogic has licensed Tensilica's Xtensa processor technology.

Sandforce
SandForce, Inc., is a semiconductor company developing System on Chip products that boost the storage I/O performance of next generation servers, workstations and personal computers by orders of magnitude. They are using the Diamond Standard 108Mini RISC controller core in their high-performance storage controller chipset designs.

Server Engines
Server Engines licensed Xtensa configurable processor cores for its BladeEngine 10Gb Enterprise I/O controller and other designs in progress for the 10-gigabyte enterprise Blade Server market. Server Engines’ revolutionary architecture, which incorporates multiple Xtensa processors, allows the integration of networking and storage into a single 10 Gb fabric, lowering capital and maintenance costs while providing state-of-the-art performance

Stretch Inc.
Stretch Inc. is delivering a new kind of software-configurable processor, the first to embed programmable logic within the processor. Stretch uses Tensilica Xtensa technology.

Valens Semiconductor
Valens Semiconductor selected the Diamond Standard 108Mini as the controller for a SOC design that will enable high-quality transmission of audio and video in a home networking environment.

Arc International (now Virage Logic)

Arc processors are widely accepted. These processors are somehow less expensive than Tensilica. List of licensees is longer than Tensilica however it appears that Tensilica processors are dominantly used in the high speed network processors. For the complete and up-to-date list of Arc users see [9].

Network Processors

Network Processors (NP) are programmable devices specialized in implementing data plane packet processing. NPs are often used in switches, routers or network adapters [18]. They are required to keep up with the incoming data rates of one or more media interfaces, perform multiple memory accesses required for packet storage and retrieval as well as table lookup and extraction of packet fields. NPs add HW architecture enhancements to counter the following general processor limitations:

- NPs cannot rely on data caches due to the nature of packets arrival (lack of locality)
- Minimize effects imposed by the memory latency
- Packet processing specific operation speedups(for example CRC calculations)

Therefore NPs are designed as multiprocessors supporting multithreading, with multiple memory hierarchies as well as separate processing for fast and slow/control data paths.

The following diagram shows a basic NP based packet processing system (Figure 4). NP is connected to packet ingress (Rx) and egress (Tx) ports. Ingress/egress ports are connected to the NP Media Access Controller blocks (MACs could be internal or external to NP).

On the Rx side NP processes incoming packets (packet data integrity/CRC, header parsing etc) and performs a routing lookup by accessing external lookup memories and matching packet header fields against values stored in lookup tables. Based on the lookup results packets are stored in relevant queues located un an external packer buffer memory. In addition the NP makes scheduling decisions and forwards packets from the packet buffer to the host memory subsystem utilizing DMA. NP scheduling algorithm needs to maintain packet ordering.

On the Tx side the host system sends packet data via the host I/F (PCIe) to NP. NP processes packet data (forms packet header, attaches CRC etc) and forwards packets to the Tx MAC.

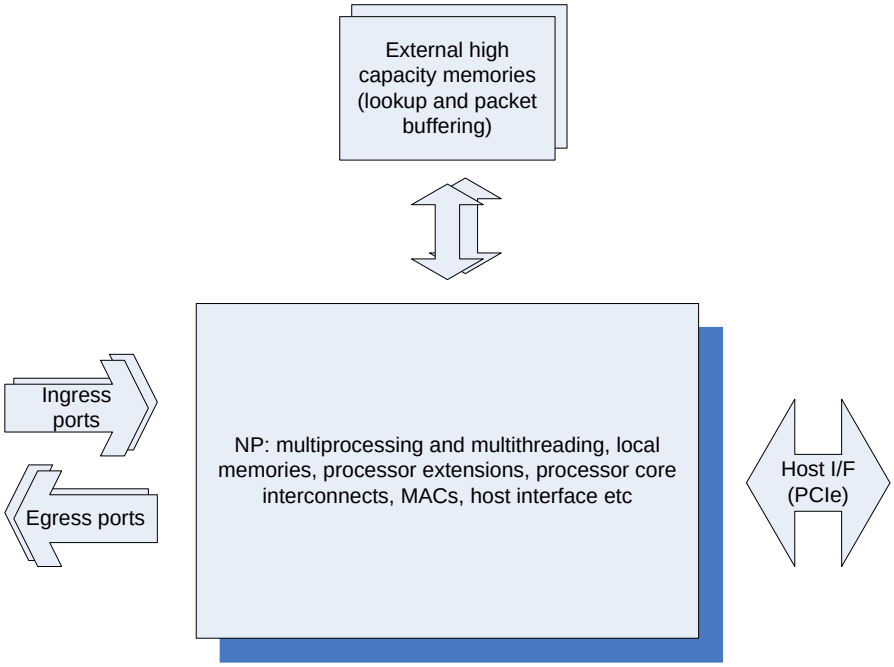


Figure 4 - NP based system

The main issue NPs need to deal with is the external memory access times versus incoming packet data rates. For example OC-48 (2.48Gbps) the worst case packet arrival time (minimum packet size) is 100 clocks at 600MHz NP clock rate or 160nS [18]. For 10Gbps Ethernet this time is in 50nS range.
On the other side SDRAM access time is in 150nS range (*). Therefore in order to maintain incoming packet data rate effects of the memory latencies have to be minimized. Off course in a multiport system (more than one ingress/egress ports) memory latencies are even more detrimental.

(*) – includes actual SDRAM page activation, bank refresh, data path latency, data retiming etc

One frequently used method to process packets at the 10GE data rates is to introduce parallelism to the NP architecture. The number of packet processing elements is increased so each Rx packet is distributed to a different processing element. The following picture depicts this scenario (Figure 5). Here the number of processing element is assumed to be 8 while the actual hardware configuration is built around one processing core with 8 threads.

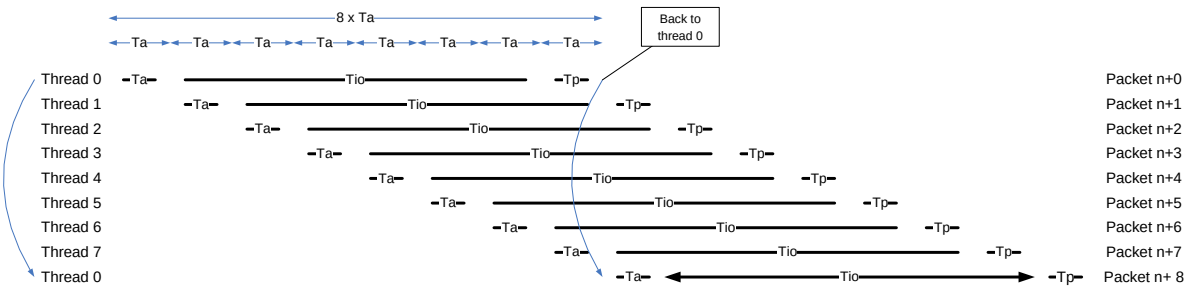


Figure 5 - Multithreading and packet processing

From the figure:

Ta – packet arrival time, each time period Ta a new packet is received (50nS for 10GB),
Tp – packet processing time (includes thread swap in/out in a multithreaded system, core code execution),
Tio – IO delay (memory latency).

Packet n+0 is assigned to the thread 0 (processing element 0). The core is executing thread 0 code and issues a lookup request (the lookup memory access). While thread 0 is waiting for the lookup results for the packet n+0, the processor core swaps it out in order to process packet n+1 that just is being received. This packet is assigned to the thread 1. While thread 1 is waiting for the lookup results it is being swapped out and packet n+2 is assigned to the thread 2, etc. If the thread 0 is completed with packet 0 processing by the time when packet n+8 arrives we conclude that this packet processing architecture meets the packet bandwidth requirements. Obviously this requirement is met if:

$Tp \leq Ta$

Also:

$Ta + Tio + Tp \leq 8 \times Ta$ or $Tio \leq 6 \times Ta$ $(Ta = Tp)$

The above could also be achieved with a multi-core system. However it should be noted from the above figure that only one core is really required since it processes packets in a pipeline fashion, one-by-one. On the other side, time to process individual frames Tp is limited to be less or equal to the minimum packet arrival rate Ta. In 10GE (10 Gbps Ethernet) processing time would be in 50nS range or 31 clock for 600MHz NP. For a multiport NP this number of clocks would be respectively lower. Obviously there are not enough clocks for the processor to do any in depth packet analysis. Therefore multicore NP architecture is required.

Maintaining packet ordering is an important aspect of NP architecture. In the first approach this order is achieved by assigning incoming packets to threads in order while threads are forced to complete in order by the processor core. This approach would work well for packets of the same length. If a mix of short and long packets enters the packet processing element and thread that is supposed to be the next is still processing its packet the complete ingress pipe would need to be stalled.
In the second approach packets are assigned to threads asynchronously and packets are assigned sequence numbers based on their arrival time. Packets exit processing elements in the sequence number order.

Typically there are two types of packet processing operations: fast path and slow path.

The fast path operations need to be executed as fast as possible in order to maintain the packet line rate (OC-48, 10GE etc). Example of fast path packet processing is the IPv4 forwarding (Internet Protocol version 4). Here destination IP address is extracted from a packet header and used to determine the egress (output) destination where this packet should be sent, IPv4 packet header field is updated and packet sent to its destination.

The slow path operations are required only in some cases while it takes longer to execute. Examples of the slow path processing are packets with some specific header options turned on.

In addition to data packets destined to fast or slow path there is also a need to process control packets. The control packets for example are required to update packet routing tables.

In the following text the Intel IXP 2400 network processor is used as a case study (Figure 6).

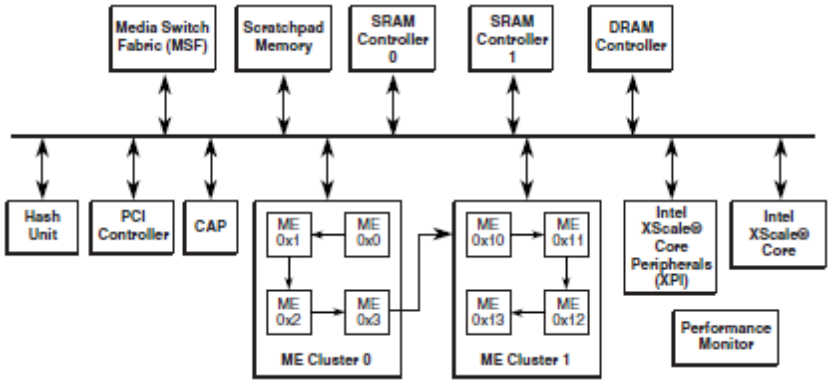


Figure 6 - Intel IXP 2400 NP block diagram [19]

Intel IXP NP is designed as a distributed multiprocessor architecture. The two major parts of the processor are:

- Xscale – a RISC based GPU used in slow and control path packet processing as well as in NP housekeeping tasks
- ME – specially built RISC based micro-engines used for fast path packet processing

Figure 7 portrays packet processing in IXP.

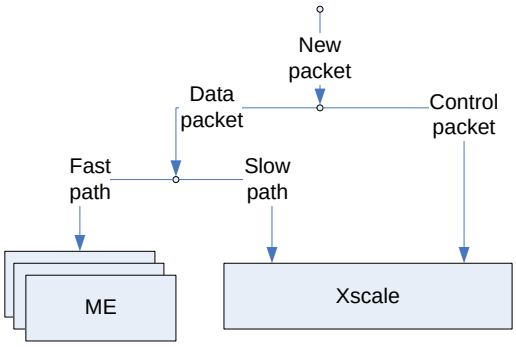


Figure 7 - IPX packet processing

IXP employs 8 ME cores where each core supports 8 hardware threads. MEs are programmable general purpose engines able to execute any type of packet processing. Assigning packets to different ME threads allows parallel packet processing (refer to Figure 5). ME threads are hardware supported contexts where each context is a set of registers. This allows easy context swap in with low core overhead. Threads are swapped out if there is a thread ready for processing and the current thread becomes stalled due to a pending IO access (a routing table memory read).

For fast packet processing using one ME does not suffice to meet ingress bandwidth requirements therefore multiple MEs are required. There are two ways of organizing multiple MEs based on how packet processing tasks are executed: pipelining and multiprocessing.

In the pipeline configuration each ME executes different tasks (Figure 8).

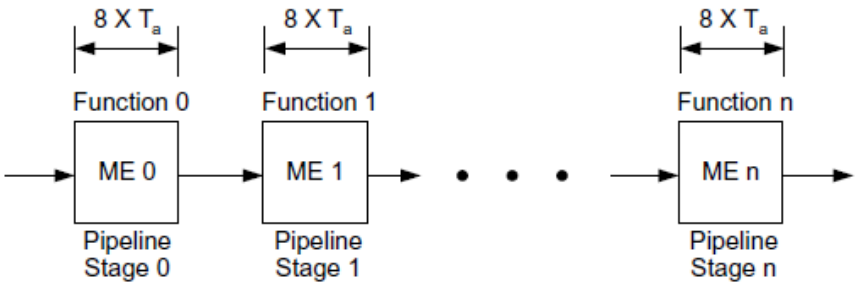


Figure 8 - Parallel Processing with Pipelined MEs [18]

As packets travel from ME 0 to other chained MEs different processing steps take place. As it was explained in Figure 5 each ME is allocated 8 x Ta processing time therefore for a pipeline on N MEs the total processing time is:

Pipelined MEs processing time: 8 x N x Ta.

Advantages and Disadvantages of the pipeline approach:

Pros:
- States required to perform a particular functionality can be held in a ME local memory.
- Complete ME program space can be allocated to one particular task.

Cons:
- States local to a packet (header updates) need to be communicated from a pipeline stage to the next stage. This information can be passed in line (packet header updates) or side-band (a separate interface).

In the multiprocessor configuration each ME executes all of the packet processing tasks. All MEs work in parallel and process packets simultaneously. In a configuration with N MEs, the total processing time is:

Multiprocessing configuration processing time: 8 x N x Ta

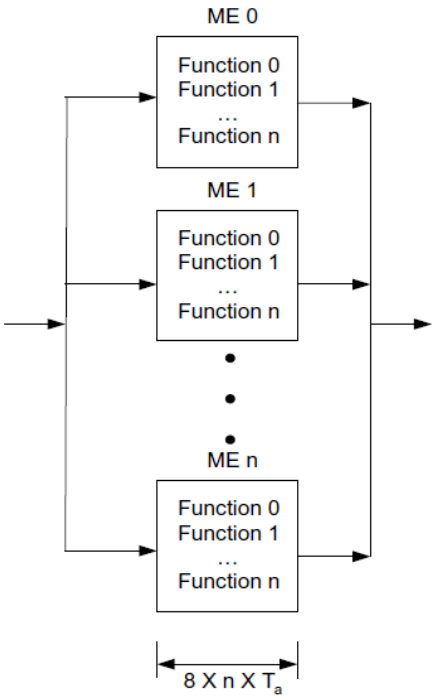


Figure 9 - Parallel Processing with Multiprocessing [18]

Advantages and Disadvantages of the multiprocessing configuration approach:

Pros:

- Packet state does not need to be communicated to other ME therefore it is kept locally to ME.

Cons:

- The ME program space is shared between multiple functions. This could be a bottleneck if many functions need to fit in the local instruction space.
- Function states used across all packets need to be stored in an external memory while maintaining the state coherence

In real applications a combination of pipelined and multiprocessing configurations are used. Packet forwarding from one ME to the next one is achieved utilizing FIFO structures. These FIFOs offer elastic storage feature allowing different processing speeds for different processing stages (required to support ME asynchronous packet processing). Therefore if one stage, consumer, is temporarily blocked or slowed down due to an IO access the elasticity buffer will hold on to the data allowing the producer stage to keep working till this temporary congestion is removed by the consumer.

While on the topic of threads it is of interest to look at the software threads running on a host system (a work station/PC using a general purpose processor). Writing SW based only on an interrupt level does not provide enough levels of processing priority.

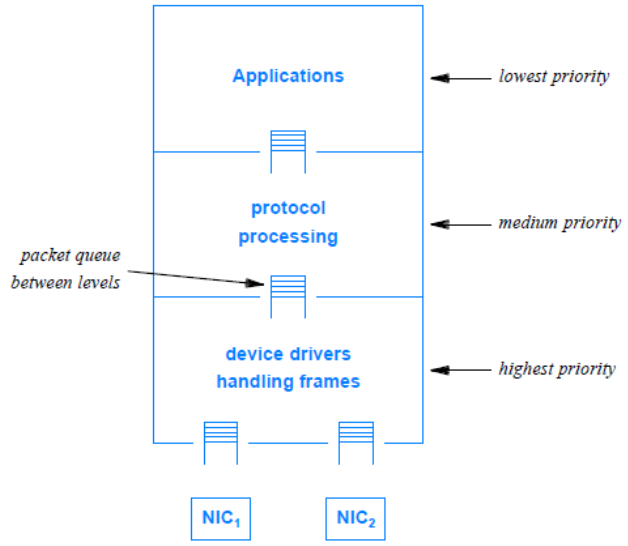


Figure 10 - Software processing priorities based on interrupt level [5]

From Figure 10, the protocol processing software has only one level of priority and that often is not enough. On some operating systems it is possible to write SW so parallel execution of different tasks (threads) is possible. These threads are called protocol processing kernel threads and they provide concurrent threads of execution. Each thread has its own program counter and an execution stack. The CPU swaps out SW threads, based on if they are active or not and swaps threads in based on their assigned level of priority. If no threads require processor attention, CPU executes an application program. All threads share the same address space therefore the thread synchronization is required as to avoid racing conditions when threads update or read shared data. Number of threads is practically unlimited (limited by available memory resources) allowing a high number of priorities to be used.

The most common thread configurations for layered protocols are:

- One thread per protocol layer,
- One thread per protocol,
- Multiple threads per protocol,
- One thread per packet etc

Each of the above scenarios has advantages and disadvantages.

In “one thread per protocol layer” (see Figure 11) configuration, the lowest layer threads are given the highest priority. In this case the thread that processes L2 protocol level has the highest priority. This is so the lowest level of processing, controlled by HW interrupts, the level responsible for IO operations (packet data transfer from a network adapter to the system) would get enough CPU clocks. This HW level cannot fall behind since packets could be dropped by the adapter.

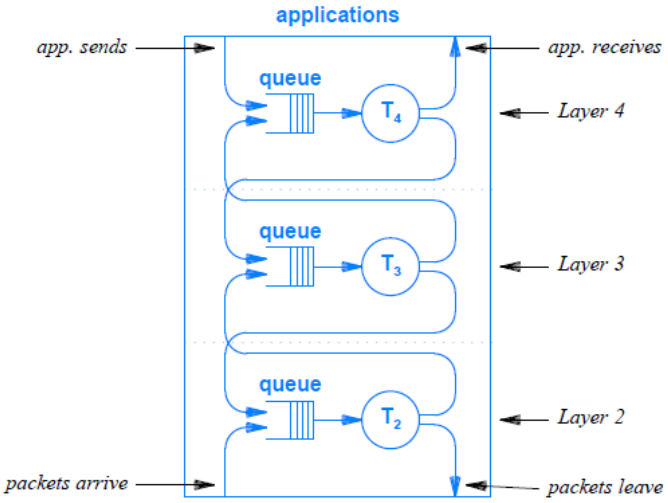


Figure 11 - One kernel thread per protocol layer [5]

Packets are passed between threads via packet queues; however the packet data is actually not moved. Packets are stored in the common address space; therefore only packet addresses need to be communicated between threads. In addition each thread processes Rx and Tx packets.

Some layered protocol stacks may have more than one protocol per layer. One example is TCP/UDP. In this case it may be easier to divide the processing software into separate threads where each thread is written for one protocol. The "one thread per protocol" configuration on Figure 12 supports TCP and UDP as two separate threads with the same processing priorities.

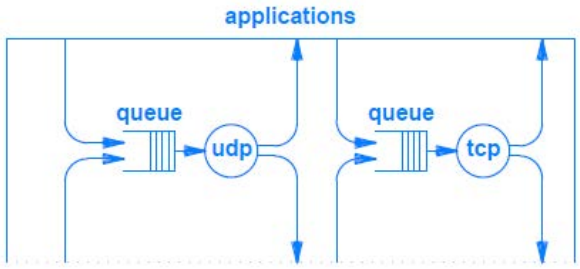


Figure 12 - One kernel thread per protocol [5]

In a HW versus SW threads summary it could be concluded that HW and SW could have similar thread structuring (per protocol, per packet etc) but the HW thread swap in/out has much lower overhead due to the dedicated context HW storage (registers) that is floor planned right beside the core itself. On the other side number of HW threads is bounded by the HW design, as for example it is for IPX with 8 threads per ME.

IPX uses distributed shared memory architecture. Besides supporting a number of ME local memories there is support for shared QDR SRAM and DDR SDRAM memory technologies.

QDR SRAM interface is used for fast access to lookup tables. QDR memories are SRAM based and offer low latency access but for higher component costs versus SDRAM. Considering Figure 5 where the lookup delay is a

dominant component of Tio, using fast access memories directly translates into NPs ability to free up ME threads for new packets (Figure 13).

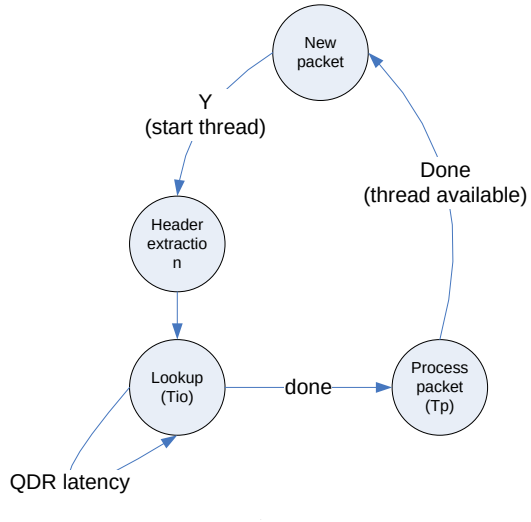


Figure 13 - Lookup Memory Latency

DDR SDRAM interface is used for high capacity, high bandwidth packet buffering. SDRAMs' high access latency is acceptable for packet buffering. This latency is much smaller compared to the system latency yet at the same time it does not affect when ME thread can be freed up.

In either of memory technologies ME do not control actual memories (open/close bank policies, refresh etc). ME interface with blocks responsible for low level memory control as well for memory access arbitration.

NPs add specialty hardware (processor extensions) to perform functions specific to packet processing. These extensions often offload host software from time consuming tasks or to speedup HW processing. One example of the SW offloading is CRC calculations. CRC is used in order to verify packet integrity; it is calculated by the packet producer and verified by the packet consumer.

CRC calculation is done over the packet length. If it is done in SW, CPU would need to bring in every packet word from the system memory to its datapath and perform the calculation. On the other side in the NP hardware it is much easier and faster to calculate CRC while the packet is being received by a processor core. One manageable design inconvenience is that some CRCs are stored as packet header therefore store and forward memories are required to assemble the packet header.

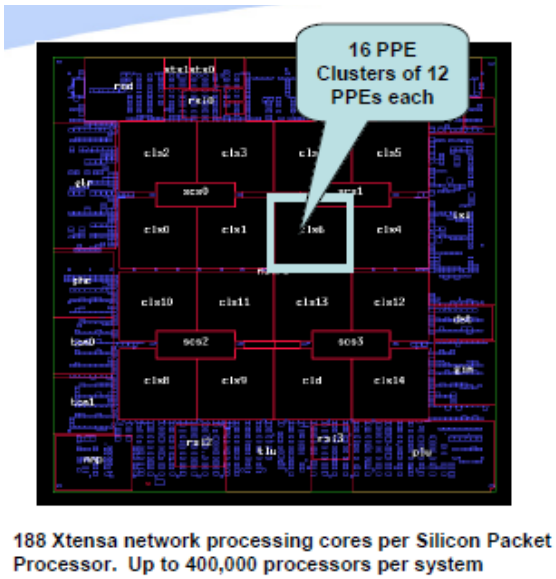
It could be generalized that every data path intensive calculations are much faster to complete in an embedded hardware such as a NP. Other examples are DES and AES used for encryptions and authentication – all implemented in IPX.

Number of processing elements used in NPs varies widely. Terminology used also varies. However at least in one industry study [20] the following is used:

- Multicore – more than one core
- Manycore – 10 cores and up
- Massively parallel – 64 cores and up

One interesting multicore (massively parallel) example is Cisco network processor using 188 Tensilica Xtensa PPE (Packet Processing Engine) cores (Figure 14). On the other side ASICs based packet processors (NIC) may have

only a few embedded processors used for slow path packet processing such is iSCSI - an IP based storage networking standard.
One of the newest and highly promising server processors is Sun Niagara UltraSPARC-T2. This is an 8 core, 8 threads per a core.



188 Xtensa network processing cores per Silicon Packet Processor. Up to 400,000 processors per system

Figure 14 Cisco NP

In appears that there is still a significant debate in semiconductor industry if the hardware level multithreading is beneficial [20]. DEC Alpha server processors and Ubicom embedded processors were the first to use hardware multithreading. Industry giants Intel used hyper-threading technology while IBM uses only 2 threads in its POWER6 processor. At this time Sun UltraSPARC-T2 is the leader in multithreading (total of 64 threads) with a strong believe that the memory latency problem is minimized. This comes with an assumption that there is a thread ready to go while the current thread is experiencing a cache miss.

A reasonable conclusion based on the above is that use of HW multithreads is here to stay, however the number of threads depends on the application. Packet processors coping with ever increasing packet bandwidth requirements having packets as individual unit of processing may take advantage of hardware multithreading well. However if memory latencies (or any other IO delay) are short due to having on-board fast memories the multithreading may not be beneficial.

Tensilica

Configurable Processors

Tensilica Co. is the major producer of RISC based soft processor cores. These cores are configurable via Xplorer processor configuration software tool. Some of options provided are selection (ways) and size of instruction/data caches, memory management options, use of multiply-add logic etc. In addition to configurability Tensilica processors offer a Verilog based tool (TIE) allowing addition of extensions to the core datapath.

TIE

TIE (Tensilica Instruction Extensions) is a set of rules, defined as a language, one would use to describe and add custom logic (extensions) to a processor core. These TIEs could be very simple (add a bit, a flip flop) or complex (many flops, arithmetic operations etc). For detailed description of available TIE instructions see [8]. In the following text we look at commonly used TIEs for data exchange in packet processing applications:

- Queues,
- Ports and
- Lookups.

In addition to this document also refer to [22] for more on useful packet processing TIEs and their use. It also should be noted that these TIEs are useful in inter-processor communications as well as in processor to RTL (custom logic) interconnects.

TIE Queues

Due to their complexity and speed requirements SoCs are designed as pipelines of functions. Data being processed, a packet, is moved from one functional pipeline stage to another while each pipeline stage performs different functionality. Pipeline stages could be individual embedded processors and/or dedicated RTL state machines. Individual stages may be with different processing speeds.

To effectively move data between functional pipeline stages providing enough data speed “cushioning” required due to different stage processing time (*) memory buffers are employed. These memories are instantiated between the producer and the consumer modules.

(*) This difference in processing speeds can be caused by the actual processing complexity or by the fact that different modules run in different clock domains.

Some consumers require a random data reads while some consumers process incoming data in the sequence of data arrival. DSP data streaming and most of packet processors process data in sequence. For these applications memory buffering can be implemented as FIFO (First In First Out) structures (Figure 15 b). For applications that require random data access (Figure 15 a) some form of shared memory is required (also see Processor Busses page 26, below). In some random access applications it is possible to pass out-of-order data in a side-band FIFO and streaming data in an in-band FIFO. One example would be packet based applications where consumers need to have deep packet before making packet processing decisions.

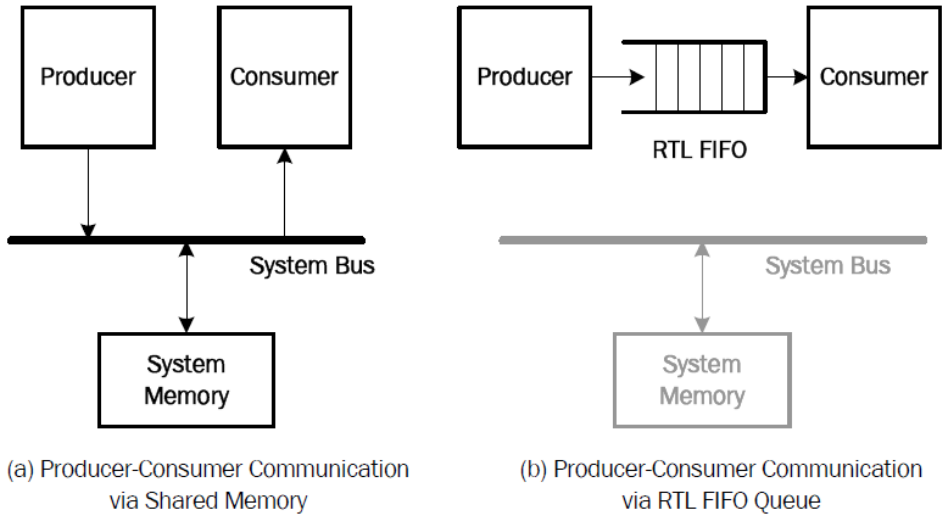


Figure 15 - Shared Memory versus FIFO Data Forwarding [14]

Tensilica TIE queues can be constructed as input or output queues (Figure 16). They also support an empty (input queue) and full flags (output queue). Removing an item from the input queue head is accomplished with a pop operation while adding an item to the output queue tail is done with a push operation.

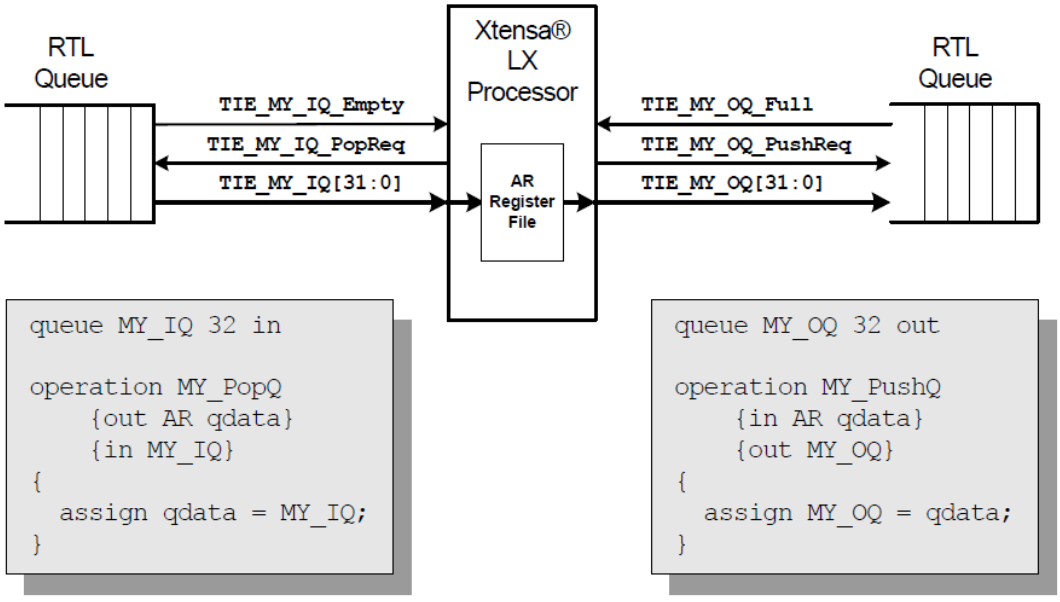


Figure 16 - TIE Queues, Flags and Push/Pop Signaling [14]

Actual Queues are not part of the processor or processor TIEs; they are RTL modules that need to be built or instantiated as IPs (see Data Queues on page 31 below and Figure 34 - TIE Queue Processor Connection Utilizing Synopsys Designware FIFO [1]). An example of input and output queue TIE code is shown in Figure 16.

Queue RTL modules need to support FIFO pointer logic: write index pointing to the FIFO tail and read index pointing to FIFO head. FIFO tail is the next free location where producer will push new data. FIFO head is the next data that the consumer will read from.

The following two figures depict a processor input and output queue RTL requirements. In either of the cases RTL needs to protect the FIFO from being over-run or under-run, provide full and empty flags etc.

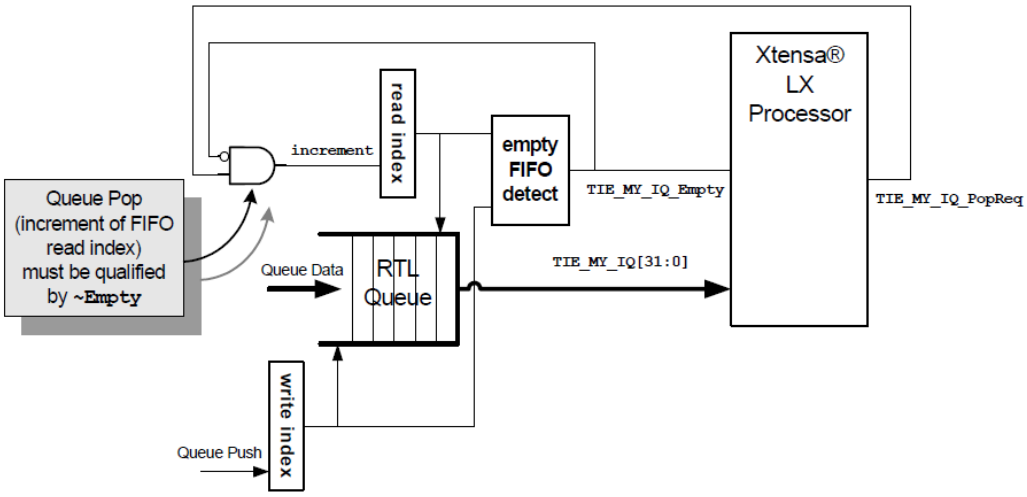


Figure 17 - Processor Input Queue Requirements [14]

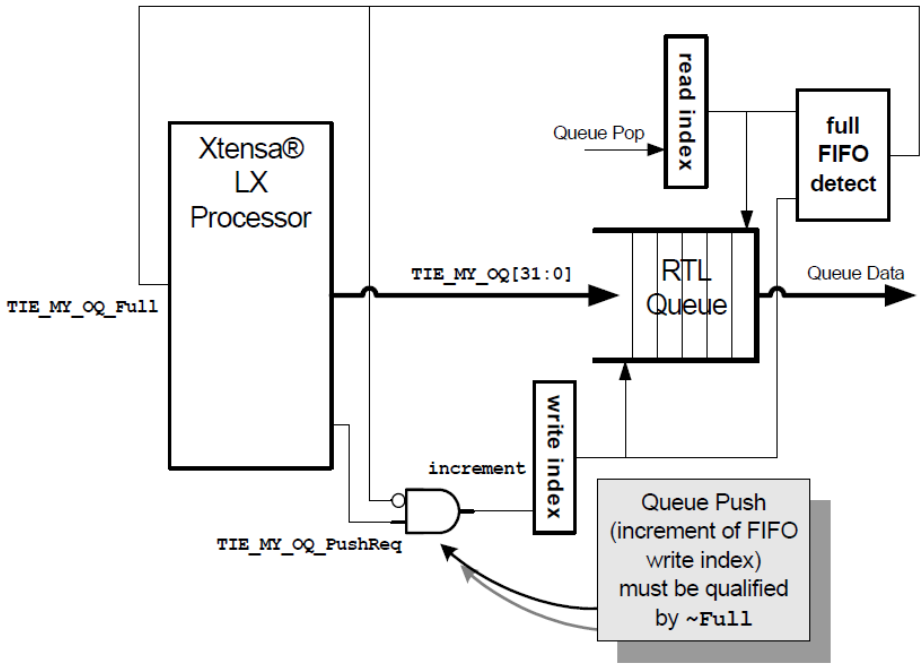


Figure 18 - Processor Output Queue Requirements [14]

TIE queues could be used as blocking or non-blocking structures. For example if an output queue is full and a push instruction is executed the processor pipeline is stalled till the full condition is gone (the full flag value is logic low) hence blocking condition. In a non-blocking scenario first an instruction that checks if queue is executed is used. If queue is full processor can continue executing different instruction and periodically check if the queue is not full. This approach makes more sense if the probability of the queue being full is high since a new instruction is introduced. If this probability is low (the consumer processing overhead is low) it makes more sense not to use this extra instruction, but stall processor if this full condition is to truly happen.

Note: it is worth noticing that Tensilica TIEs ([15]) allow defining instruction scheduling by using TIE instruction "schedule". A schedule is mechanism that can be used to define processor pipeline stages where TIE operation input operands are read and output operands are computed. For example:

```
operation foo {out AR out1, in AR in1} {} {
assign out1 = in1;
}
schedule schedule_foo {foo } {
use in1 1;
def out1 2;
}
```

Here the operation foo takes input in1 from the processor pipeline stage 1 (E stage) and places the output out1 at stage 2 (M stage in a 5-stage pipeline).

The following figure show another TIE schedule example where a multi-cycle instruction, an instruction that takes more than one clock is defined. This is often done to meet backend timing during the chip timing closure.

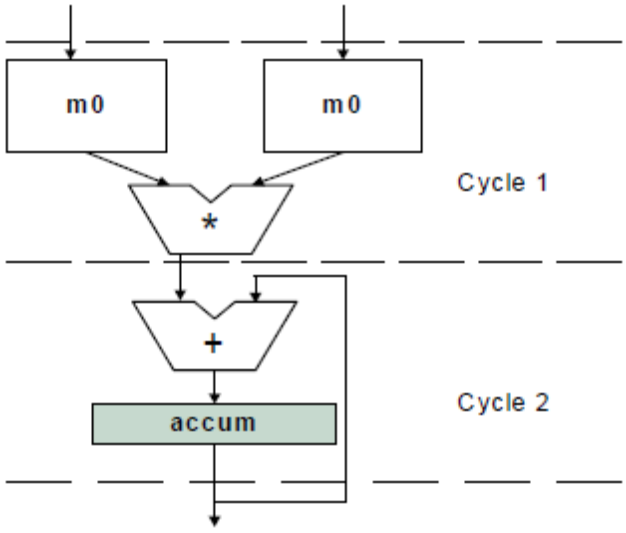


Figure 19 - processor Datapath Multi-cycle Scheduling

The following two figures show input and output queue TIE scheduling.

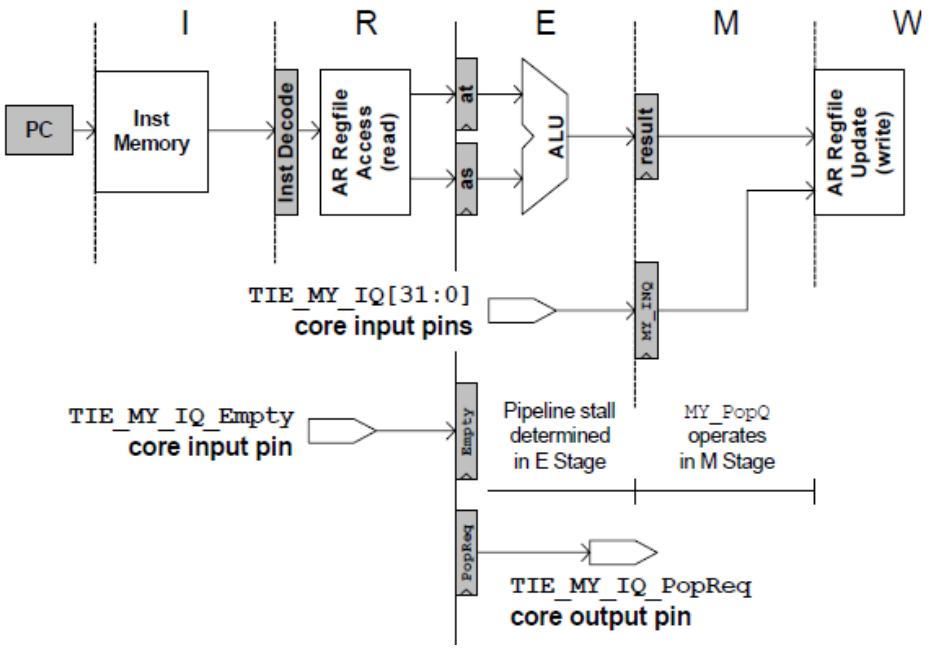


Figure 20 - Input Queue TIE Schedule [14]

The input queue scheduling (Figure 20): after a pop instruction reaches the E stage condition of the empty flag is checked. If the input queue is empty the processor pipeline is stalled till this condition is changed. If the input queue is not empty the queue data is popped out from the queue and advanced to the input of the M stage. Finally the M-stage data is committed to AR registers in the W stage.

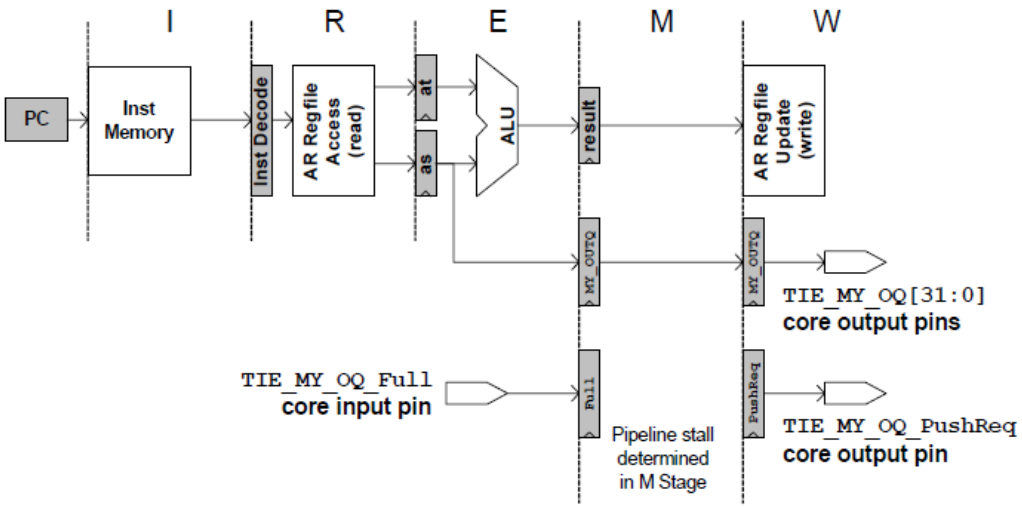


Figure 21 - Output Queue TIE Schedule [14]

The output queue scheduling is similar to the input queue except that Full flag is checked in M stage. If the output queue is not full output data is pushed from the W stage (Figure 21).

The following figure shows the concept of an input queue speculative buffering.

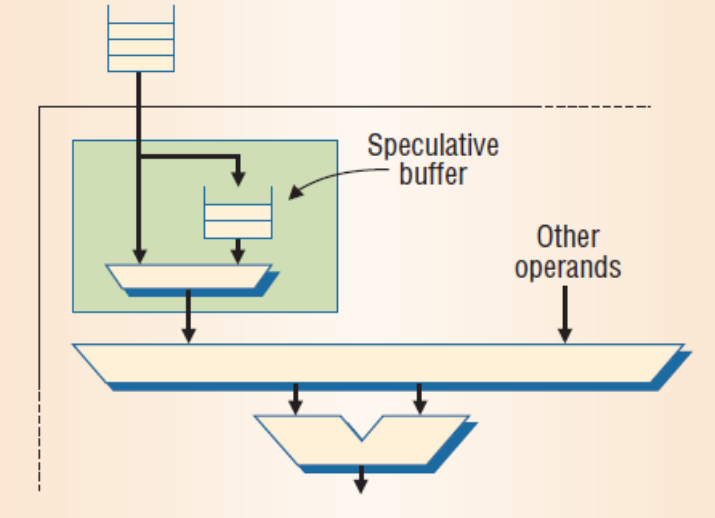


Figure 22 - Speculative Buffer Timing [17]

A queue read instruction pops the queue data and proceeds with the instruction execution. If an exception, interrupt or branch take place this queue data would have been lost. Therefore the input queue logic maintains a speculative buffer that keeps the queue data till the instruction is committed. If the instruction could not be committed due to the above reasons, the next time this instruction is to be executed the data will be taken from the speculative buffer.

TIE Ports

TIE ports could be used to interconnect processors. They could be thought of as a general purpose interconnects. If TIE ports interconnect processors output and input ports are required. However they could be used to drive external components (output TIE ports required) or read external component values (input TIE ports required).

To create output ports first a state (register) or states need to be generated. States could be used internally only however for a processor to output its state they need to be declared with a TIE keyword export. Similarly input wires (input ports) need to be declared. The following TIE code defines an 8b export state with reset value of 0. State name is SYSTEM_CFG_OUT.

```
state SYSTEM_CFG_OUT   8 8'h00 add_read_write export
```

The following TIE code defines an 8b wide import wire vector and relevant operation. The operation defines how to use this TIE construct from the C based program and what the actual operation functionality is:

```
import_wire  SYSTEM_CFG_IN 8
operation READ_SYSTEM_CFG {out AR system_cfg} {in SYSTEM_CFG_IN} {
        assign    system_cfg = {24'h000000, SYSTEM_CFG_IN};
}
```

For more information refer to [22].

TIE Lookups

TIE lookup construct creates output and input ports ([13]) used by one TIE operation. The output ports can be used to send a value. This output value could be a combination (concatenation) of an external memory read command and a read address. The input port could be connected to the external memory output. One TIE lookup operation

consists of sending a command/address to the memory and reading the memory output data therefore processors can have fast memory access. This access is much faster and could be wider than the standard 32b load/store operations.

The following TIE lookup construct takes a 32b AR register value “a” (an external memory read address/read command) and places it on the lookup output bus “LU_Out”. It also takes in LU_In value (an external memory read command output) and assigns it to a 64b “x” value:

```
lookup LU {32, Estage} {64, Estage+3}
operation doLU {out XR x, in AR a} { out LU_Out, in LU_In } {
assign LU_Out = a;
assign x = LU_In;}
```

The memory read data is available 3 clocks after a read command is placed on the TIE lookup LU_Out port. The read data is used in E stage. It should be noted that the lookup command is blocking, in this case for 3 clocks, till requested data is available.

Multi Processor Systems – Data Flow

Building multiprocessing system requires building processor interconnections. These interconnect need to be efficient in terms of bandwidth and IC real estate especially considering that modern SoCs could employ 10s to close to 200 processors per a chip.

Computation power of embedded processors is increasing while the data transfer between processors or processors to RTL blocks becomes bottleneck. Conventionally inter-processors or processor-RTL communications is supported by an on chip shared bus and a shared memory.

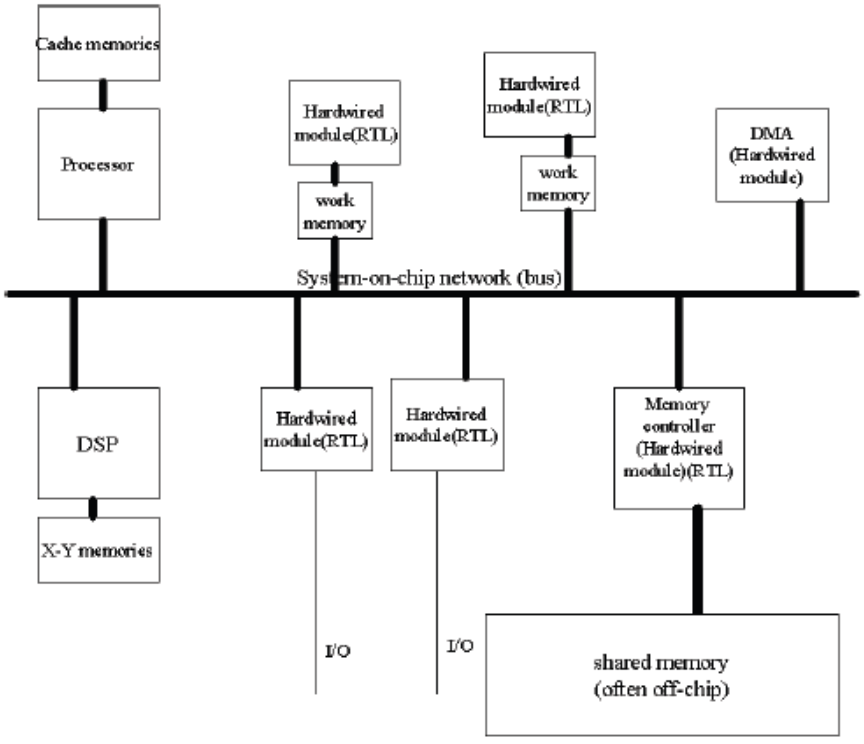


Figure 23 - Conventional SoC Architecture [16]

Data producers write data to be processed to a shared memory followed by an interrupt to a processor. The interrupted processor need to read data in from the shared memory. The system processing overhead becomes dominated by the shared bus access arbitration and interrupts mechanisms.

The following figure depicts the above scenario. After being interrupted the processor need to get access to the shared bus, output the shared memory address and read control signals, wait for the memory latency and read the data in.

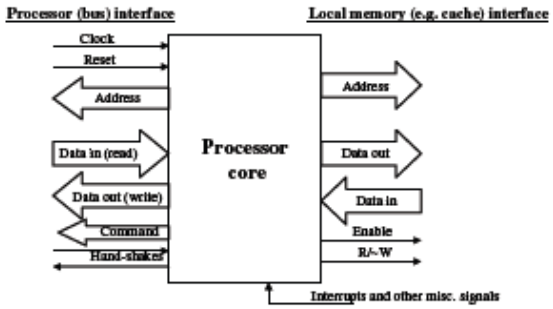


Figure 24 - Typical SoC Processor Interface [16]

In order to speed up the data exchange it would be very beneficial to avoid the shared bus and memory associated latencies. Such a system would be faster and would be based on the direct data exchange (Figure 25).

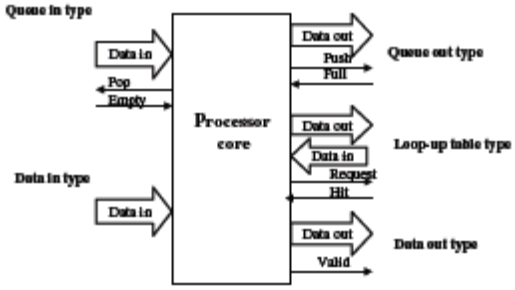


Figure 25 - Direct Dataflow Interface [16]

Chapters that follow give an overview on three major approaches how to share inter-processor data: busses, direct connections and queues.

Processor Busses

In this architecture processors communicate by busses with shared access. Bus access requires access arbitration. Often processors and slave devices have different bus widths limiting the processor bandwidth. On the other side processor can maximize their transfer rate if cache line block transfers are allowed.

When designing a system bus the following functionality need to be considered:

- Bus width and clock rate: The width and clock rate directly translate to the bus bandwidth but to power consumption and chip resources as well.

- Arbitration: arbitration mechanism affects the bus bandwidth (every time a bus turn-around takes place a number of clocks are potentially lost) and individual bus requestor latency. Some of possible arbitration policies are: strict priority, round-robin, weighted round-robin, deficit weighted round robin etc.

- Transfer type

    - Fixed block transfers: power of two blocks
    - Variable block transfers: arbitrary length transfers
    - Split transactions: read requests are split in two, one access to send read request (address, how many reads if variable block transfer etc) and number of returned read data phases. The goal is to have more than one split transaction.
    - Atomic transactions: a locking mechanism is used when one bus requestor locks the bus after having bus access granted. This is done to make certain that processors would not read data that is about to change by the processor that is currently have the bus access.

Traditional processor data exchange utilizes shared memory busses. Some of the often used are:

- Remote global memory with a general processor bus
- Local memory with a general processor bus
- Multi-ported local memory with a local processor bus

Remote global memory with a general processor bus

Global bus, shown on the next figure, supports a number of different transaction types. When processor determines that read requests are not for the local memory (a cache miss or address space points out of local memory) it needs to make a non-local memory read request. First it requests the global bus access. After the access is granted the processor places the request by sending the target read address. The addressed devices in return places read data on the bus.

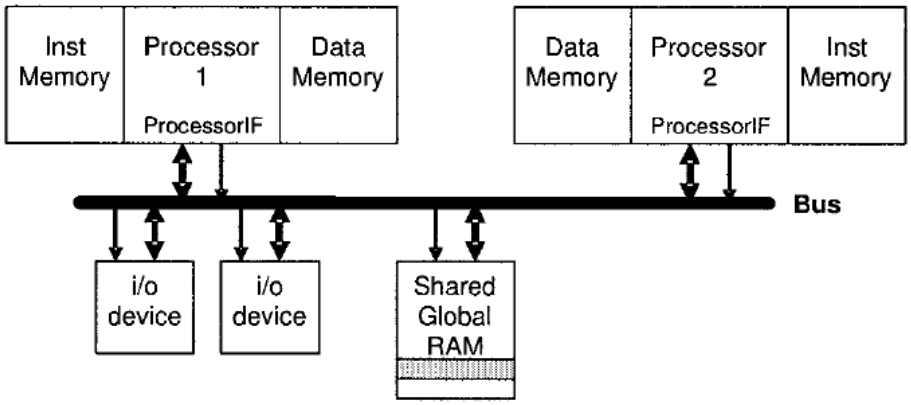


Figure 26 - Remote Global Memory [1]

Similarly when two processors communicate via a global shared memory one processor (producer) must get control of the bus and write data into the memory (see Figure 27). Consumer processor also needs to get control of the bus and read relevant data. Therefore for every data word that processor exchange two memory accesses are required.

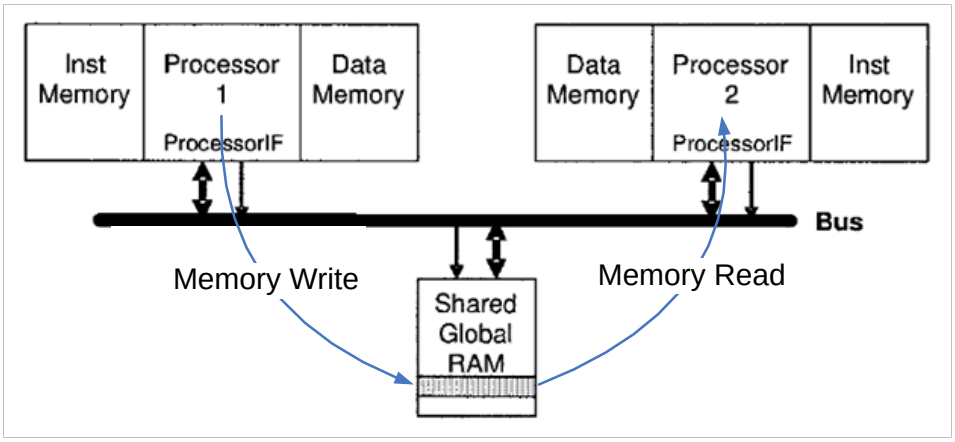


Figure 27 - Remote Global Memory and Processors data Exchange [1]

The main disadvantage of global shared memory (beside double memory access) is that this architecture does not scale well if number of consumers is more than one. The global bus arbitration becomes costly due to the arbitration overhead.

Local memory with a general processor bus

Some configurable processors allow external accesses to their local memory (see Figure 28). In this case Processor 1 wishing to read Processor 2 data over a general processor bus need to:

- Get the general processor bus access and
- After Processor 1 read request gets to the Processor 2, this request must be granted followed by an access to the Processor 2 local memory. Note this access needs to get the local memory access over a number of Processor 2 local access requests.

Therefore data read requested by Processor 1 may sustain a significant latency (due to the two levels of arbitration). If Processor 1 employs a push-like logic it could post (push) a number of Processor 2 local memory access requests. In this case the access latency is somehow hidden from Processor 1 since Processor 1 could do other tasks.

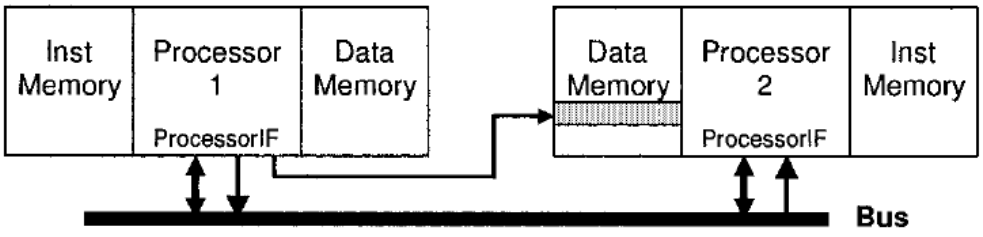


Figure 28 - Accessing Other Processor's Local Memory [1]

Multi-ported local memory with a local processor bus

In order to avoid global bus arbitration penalties (see "Remote global memory with a general processor bus") and/or arbitration penalties for accessing other processor local memory (see "Local memory with a general processor bus") some architectures use single or multiport local memories with a shared access.

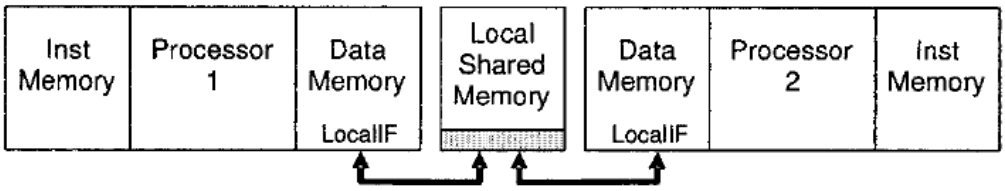


Figure 29 - Two Processors Sharing Local Memory [1]

If this local shared memory is a single port a simple arbitration mechanism is required. The (true) multiport memories do not require port arbitration and they offer two times higher system throughput versus single port memories. However on per a bit basis multiport memories are 2x larger then single port therefore they should be used only if required by the system throughput or if the size of the shared memory is small.

Direct Processor Interconnects

In the text that follows we consider direct processor interconnects as available in Tensilica processors: direct connect ports and data queues.

Direct Connect Ports

Direct processor communications allow one processor (producer) to directly place data into other processor (consumer) register space or even the execution unit without an intermediate storage. Inter processor communication overhead/latency is minimized.

The following figure illustrates a direct port interface. The producer writes operation results to its output register. This output register is wired to the consumer input port.

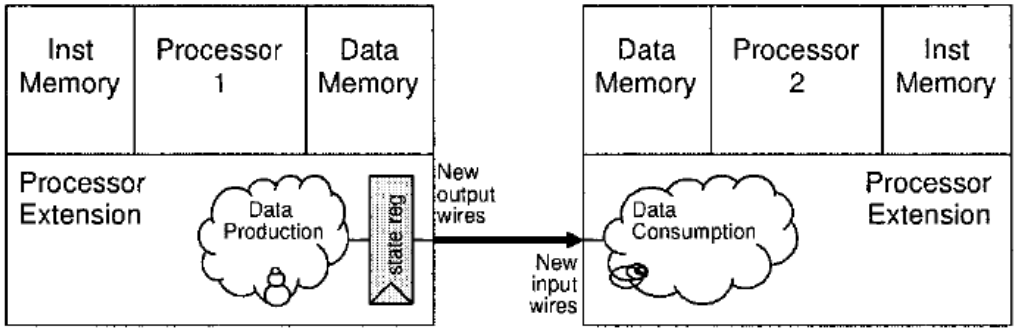


Figure 30 - Direct Processor Connection [1]

It is important to note that this direct interconnect does not need to be power of two nor 32b wide (standard RISC datapath/register width). Tensilica processors support direct port interconnects by providing export state and import wire TIEs ([22]) of an arbitrary width.

The producer processor needs to signal the consumer processor of the data availability. There are two basic approaches:

- Adding handshaking signals
- Interrupts

Handshaking signals need to inform the consumer of pending data as well as to signal to the producer that data was used and new data could be sent to the consumer. The following waveform depicts this two way signaling:

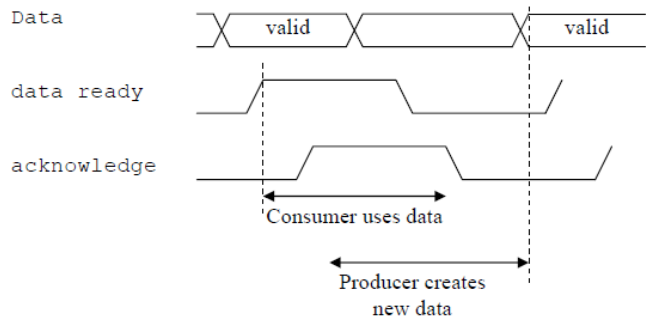


Figure 31 - Direct Processor Connection with Handshake Signaling [1]

The producer outputs the actual data as well as “data ready” signal. The consumer outputs an “acknowledge” signal to the producer. If previous data was acknowledged the producer outputs new data.
“data ready” and “acknowledge” are just another set of export states and import wire TIEs, the same as the data bus.
It should be noted that this approach does not allow a continuous burst of data. Each data phase consumes a number of clock cycles.

Interrupt driven handshake, see the following figure, requires two processors to exchange interrupts.

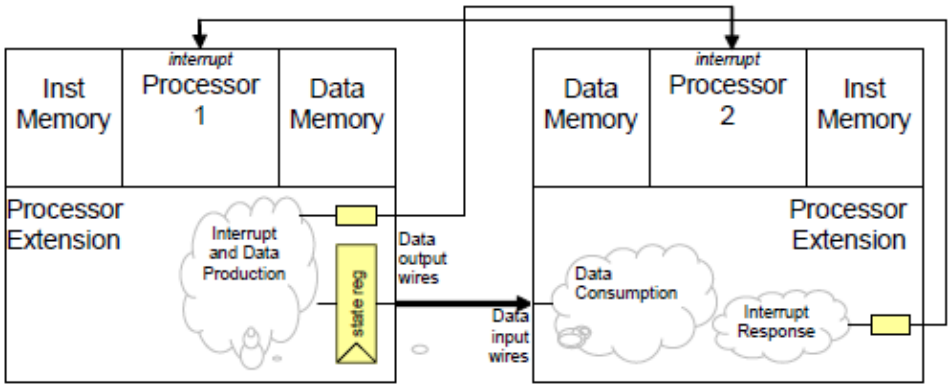


Figure 32 - Direct Processor Connection with Interrupt Signaling [1]

The producer asserts an interrupt after the data is output. The consumer will service this interrupt when possible (HW interrupts have higher priority than application code being run on the consumer). After data is processed the consumer interrupt handler asserts its interrupt connected to the producer. At this point the producer interrupt handler can place new data at its output.

Data Queues

Due to their low latency (signaling and data communication overhead), data queues are the fastest inter processor communication architecture. Data rate available are equal to the clock rate x data bus width. Therefore data rates in 10Gbps are achievable.

The queue width does not need to be the same as the processor data path nor it needs to be power of two. The interface itself takes care of the handshake (HW) and there is no requirement for SW to be involved. The producer pushes data into the queue tail with an assumption that the queue is not full. If queue is full the producer stalls (this functionality is applicable to Tensilica queue TIE). Similarly the consumer pulls the queue data out (pop operation) assuming the queue is not empty. If the queue is empty the consumer stalls.

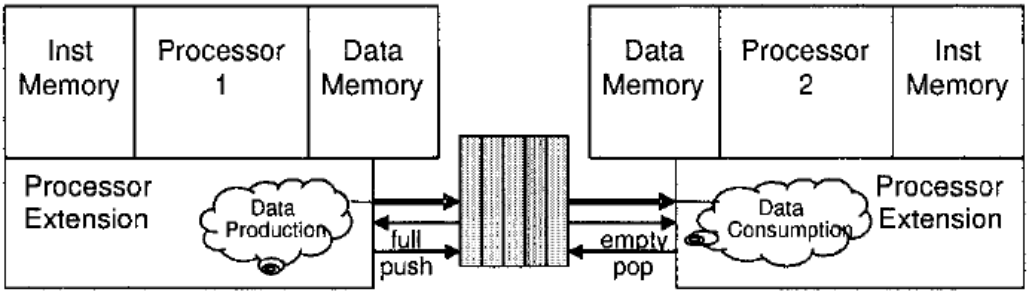


Figure 33 - Processor Connection Utilizing Queues [1]

The above described mechanism introduces stalls if queues are full or empty. If these stalls (blocking architecture) are not acceptable the producer or consumers could check if the stall is possible by checking queue empty or full flags (non-blocking architecture). Although these operations prevents possible stall they introduce new processor operations. Therefore full/empty checks are applicable to some applications. It should be noted that the processor stalls due to queues being full or empty are TIE supported (0-instruction overhead) while checking full/empty flags are software operations.

The following figure shows how to interconnect the queue TIE and Synopsys FIFO IP available (from Synopsys IP library called DW- Designware). This FIFO supports push/pop signaling and also provides full and empty flags. Due to the FIFO specifics a couple of gates are added to the interface, for example if the FIFO is full we gate push signal not to create FIFO error.

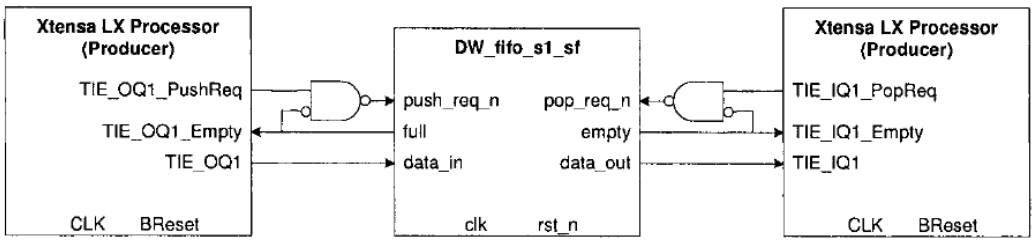


Figure 34 - TIE Queue Processor Connection Utilizing Synopsys Designware FIFO [1]

In “Remote global memory with a general processor bus” above it was noted that the global bus architecture is problematic if a number of consumer processor grows. The following figure shows a multiprocessor interconnect topology that scales well.

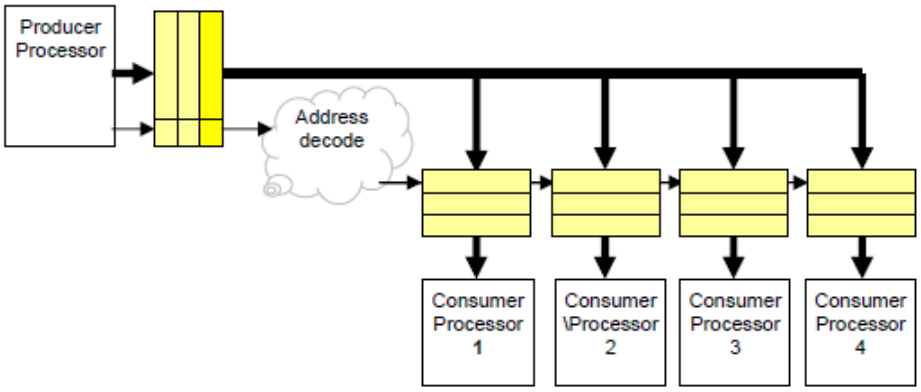


Figure 35 - Producer Creates Data and Address [11]

Here each consumer processor does the address decoding (a destination specifier is added to the actual data) to determine if arriving data is for its queue while the producer pushes output data into a common queue. Consumer queues are optional however their use is recommended so slow processors would not stall other processors.

A different approach of accessing queues would be by using the local memory interface and memory mapped queue. If the producer local memory address is the same as the queue memory mapped address the data push operation takes place (see figure that follows).

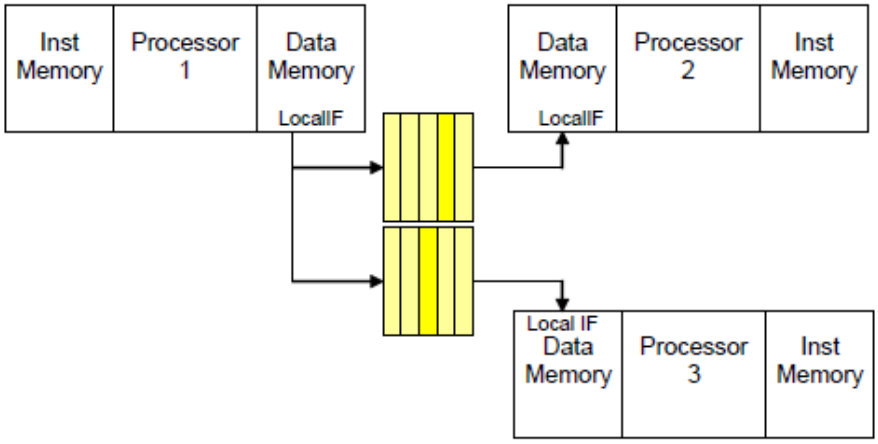


Figure 36 One Producer and two Consumers with Memory Mapped Queues [11]

Similarly if the consumer load operation address is the same as the queue memory mapped output a pop operation takes place.

Queue size selection especially for wide data paths affects chip costs. Therefore queues need to be sized appropriately. Too big queues are costly, too shallow queues negatively affect performance due to the queue full conditions. Sizing a queue requires good system level knowledge as well as running relevant system level simulations.

In some instances it is possible to have queues as small as one data word deep, as shown on the next figure, sometimes referred to as mail-boxes.

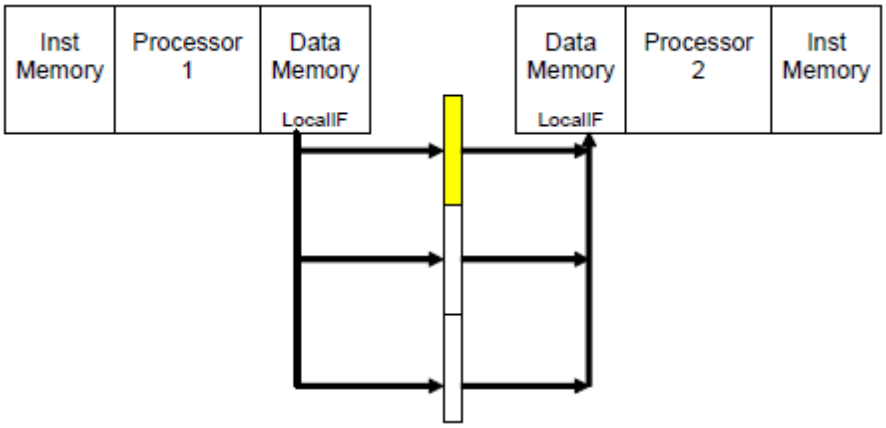


Figure 37 - Memory Mapped Mailbox Registers [11]

Flow-through processing

Flow-through processing ([22] and [1]) can be defined as a design functionality where data to be processed is input to a functional/processing unit, processed and output with a pipelined throughput of one clock (*). An opposite architecture is a store and forward architecture. In the former case data need to be stored into a memory and pulled out from the memory to be further processed and output, a classical generic processor approach. Store-and-forward architecture throughput is smaller than the flow-through architecture.

(*) pipeline throughput here means that data is being input, processed or output on every clock, while the same piece of data will have the latency of more than 1 clock.

The following figure shows a Tensilica flow-through processor build from a standard core equipped with queue TIEs. Input data from the 2 input queues is combined and output via an output queue. The processor datapath is skipped while the data throughput is equivalent to the data path width multiplied by the clock rate.

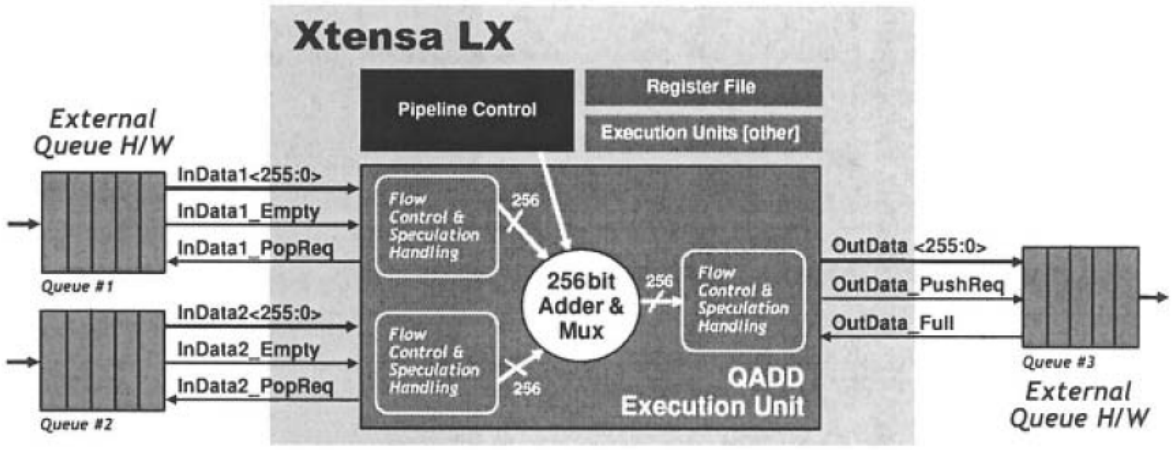


Figure 38 - Flow-through processor Error! Reference source not found.

In addition to allowing building flow-through processing unit the TIE queue also allows very efficient inter-processor communication. Rather than using a shared data bus and local memories processor exchange data in continues fashion employing the TIE queue.

This flow-through processing in a multi processor environment applicable to packet processing is the topic of this project.

This project report documents all development steps taken during a packet routing system design. The complete system is designed using Tensilica embedded processor cores. The system is made from 5 main processors running custom C code. All processors, processor memories, general purpose IOs, queues and a lookup device are connected, modeled and simulations completed using Tensilica XTMP modeling protocol.

References

[1] Matthias Gries and Kurt Keutzer, Building ASIPs the Mescal Methodology, *Springer, 2005, ISBN-13: 978-0387260570*

[2] Jari Nurmi , Processor Design System-on-Chip Computing for ASICs and FPGAs, *Springer Netherlands, 2009, ISBN-13: 978-9048173853*

[3] Tilman Glökler and Heinrich Meyr, Design of Energy-Efficient Application-Specific Instruction Set Processors (ASIPs), *Springer US, 2010, ISBN-13: 978-1441954251*

[4] Tilman Glökler, Andreas Hofmann and Heinrich Meyr: Methodical Low-Power ASIP Design Space Exploration, Journal of VLSI Signal Processing 33, 229–246, 2003 Kluwer Academic Publishers

[5] Douglas E. Comer, Network System Design using Network Processors, Pearson Education

[6] Tensilica ®: List of Costumer Profiles - http://www.tensilica.com/company/customer-profiles.htm

[7] Tensilica ®: http://www.tensilica.com/markets/customer-gallery/network-infrastructure.htm

[8] Tensilica ®: Instruction Extension (TIE) Language, Reference Manual

[9] ARC International: http://www.arc.com/licensees/

[10] David Goodwin, Tensilica ®: "Customized Processors, Customized Interconnects A Requirement for Next Generation Embedded Systems" , DAC 2007

[11] Tensilica ®: Get Your ASICs and SOCs Off the Bus

[12] Tensilica ®: Implementing FIFO Operations Using TIE Queues, Application Note, October 2007

[13] Tensilica ®: TIE - The Fast Path to High Performance Embedded SOC Processing ,Application Note

[14] Tensilica ®: Using TIE Queues with Xtensa LX Processors, Application Note, April 2010

[15] Tensilica ®: Area-Efficient TIE Generation Using the Schedule Construct, February 2009

[16] Tensilica ®: A New Kind of Processor Interface for a System-on-a-Chip Processor with TIE Ports and TIE Queues of Xtensa LX, Tomonari Tohara

[17] Steve Leibson and James Kim, Tensilica ®: Configurable Processors, A New Era in Chip Design, July 2005

[18] Muthu Venkatachalam, Prashant Chandra, Raj Yavatkar: A highly flexible, distributed multiprocessor architecture for network processing, Intel Communications Group, Intel Corporation, JF3-462, 2111 NE 25th Ave, Hillsboro, OR 97124, USA

[19] Intel®: IXP2400 Network Processor, Datasheet, Feb 2004

[20] Tom R. Halfhill: "The Future of Multicore Processors" Dec. 31, 2007

[21] Richard Goering, Behavioral synthesis revived for ESL design,EE Times, 5/10/2004

[22] Josip Popovic, Project Report II, Data Network Router Design with Embedded Tensilica Processors Utilizing Multiprocessor XTMP Development Environment. SYSC 5906, Summer 2010, Developed for Professor Bolic, SITE, University of Ottawa