TECHNICAL UNIVERSITY OF CRETE

ELECTRONIC AND COMPUTER ENGINEERING DEPARTMENT

Thesis:

Design and Implementation of a Network Processor Input/Output Module

By Ioannis Sourdis

Supervisor:	Assistant Professor Dionisios Pnevmatikatos
Members:	Professor Apostolos Dollas Assistant Professor Georgios Stamoulis

CHANIA, FEBRUARY 2002

...στους γονείς μου.

...στην Κορίνα.

...στη Χριστιάνα.

Γιάννης Σούρδης

Ευχαριστίες

Για την ολοκλήρωση της διπλωματικής μου εργασίας θα ήθελα να ευχαριστήσω τον Μάρκο Κιμιωνή και όλους τους μεταπτυχιακούς και προπτυχιακούς φοιτητές του εργαστηρίου για τη βοήθειά τους .

Επίσης θα ήθελα να ευχαριστήσω για τη συνεργασία μας στο Protocol Processor Project τους Γιώργο Κωνσταντουλάκη, Φάνη Ορφανουδάκη, Κώστα Πραματάρη, Κυριάκο Βλάχο, Christoph Baumhoph, Νίκο Νικολάου, Στέλιο Περισσάκη, Prabhat Avasare, Jorge Sanchez, Νίκο Ζερβό και τα υπόλοιπα μέλη των συμμετεχόντων στο Pro3 (Lucent Technologies Nederland, Hyperstone Electronics, IMEC, Ellemedia Technologies και inAccess Networks, ΕΜΠ).

Ευχαριστώ ακόμη τα μέλη της εξεταστικής επιτροπής Γεώργιο Σταμούλη και Απόστολο Δόλλα για τη πολύτιμη βοήθειά τους.

Θα ήθελα ακόμη να ευχαριστήσω τον Αποστόλη Δημητρομανωλάκη και το Γιάννη Αικατερινίδη για τη βοήθειά τους.

Ευχαριστώ επίσης τον καθηγητή Απόστολο Δόλλα και τον Παναγιώτη Στογιάννο για το σημαντικό ρόλο που έπαιξαν στην απόφασή μου να ασχοληθώ με το Hardware!

Ακόμη ευχαριστώ το Διονύση Ευσταθίου και το Γιάννη Ζήση για την άριστη συνεργασία μας στα πλαίσια του ερευνητικού προγράμματος Pro3.

Τέλος θα ήθελα ιδιαίτερα να ευχαριστήσω τον επιβλέπων καθηγητή μου Διονύσιο Πνευματικάτο για τις χρήσιμες υποδείξεις και την άριστη συνεργασία μας.

TABLE OF CONTENTS

TABLE OF FIGURES

Abstract

The rapid growth in the network sizes, the continuing trend for convergence of data, voice and video traffic and the tremendous growth in data traffic (particularly that associated with the Internet), has demanded the development and deployment of high-capacity telecommunication systems. Transmission lines with speeds of several Gigabits/sec are already available. To meet the protocol processing demands of the fast lines, we must employ high-performance protocol execution engines. A *network processor* aims in accomplishing wire-speed protocol processing, at a reasonable cost. The Protocol Processing Project (Pro3) aims in developing an integrated and affordable system, which is able to execute high-level telecom transport protocols (TCP/IP, Network Address Translation, Packet filtering, etc.) at wire speed. This thesis's goal is the design and implementation of the Protocol Processing Engine (PPE) module of Pro3. PPE interfaces the internal datapaths of Pro3 with the central processing core that executes the protocols. We organized PPE in a three-stage pipeline (input, process and output) in order to improve its performance. The pipeline processing of packets gives rise to data hazards that must be handled. The correctness of the design has been initially confirmed with "ad-hoc" testing. We then employed a systematic verification approach. First, we automatically generated input files and their corresponding results. Then, we performed simulations and finally automatically verified the simulation results against the expected ones. The Hardware Description Language (VHDL) source code of the design was compiled (synthesized using the Synopsys Design Compiler), and linked to a netlist, which was also verified using gate-level simulation. Each module was initially verified individually (functional and gate-level simulations), and then with its neighbouring modules, and finally with the rest of the Pro3 modules in chip-level simulations.

Chapter 1: Introduction

The trend of data, voice and video traffic convergence and tremendous growth in data traffic, particularly that which is associated with the Internet, has prompted many discussions on the hardware and software needed for the future. Newer bandwidth-eager end-user software applications and faster processors in desktop and server systems are placing enormous demands on the current networking architecture. As a result, the network is evolving into a highly complex and sophisticated environment: networking bandwidth use continues to double every four months; guaranteed quality and priority customization is anticipated to all data, voice and video applications. The way telecommunication systems work is that networks run packets and software is the key in making networks work.

The rapid growth in the dimension of networks, along with the always-increasing user's demands for networking services, has imposed the development and deployment of high-capacity telecommunication systems. Such systems involve modules of high throughput, which have their time critical functions realized in application specific standard products. The power required for the processing of protocol functions at wire speed is usually obtained either by generic microprocessors or by Application Specific Integrated Circuits (ASICs). Both approaches have advantages and disadvantages. General-purpose microprocessors are designed to perform a variety of functions, but suffer from reduced performance. With time-to-market becoming the dominant force in the networking world, many companies have turned to Reduced Instruction Set Computing (RISC) technology. RISCs can execute their code very fast due to instructions simplicity, but since the decisions are made in software, the RISC is much slower than the ASIC. On the other hand ASICs are designed to meet a specific functional requirement with high efficiency, but are very difficult, if not impossible, to change once they have been designed or modified with a simple software upgrade (chapter 2). Another option is a hybrid approach. Microprocessors are inadequate in supporting the protocol processing requirements for the entire set of active sessions. This constitutes a major system resource bottleneck, because the complexity of the protocol algorithms requires higher computational power than that offered by today's processor technology [4, 17, 18, 19]. Hybrid approach combines both chip technologies, using a RISC processor as the central core, and ASICs to perform the specific tasks (Pro3 approach, chapter 3). These components called Network Processors have exhibited an enormous advance in the turn of the millennium. As for the higher layer protocol functions that are not performed at wire speed, often today, more than one high performance processing units are employed; these processing units include routing protocols, statistical compiling and reporting, error processing, connection admission control, network and transport layers protocol processing and traffic and resource management.

In the last few years, telecom industry has witnessed an enormous advance in processor technology, designs, capabilities and applications [4]. Because of their flexibility, general and specialized processors are being developed by many industries. Powerful architectures (i.e. RISC), specialized processors, signal processors, graphics processors etc. are developed. *Network processing architectures* [7, 8, 9, 10, 11, 12, 13, 14] are also developed in order to achieve wire rate protocol processing.

Protocol Processing Project (Pro3) [5] is enhanced with hardwired functionality devoted to speed up low level streaming and networking operations. Pro3 has to do with the design and implementation of a network processor. It includes RISC processors and ASIC in a single chip, trying to perform in wire rates. Our design, input/output module of Pro3 network processor, is responsible of putting incoming packet information to a RISC core (protocol processor) and taking out updated packet information. Its goal is to improve performance of the Protocol Processing Engine (PPE) by creating a three stages pipeline (input, process and output).

Generally, the following pseudo code describes packet processing:

```
For each packet {
   1. Identify connection ID (flow), packet classification
   2. Get state information (i.e. last packet seen, etc.)
   3. Consult selected fields (parts of header, body)
   4. Execute protocol code on state and selected fields
   5. Update (?) packet and flow state
   6. Send (?) updated packet
   7. Create (??) other control packets
}
```

When a packet arrives, network processor identifies its flow (packet classification). It obtains the flow state information for the corresponding flow. The flow state information consists of the data needed to correctly process an arbitrary packet of that flow. For example, flow state may include:

- The number of the last packet seen.
- The packets, which were transmitted, but are not yet acknowledged by the network processor.
- A list of timers' information (i.e. how long had it been since last packet arrived).

Protocol code is executed on flow state and selected fields. Packet fields that are needed for protocol processing are extracted to the modules that process protocols. Process results may be:

- Reject packet (firewall or there is acknowledgment that this packet has already got on its destination),
- Packet modification,
- Update flow state,
- Creation of additional control packets,
- Or combination of the above.

Chapter 2:Network Processing Architecture Alternatives

There are several different architectural solutions for packet processing. Each having some advantages and disadvantages. General-purpose microprocessors are used for their flexibility to adapt to protocol changes in the field and short time to complete the software development, but do not have enough performance to process data at wire rates. On the other hand, hardwired (custom) chips are designed to process packets at wire rates, but are not flexible. They are difficult and expensive to modify, to add features, fix bugs or adapt to rapidly changing network processors. Network processors are proposed as a solution to both these problems. They are processors optimized to perform packet processing at wire rates and are programmable and therefore flexible. There is a trade off between performance and flexibility in these three solutions (General-purpose microprocessors, Network processors, dedicated ASIC). General-purpose microprocessors are very flexible, but don't have enough performance, network processors are less flexible but have much better performance and finally, dedicated ASICs are not flexible but can perform at wire rates. This trade-off [16] is shown in figure 2a.

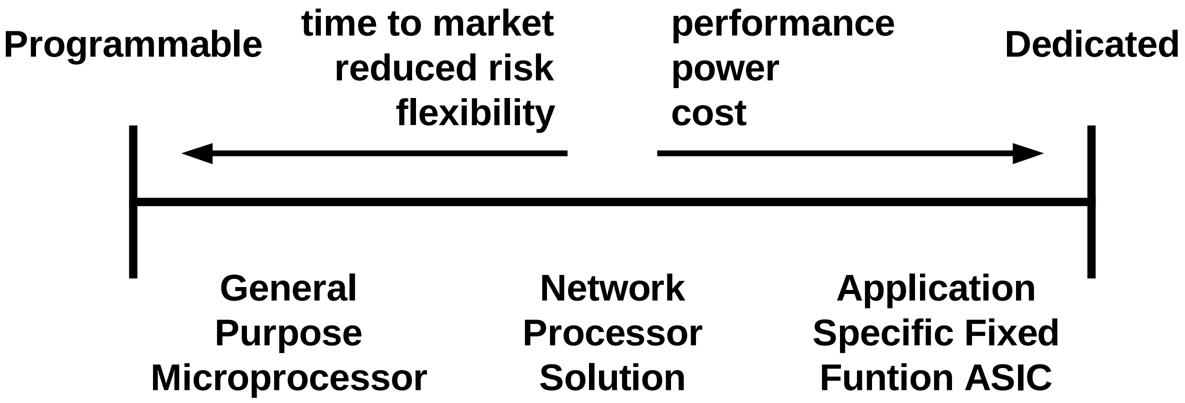

Programmable
time to market
reduced risk
flexibility
performance
power
cost
Dedicated
General Purpose Microprocessor
Network Processor Solution
Application Specific Fixed Funtion ASIC

Figure 2a: Programmable processors vs. Dedicated hardware

Through advances made in semiconductor technology, the philosophy of network system design has changed [21] (figure 2b). In 1995, networks employed traditional routing/switching devices, such as a general purpose CPU, a packet –processing engine, and a forwarding engine. By 2000, hybridized architectures were common. Such systems consisted of a general purpose CPU and a custom ASIC or an off-the shelf application- specific standard product (ASSP), or a combination of these devices. After 2001, technology-driven systems consist of a dedicated control CPU and a full-fledged application-specific network processor.

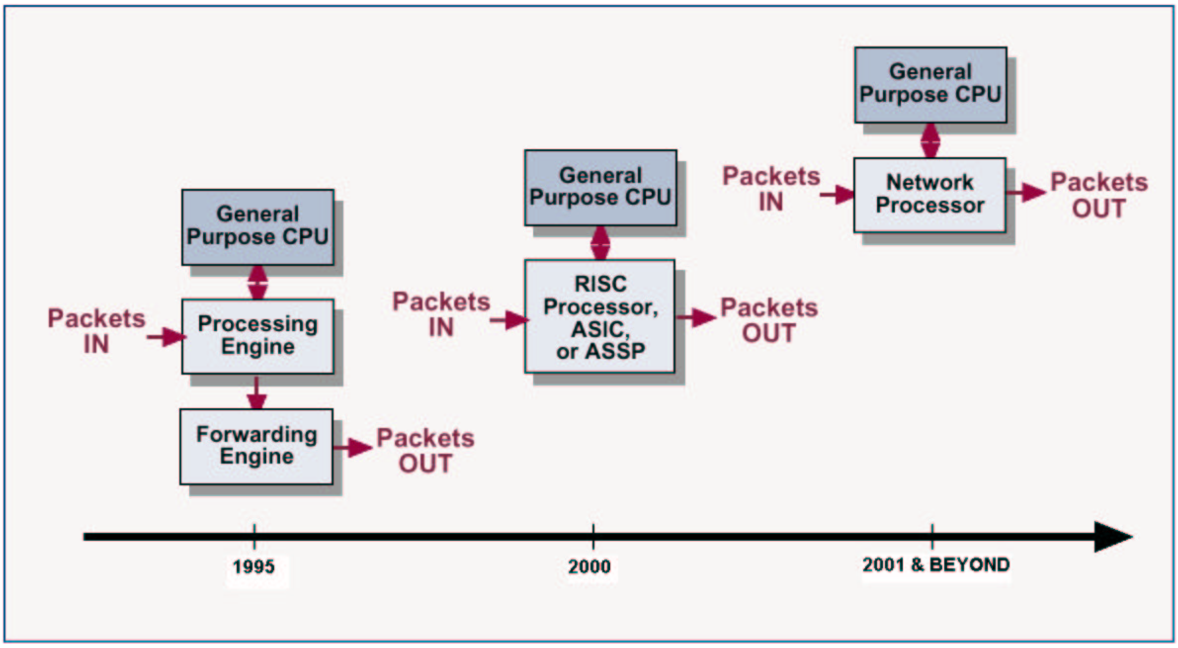


Figure 2b: Evolving the Network Design System Philosophy

This migration of system architecture has also shifted Network Processing functionality from hardwired solutions to programmable solutions (figure 2c). Note the change from fixed to programmable media, as well as the change from ASIC/ASSP technology to a single-chip (Network Processor) solution.

The target of every network processor is to perform packet processing with the flexibility of a microprocessor, but with the performance of a dedicated ASIC. There are two issues that concern network processors' performance. The first one is packet (data) processing and alternative solutions that improve data processing performance [16]. Network processing requires transfer of packets and additional information to and from large memories with large access latency. Therefore, memory latency and the alternative ways [16] to "hide" it, is the second issue.

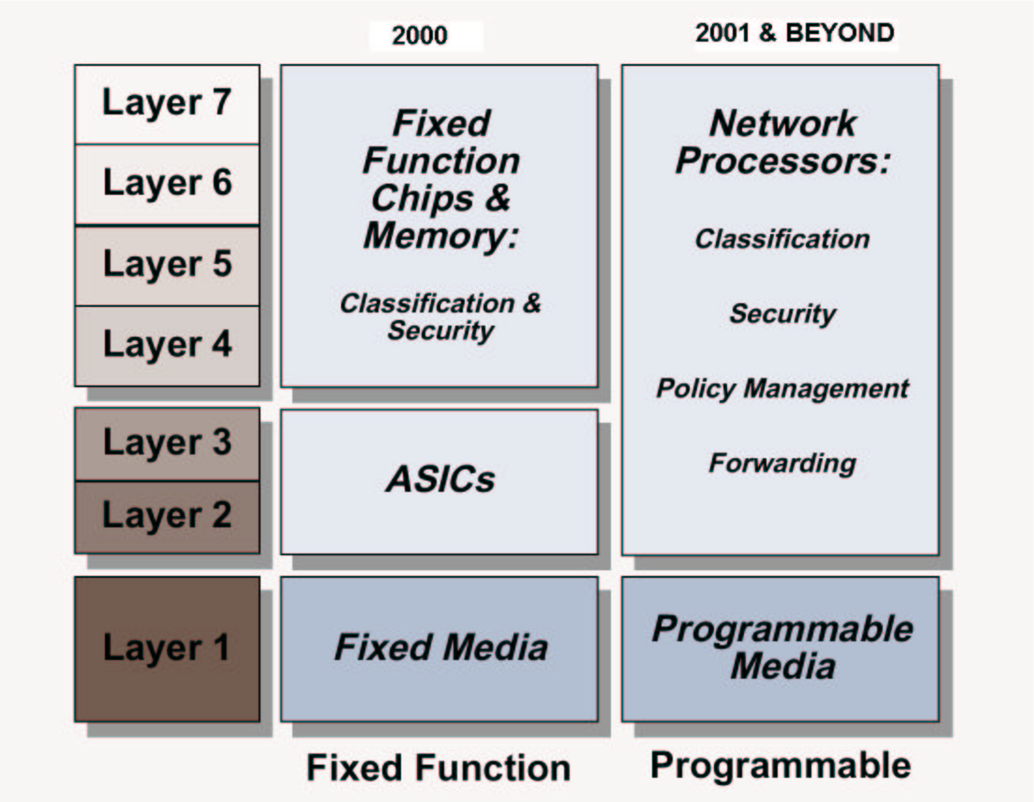


Figure 2c: Graphical comparison between network processor-based systems and their hardwired counterparts.

2.1 Efficient Data Processing

Regardless of specific function, most devises that fall under the category of "network processors" are based on either a multi-processor (highly parallel) or a multi-stage (highly pipelined architecture. In both cases, some type of hardware functional unit (state machine, programmable microprocessor, etc.) is combined with specialized software to support packet-oriented functions. Figures 2.1a and 2.1b graphically depict simple network processor architectures based on pipelined and parallel/multiprocessor approaches respectively. Parallel processing, pipeline and specialized hardwired modules can be used to achieve protocol processing at wire rates.

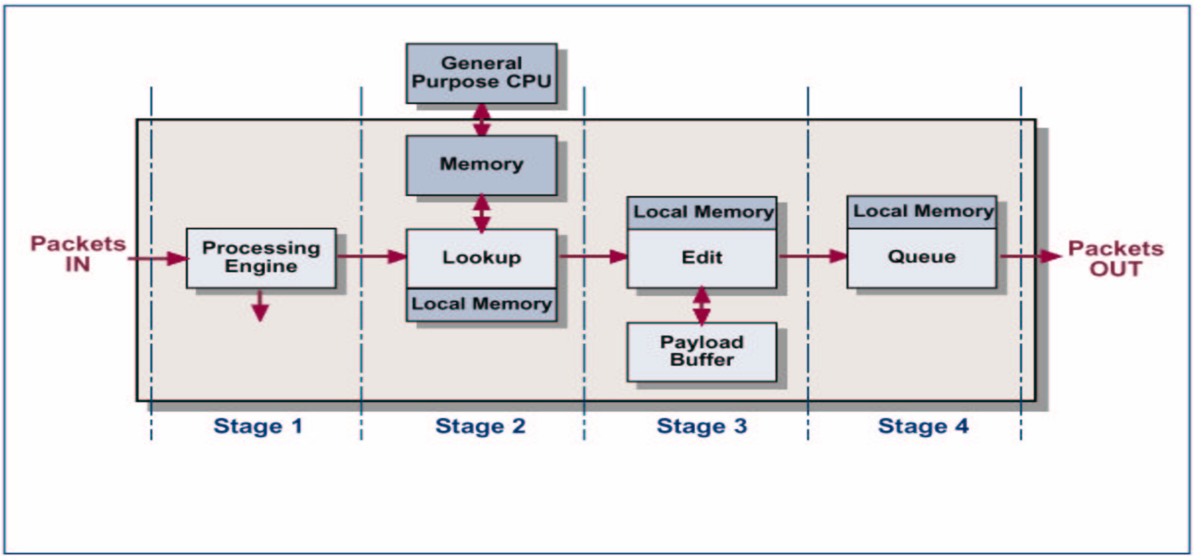


Figure 2.1a: A Simple network Processing Engine Based on a Pipelined architecture.

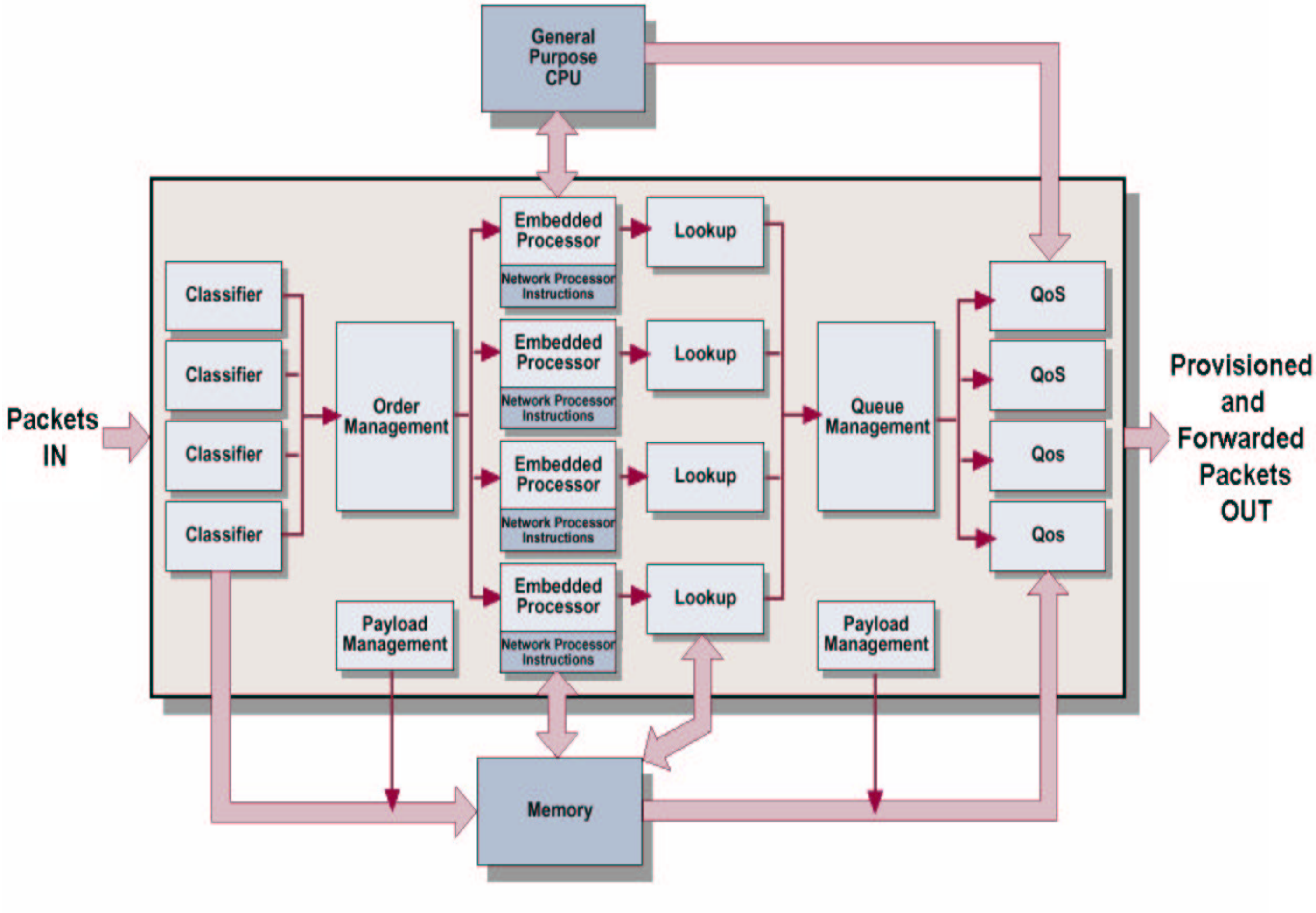


Figure 2.1b: A Simple Network Processing Engine Based on a Parallel/Multiprocessor Architecture.

2.1.1 Parallel Processing

Generally, network processors use parallel processing units (figure 2.1.1a) to improve performance and perform processing at wire rates. Especially for some protocols that each packet can be transported independently (such as IP protocol) is simple to use parallelism. Different packets can be processed in parallel in different processing units (Multi Instruction Multi Data -MIMD- processing). The most common approach is to use many RISC processors in a network processor interconnected to shared memory through a common bus. Data can be demultiplexed into streams and scheduled on the parallel processors through an I/O bus, too. However, flow oriented packet processing causes inter-packet dependencies. If packets of the same flow end up in different processing units inter-processor communication is required in MIMD architectures – this increases the complexity of network processors and reduces the performance of the parallel processing units. This inter-packet dependency can be avoided by scheduling packets of the same flow to end up in the same processing unit.

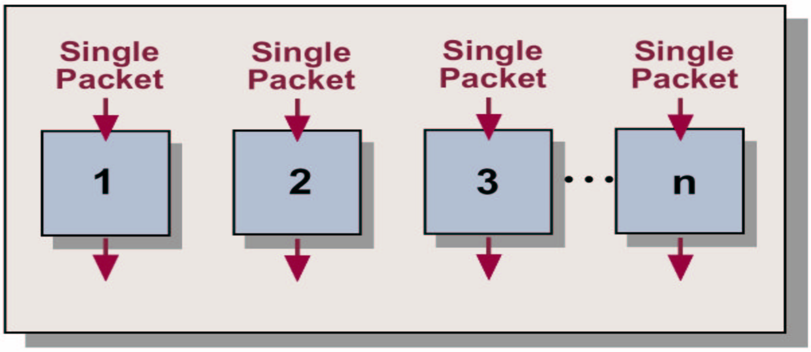


Figure 2.1.1a: Parallel Processing

2.1.2 Pipelining

Pipeline is another way to exploit parallelism (figure 2.1.2a). It breaks packet processing into steps, each step is processed in one pipeline stage. Assuming that pipeline architecture has *n* pipeline stages, at any time *n* packets are processed concurrently. Packets are processed simultaneously in these stages, but only one result can be completed in each pipeline cycle. This technique can be combined with parallel processing (Pro3 approach § 3.2).

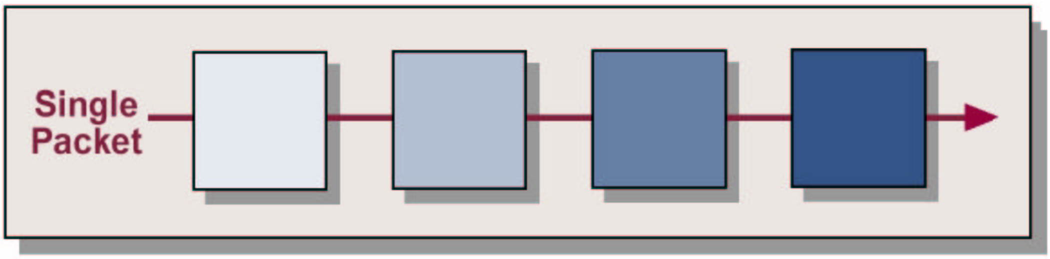


Figure 2.1.2a: Pipelining

2.1.3 Specialized Instruction Sets or Units

Analysis and modification of packets involves the manipulation of arbitrary number of bits. General-purpose processor instruction sets such as RISC are optimized for computation on a fixed word size. This leads to several shifting and logical operations to analyze and edit packet data fields. A common operation of this type is to compare a packet field with a value during packet classification. This may be used to identify a protocol. Since data comparison in RISC processors is performed on word boundaries and the bits in the packet field are not aligned, several shifts and logical operations are required to perform the comparison. A solution to this problem is to add special instructions to conventional instruction sets, such as: insert, extract field instructions, to make these operations more efficient. Another solution is to perform operations such as packet classification, which are common for all packets, in specialized hardwired pre-processing or post-processing units (either programmable or not).

2.2 Dealing with Memory Latency

Data transfer form and to large memories and various other memory operations such as table lookup, queuing and instruction fetches are required during packet processing. Hardware mechanism are required to “hide” these latencies, otherwise processing units will remain idle, for a great deal of time, waiting for data.

2.2.1 Multithreading

Multithreading is a technique that can “hide” these latencies by switching between multiple independent threads. Threads may be different programs or independent segments (such as procedures) of the same program. Since packet processing exhibits a high degree of parallelism, it is possible to identify independent threads. In a multithreading processor, each thread has its own context. A context includes all state unique to a given thread, such as registers, stack pointers, program counters, etc. Normally a memory access by a processor would block it until the data was returned. A multithreading processor can switch between contexts rapidly and the thread that performed the blocking operation can immediately pass control to another thread that is ready for execution. Having a large enough number of threads could completely hide memory latency and keep the processors’ resources fully utilized.

2.2.2 Memory Managers and Types of RAM

Memory manager units can perform complex memory operations that would block a processor for many cycles if it had to perform them. These individual units can manage memories (memory initialization, allocation, read, write) and perform operations like queuing, table-lookup and tree searches. Modules that use memories need only to send their request to memory manager and request will be performed autonomously.

The selection off on-chip and off-chip memories is also very important. SRAMs have better performance, can maintain its data as long as power is provided thus do not need refresh, are small and expensive. On the other hand, DRAMs have worse performance, they need time for refresh, but they provide the lowest cost per bit and greatest density among solid-state memory technologies. The traditional asynchronous DRAM underwent some limited changes. Examples were fast-page-mode (FPM), extended-data-out (EDO) and burst-EDO (BEDO), each provided faster cycle times if accesses were from the same row and thus more bandwidth than the predecessor. Conventional, FPM and EDO DRAM are controlled asynchronously by the memory controller; the memory latency is thus some fractional number of c.c.. An alternative is to make the DRAM interface synchronous such that the DRAM latches information to and from the controller based on a clock signal. SDRAM devices typically have a programmable register that holds a bytes-per-request value. SDRAM may therefore return many bytes over several cycles per request. The advantages include the elimination of the timing strobes and the availability of data from DRAM each clock cycles. RAMbus DRAM (RDRAM) tries to get rid of the latency by actually narrowing the bus path and treating the memory bus as a separate communication channel. Generally speaking, DRAMs use parallelism between their banks to serve many applications and utilize memory's bandwidth. DRAM operations are scheduled, possibly completing memory references out of order, to optimize memory system performance.

2.3 Examples of Network Processors

Seeing the unique needs of network processors, many companies such Intel, MMC, Motorola and IBM have introduced new designs for this fast paced market.

2.3.1 Intel's IXP 1200

Intel's IXP 1200 [7, 8, 9] (figure 2.3.1a) architecture consists of 6 micro-engines sharing a bus with memory. The micro-engines are managed by a StrongARM core processor. It has PCI bus to communicate with the host CPU, memory controllers and a bus interface to network MAC devices. The device operates at 166 MHz. Each micro-engine supports 4 threads, which helps to eliminate micro-engines waiting for memory resources. Micro-engines have a large register set, consisting of 128 general-purpose registers, along with 128 transfer registers. Multiple IXP 1200's can be aggregated in serial or parallel. IXP 1200 uses parallel context to hide memory latency. [7] evaluates IXP 1200 performance, implemented a prototype IP router on this processor, and demonstrated that it can sustain line speeds for 8 X 100 Mbps Ethernet ports. Moreover emulating infinitely fast network ports [7] shows that the IXP 1200 is capable of forwarding minimum-sized Ethernet packets at a rate of 2.69 Mbps. This experiments indicated that DRAM (of evaluation system) is the bottleneck and there is processing capability available to get at least 26% improvement with faster memory.

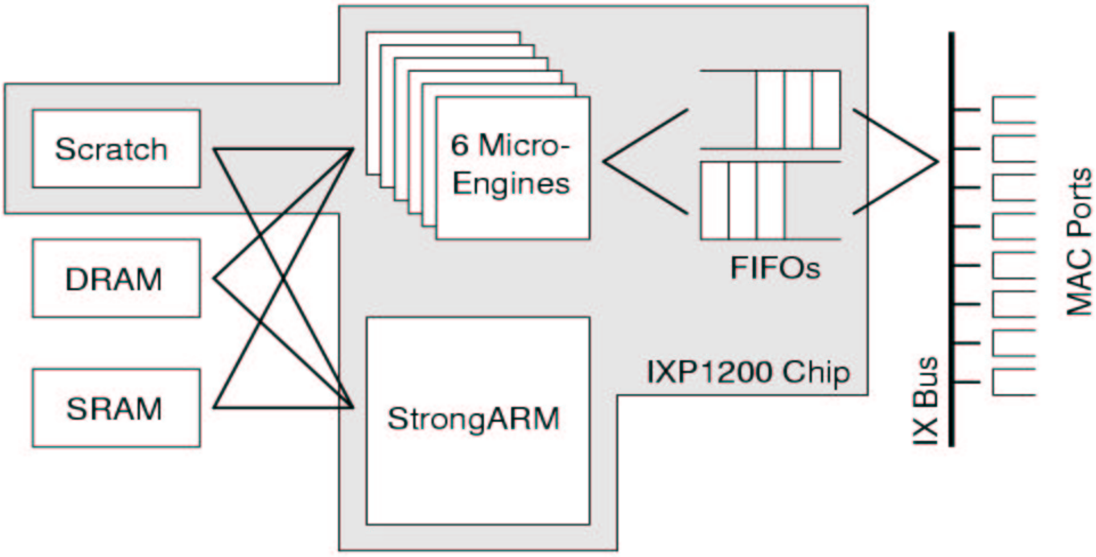


Figure 2.3.1a: Block Diagram of an Intel IXP 1200 Evaluation System

The following tables show IXP family specifications, from Intel's web site:

Network Processor	IXP1200	IXP1240	IXP1250	IXP1250 extended temp
Core speeds	166, 200, 232 MHz	166, 200, 232 MHz	166, 200, 232 MHz	166 MHz
Operating temperature	0° to 70°C	0° to 70°C	0° to 70°C	-40° to 85°C
Package	432-pin HL-BGA	432-pin HL-BGA	520-pin ESBGA	520-pin ESBGA

Table 2.3.1b: Intel IXP Network Processor Family Comparison

Microengine Core Speed	166 MHz	200 MHz	232 MHz	166 MHz extended temp
IX Bus speed	66 MHz	85 MHz	104 MHz	66 MHz
Micro engine control store	2K	2K	2K	2K
Peak IX Bus bandwidth	4 Gbps	5 Gbps	6.26 Gbps	4 Gbps
Internal power supply	Vdd=2V+/-5%	Vdd=2V+/-5%	Vdd=2V+/-5%	Vdd=2V+/-5%
External power supply	3.3+/-10%	3.3+/-10%	3.3+/-10%	3.3+/-5%
Power dissipation	3.80 watts	4.48 watts	5.19 watts	3.80 watts
Memory interface speed	83 MHz	100 MHz	116 MHz	83 MHz

Table 2.3.1c:Intel IXP 1200 Bus Speed and Bandwidth Comparison

2.3.2 MMC's Anyflow 5000, nP3400, nP7000

MMC [15] developed the AnyFlow 5000 (nPX5000) network processors. These have five different stages: ingress processing, switching, queuing, scheduling and egress processing. Per-flow queuing is used which allows each flow to be queued independently. Other functions handled on a per-flow basis are queuing control and scheduling.

AnyFlow 5400 Product Features

- 5.5 Gbps bandwidth per module, scalable to 22Gbps with up to four 5400 modules
- Direct interface to MMC's BitStream Processors
- Up to 16 Fast Ethernet ports or 2 Gigabit Ethernet ports supported per 5400 device
- Shared packet memory of 3MBytes per module
- Supports unicast, multicast, broadcast and trunking operations
- MMC's output streams group packets going to the same destination and having the same properties, accelerating overall pack processing
- Eight Dynamic Discard Threshold groups provide full flexibility for programmable stream processing
- Support for 802.3x Flow Control generates XON/XOFF credit messages to Ethernet via BitStream Processors
- Scheduling support for 4 CoS per output port
- Full support for Hierarchical Weighted Fair Queuing through MMC's SP/WRR Scheduler with multi-criteria priority selection
- CPU interface provides synchronous 55MHz operation

MMC also developed the nP3400 [13], which integrates a programmable packet processor, switch fabric and multiple Ethernet interfaces on a single chip. It contains two programmable 200-MHz RISC processors and a 4.4 Gb/s switch fabric. It has policy engines supporting 128 rules.

AMCC's purchase of MMC Networks developed nP7250 [13]. This is a straightforward 2.4 Gb/s network processor. It contains two MMC nPcore multithreaded packet processors and connects to an external search coprocessor and a host CPU that handle lower- and higher – level functions, respectively. AMCC transformed the nP7250 into the nP7510 [13] (figure 2.3.2a), the current top of AMCC's line, for 10 Gb/s rates by adding four more nPcore processing units and a chip-to-chip interface that allows multiple nP7510 chips to work together. nP7510 has six 333 MHz nPcores.

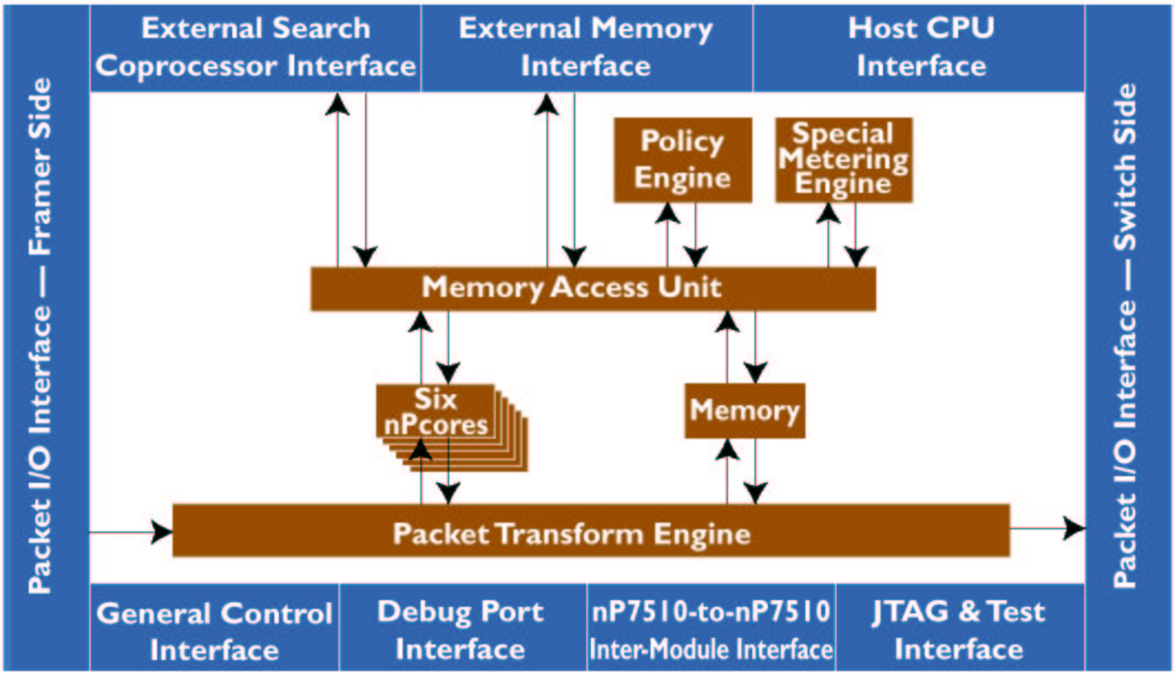

External Search Coprocessor Interface
External Memory Interface
Host CPU Interface
Policy Engine
Special Metering Engine
Memory Access Unit
Six nPcores
Memory
Packet Transform Engine
Packet I/O Interface — Framer Side
Packet I/O Interface — Switch Side
General Control Interface
Debug Port Interface
nP7510-to-nP7510 Inter-Module Interface
JTAG & Test Interface

Figure 2.3.2a: nP7510: OC 192c Network Processor

2.3.3 IBM's NP4GS3 "Rainier"

IBM developed the NP4GS3 "Rainier" NPU [14], which has good reviews [20] (figure 2.3.3.a). It has 16 programmable protocol processors and a PowerPC 405 control processor. It has hardware accelerators to perform tree searches, frame forwarding, filtering and alteration. Each processor is a 32-bit, has a 3-stage pipeline (fetch, decode, execute) and runs at 133 MHz. Each processor has seven coprocessors associated with it, including one for checksum, string copy and flow information. Hardware accelerators perform frame filtering, alteration and tree searches.

NP4GS3 function units include tree-search engines (TSE), one of which is shared with each pair of processors; two full-duplex switch interfaces; four 1 Gb/s media-access ports; and interfaces to 10 external memory arrays- eight DDR SDRAM ports and two ZBT SRAM ports.

The aggregate bandwidth of the NP4GS3 is some 4 Gb/s, allowing the part to manage a single OC-48 channel or up to forty 100Mb/s Ethernet ports. Two chips can be connected in series, using the switch interfaces and being controlled by the PPC405 core in one of the chips to double the bandwidth of the subsystem. This capability puts the NP4GS3 within striking distance of 10 Gb/s, but IBM is developing new parts that will exceed this rate in single-chip configurations [20].

For applications that need even faster data rates or more sophisticated processing, multiple NP4GS3s can be configured to communicate through an external switch fabric so that each chip performs only a portion of the packet-processing tasks. IBM designed the chip to support up to 64 devices in parallel under control of an external CPU.

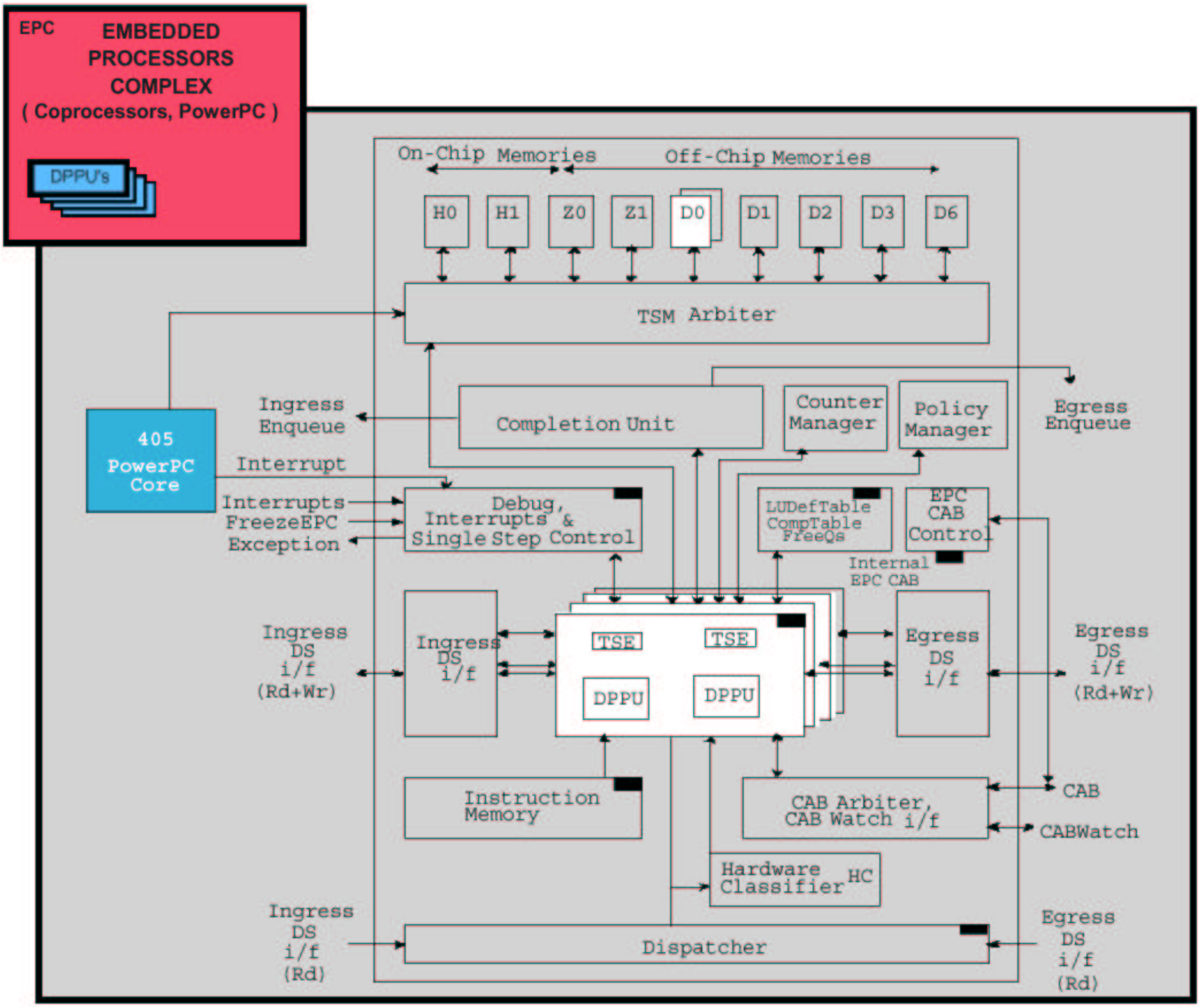


Figure 2.3.3a: IBM’s NP4GS3 “Rainier”EPC High-level Architecture. EPC Architecture, which shows how the EPC core is structured. Note that the Specialized PowerPC 405 engine called the ePPC (Embedded Power PC) resides here and can be used as the CP (Control processor) in systems containing up to 2 NPs.

2.3.4 Motorola/C-Port C-5

C-Port (acquired by Motorola) developed C-5 network processor [10] (figure 2.3.4a). It includes sixteen 200 MHz channel processors, comparable to IBM’s processors [20], along with five different coprocessors for control, switch-fabric interfacing, table lookups, queue management and buffer management. In addition, like the IBM’s NP4GS3, the C-5 supports multichip arrangements to support bandwidth faster than the theoretical peak 5 Gb/s capacity of a single device. The C-5 channel processors are about 50% faster than IBM's NP4GS3 [20], but they lack the multithreading support in the IBM cores. The C-5 also falls slightly short in local memory bandwidth. It supports only a single 128-bit array of external single-data-rate (SDR) SDRAM operating at up to 125 MHz. A maximum of 128 Mbyte of SDRAM can be connected to the chip. Finally, one external 64-bit, 133 MHz ZBT SRAM array, up to 32 Mbyte in size, is also supported to store table data for statistics, routing and quality-of-service (QoS) functions. IBM's offering provides more independent memory arrays that are narrower and faster and that provide more aggregate bandwidth and less risk of contention [20].

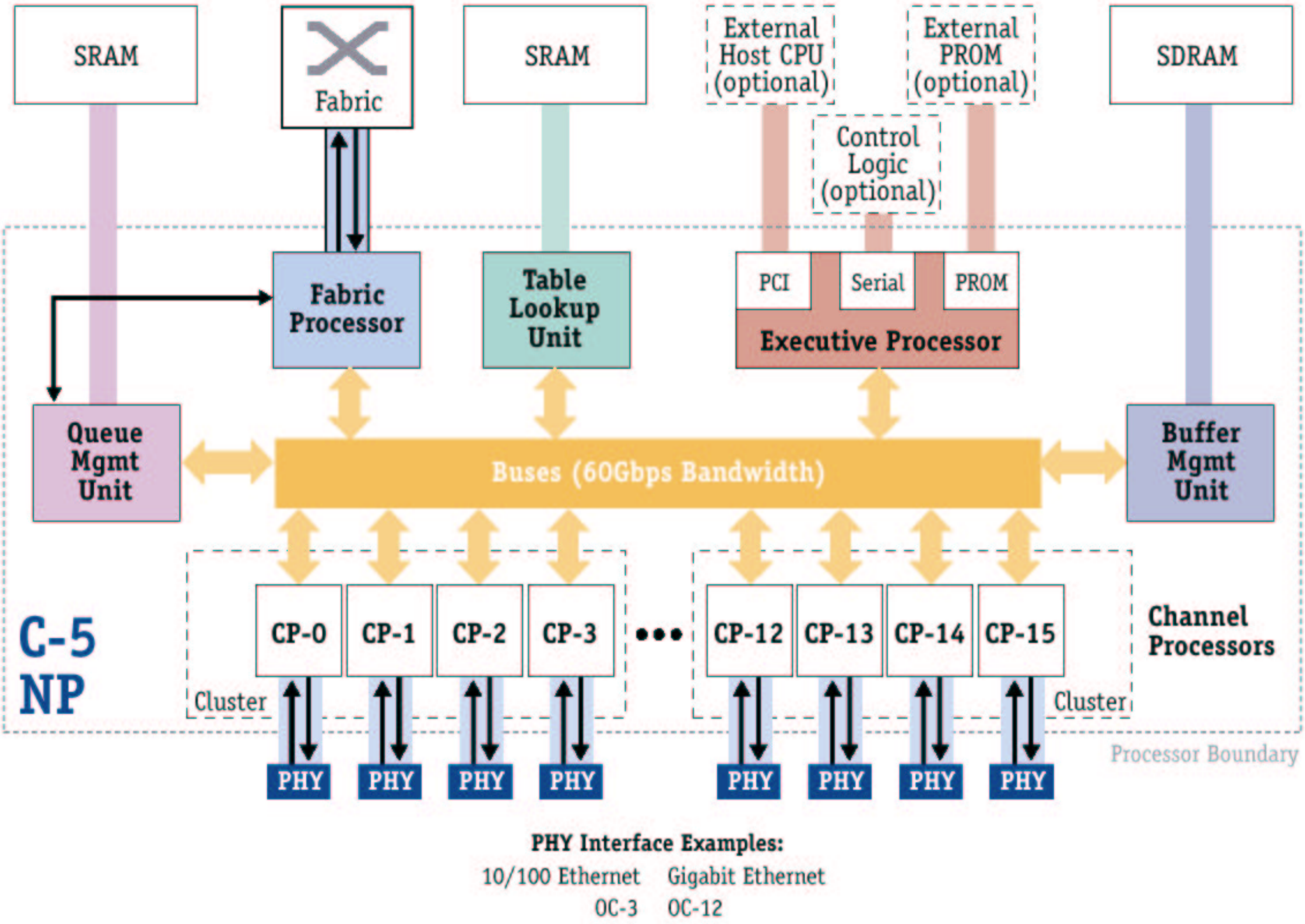


Figure 2.3.4a: C-5 Network Processor Block Diagram.

Chapter 3: Protocol Processor Project Architecture

3.1 Protocol Processor Project (Pro^3)

Pro3 system Architecture [1, 2, 3] aims in accelerating execution of telecom protocols by extending a high-performance RISC core with programmable, pipelined hardware. Processing intensive and (hard) real-time protocol functions are handled by the programmable hardware, while the remaining functions will be handled by the on-chip RISC in an integrated way. The concept of PRO3 is to provide the required processing power through a novel architecture incorporating parallelism and pipelining wherever possible, integrating generic micro-programmed engines with hardwired components optimized for specific protocol processing tasks. Analysis of protocol performance [2], [3] within such systems shows that a small specific subset of the protocol functions (i.e. less than 10%) is active during large period of time (i.e. more than 95% of the entire time span). The other protocol functionality is active when errors occur or for set-up and tear down of a protocol session. PRO3 targets at realizing at wire-speed the aforementioned specific set of the 10% of the protocol functions in the reconfigurable module and the remaining functions in the RISC core. The system is designed to support up to 2.5 Gb/s links for up to 500K flows supporting in worst-case 7.5Mpackets/sec.

Pro3 processor [5] (figure 3.1a) integrates a state-of-the-art RISC [6], with reconfigurable, pipelined module able to deliver the needed processing power to support efficiently many thousands of (different) protocol instances, hereafter mentioned also as flows. Two such modules constitute the Re-configurable Pipelined Module (RPM figure 3.1a) of the PRO3 system, which operate in parallel, facilitating the execution of protocols with different incoming and outgoing data flow processing as well as load balancing for higher throughput.

In general, within a stateful firewall system, the following sequence of functions is applied to each incoming packet: **reception, classification, state processing, and transmission** (Figure 3.1b). Each generic function consists of a set of lower level functions and can be understood as pipeline stages. Other main blocks include **data/queue manager** and higher layer protocol processing performed in SW (**Hyperstone or the external CPU).** The common path for stateful inspection up to the transport layer will be performed in the **PRO^3 hardware pipeline**, and higher layer applications on the **Hyperstone internal CPU**.

Within the ATM switch system for control plane processing the above basic blocks will also exist, re-using common HW structures or dedicated ones. Part of the higher layer processing is performed in SW running on the Hyperstone CPU. Communication with the host system (External CPU), which executes the management plane, is required.

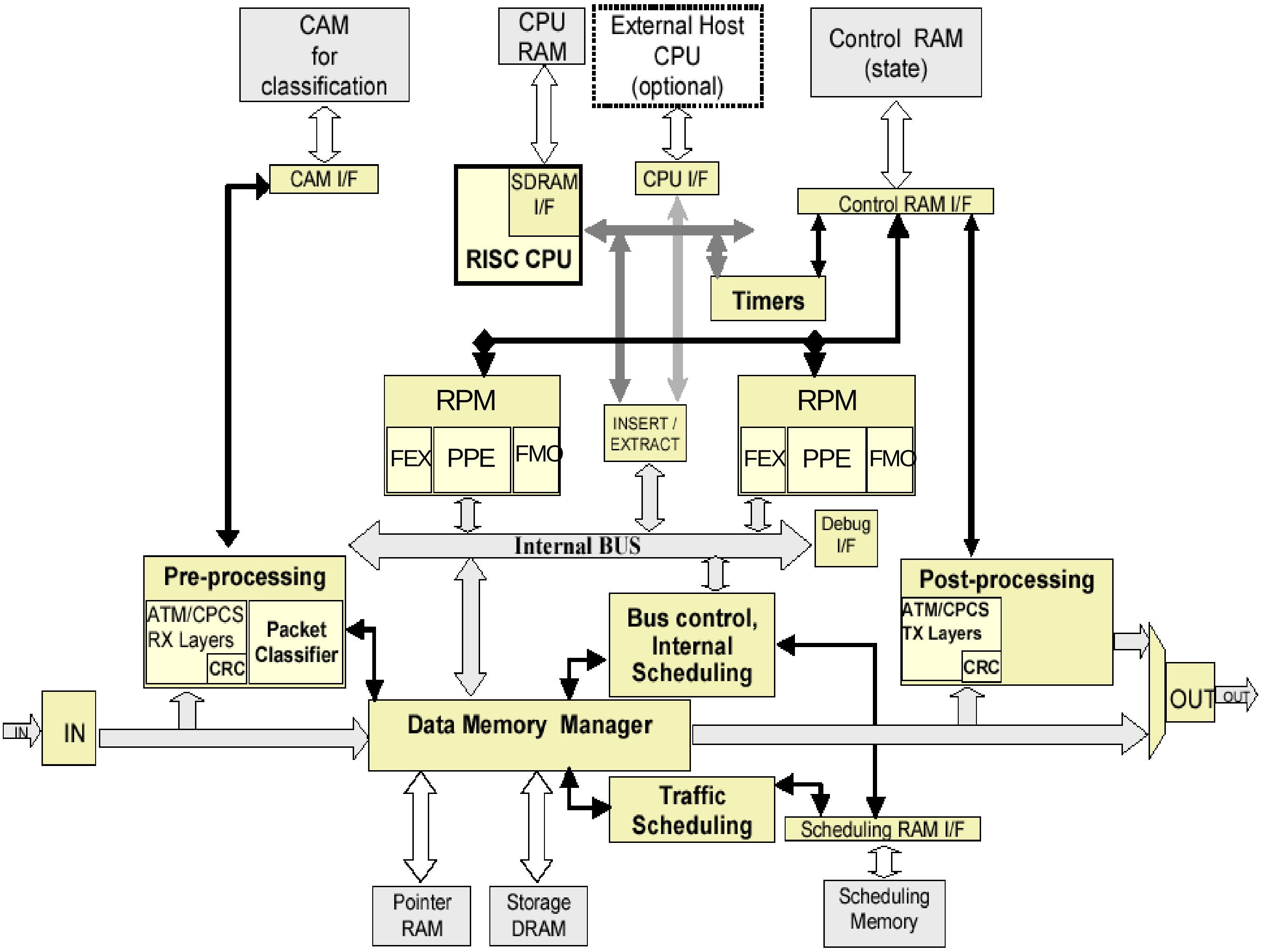


Figure 3.1a: Pro^3 Architecture

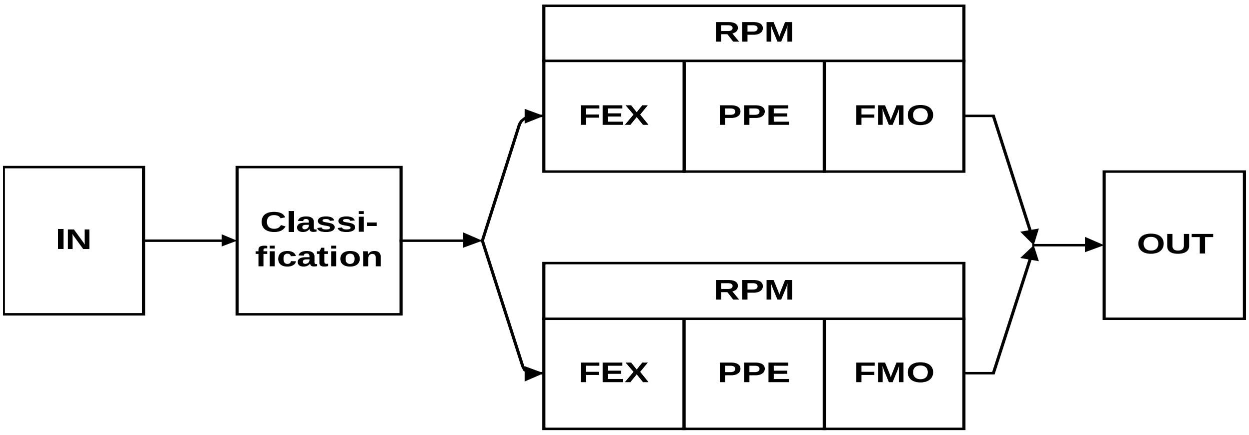


Figure 3.1b: Pro3 pipeline stages (Reception, Classification, State Processing and Transmition)

3.2 Functional architecture

Figure 3.2a summarises the PRO^3 functional architecture.

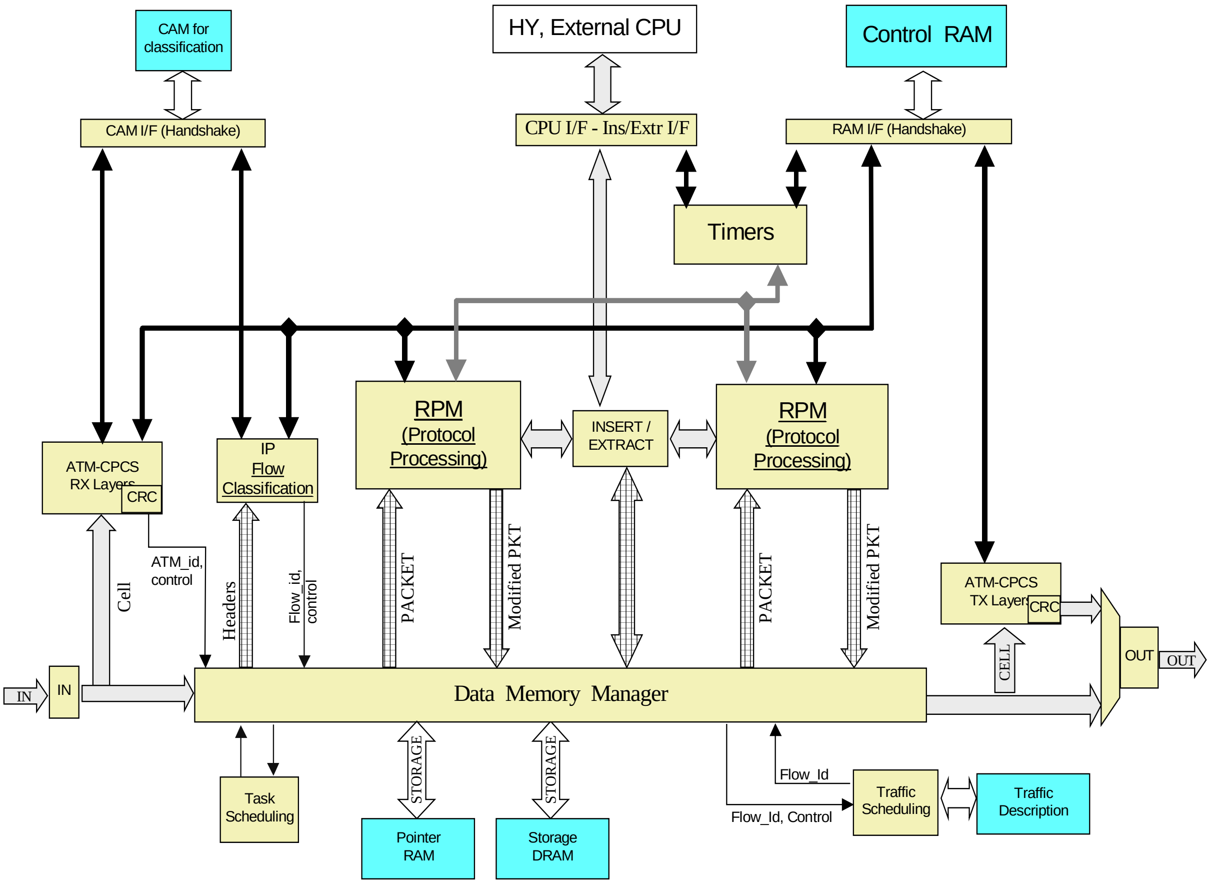


Figure 3.2a: PRO^3 functional architecture

As depicted in Figure 3.2a, the networking data are stored temporarily on the data storage DRAM and then forwarded either to one of the two RPMs or to one of the two processors (the internal Hyperstone CPU or the external CPU). The data paths and sequence of operations are described in the following section.

A simple example of a firewall system is described next. Figure 3.2b shows five PCs that are protected by a firewall. Incoming packets come from the outside world through a router to the firewall and they either pass or are rejected according to firewall's policy. Similarly outgoing packets are been rejected or pass through the firewall. Outgoing packets do not show their real source IP address, they seem to have source IP addresses other than their real ones. This function of firewall is called *masquerade*. Firewall's policy allows specific protocols to go out from an IP address and specific protocols to end up in an IP address. For example firewall can decide that nobody outside the firewall can use telnet to access IP1, or that IP3 cannot send emails and can only use http protocol. Every protocol corresponds to a connection port. These rules and the IP address of every PC determine the result of every protocol process's execution.

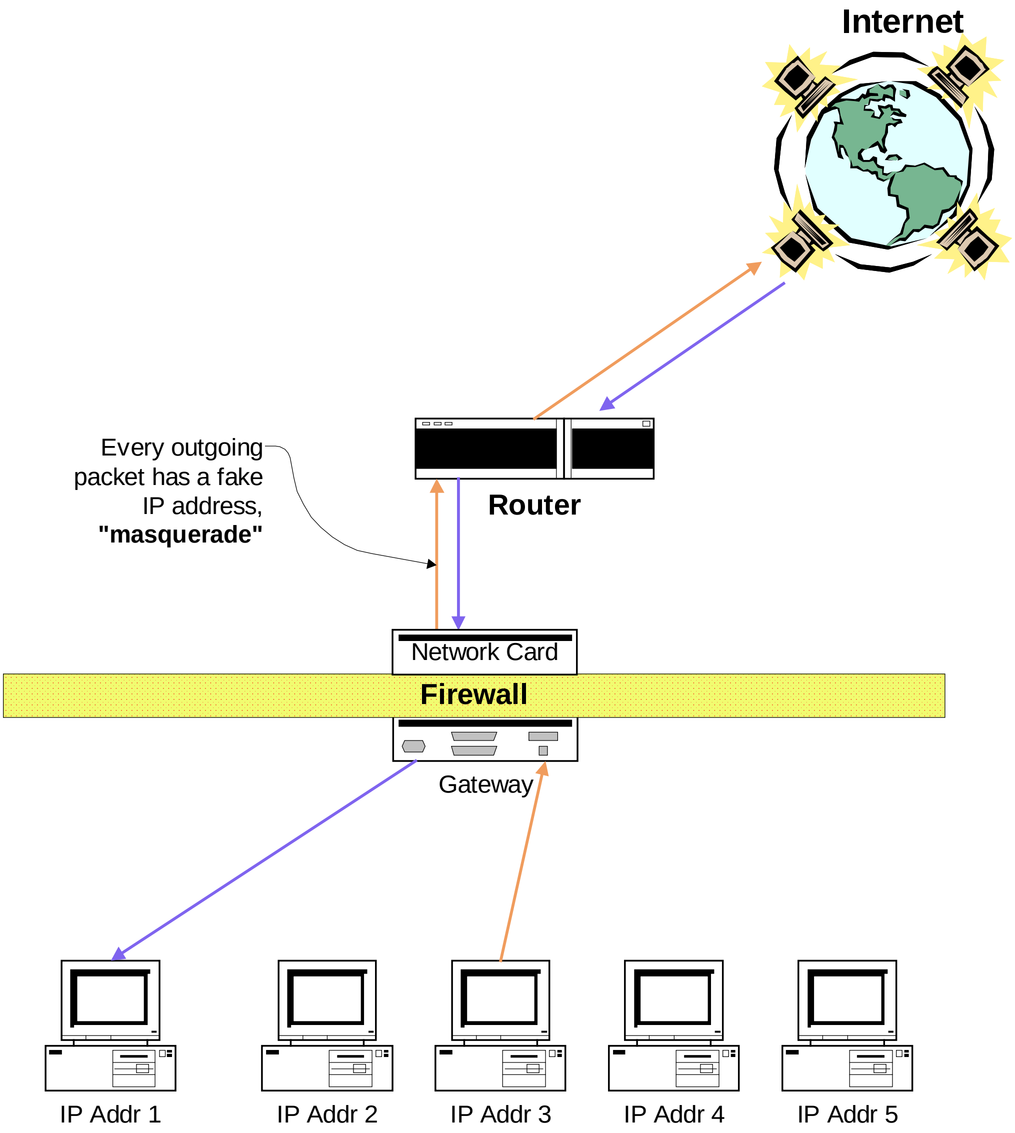


Figure 3.2b: Firewall for IP packets

3.3 Physical architecture

Figure 3.3a depicts the PRO^3 physical architecture. A debug block is added that will be used for chip verification and application firmware debugging. Furthermore, the insert/extract block is elaborated and the interconnection of the internal Hyperstone CPU and the external CPU is presented.

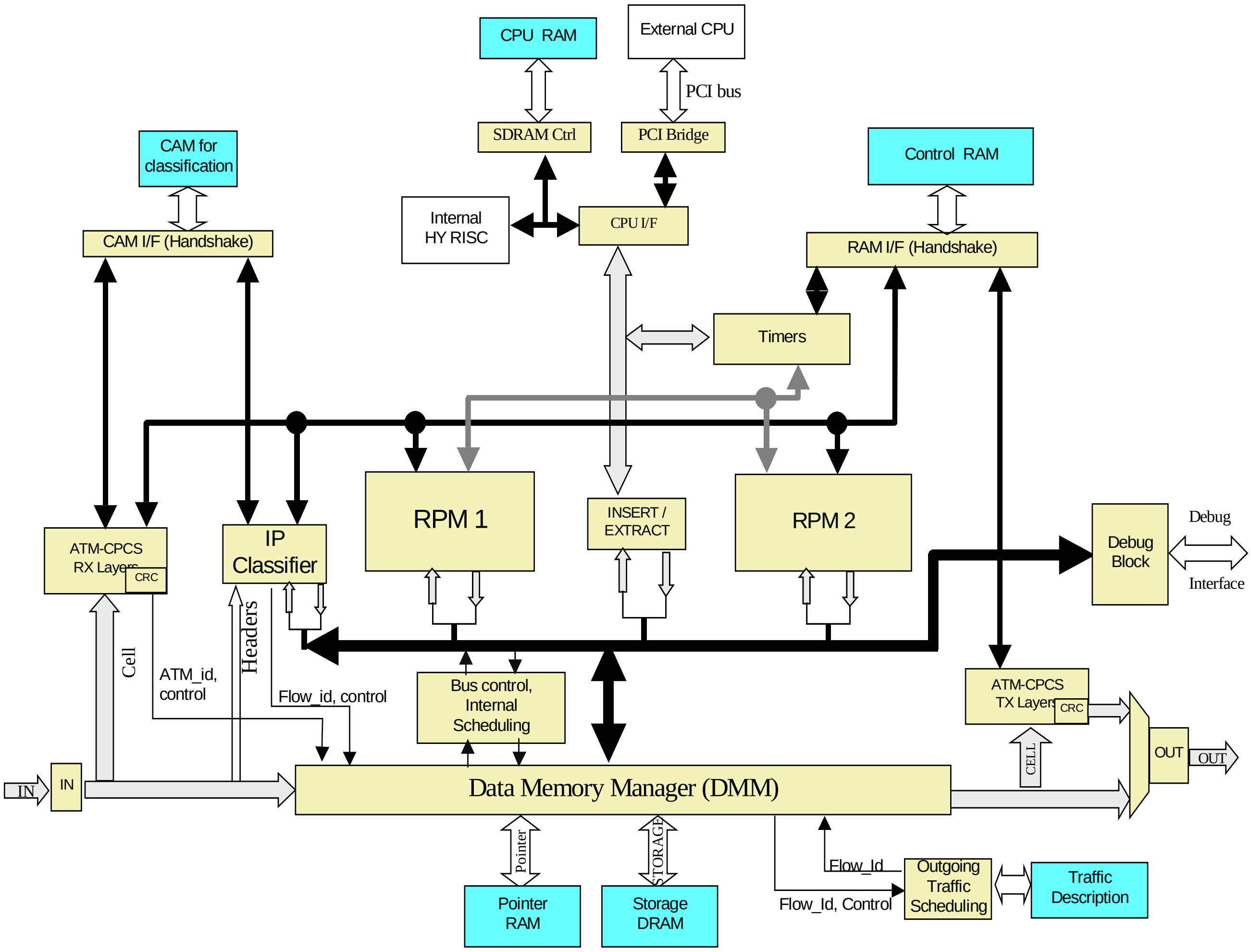


Figure 3.2.a: PRO^3 physical architecture

In Figure3.2a, a shared internal bus is shown that is used for data transfer between the RPMs and the Data Memory Manager (DMM). Apart from this point-to-point data transfer there is communication between the two RPMs or between them and one of the two CPUs.

In addition to this internal bus for on-chip communication, there is an additional microprocessor bus shared among all blocks for their configuration, control and firmware download. This bus is in fact the uP bus of the internal Hyperstone CPU.

Packet pre-processing and lower layer protocol functions are executed by means of hardwired functionality (like the full ATM/CPCS layers) as well as programmable PDU processing and packet classification by means of a RISC-like micro-engine for Field Extraction (FEX) and controller of a high throughput external Ternary CAM (Content Addressable Memory) device for flexible and deterministic classification.

Since the processor must be able to support thousands of active instances of protocol FSMs, special circuitry designed to provide a pool of timer resources for the realization of the (many) thousands of active watchdog timers. The timer information is stored in the flow state.

In the transmition flow the operation of PRO3 is based on packet buffering prior to protocol processing. A Data memory Management sub-system (DMM) is responsible for memory allocation and stream buffering. Data can be retrieved by DMM in response to specific commands and be delivered over the internal bus to the RPM modules or via the insert/extract interface to the control RISC or to host CPU.

The internal Scheduling Unit maintains a number of priority queues in order to schedule the forwarding of the packets for processing according to the priority of each flow. It is also used to multiplex the execution of data transactions from the DMM to the different internal destination via the Internal Bus. Data memory interface consists of a DDR SDRAM memory block. It has been selected that is able to offer bandwidth of up to 17 Gbps. The memory data bus is 64 bit wide running @ 133 MHz.

The shared internal bus has been defined following a worst-case approach. The bus will use a 64 bit wide data bus running @ 200 MHz. This internal bus will be bi-directional and shared among all blocks as shown in the above figure. Moreover, control and address signals will be shared. A point-to-point control interface will connect each component with the bus arbiter and task scheduler for chip control and operations synchronisation.

3.4 Internal data paths

The internal data paths and sequence of operations of both the IP Application and the ATM Application are now detailed.

3.4.1 IP Application

In this section we analyse the data paths of the IP application that is typical for monitor type of applications. Based on the functional architecture described above, a number of data paths may be traversed through the PRO^3 system, depending on the required type of processing.

Incoming packets are directly forwarded to the DMM for storage. The memory allocation is based on fixed size segments, thus a segmentation and reassembly function is built in the component. As the packet is sent to the DMM, it is also copied to the IP classifier for flow classification. The fields required for the IP flow classification of the packet are extracted, and fed to a CAM in order to take decisions regarding the packet. As soon as the packet is correctly classified, its Flow_ID is obtained and sent to the DMM in order to assign the temporarily stored packet to the proper queue. Finally the task scheduler is updated regarding the new packet.

Following a command by the task scheduler, the packet is forwarded to one of the protocol processing units or to one of the two CPUs either in its entirety or in parts. After processing is complete, updated data are returned to the DMM and the packet is scheduled for transmission.

When worst-case traffic conditions are to be serviced, pass through processing is required. This implies processing of minimum sized packets (i.e. 40 bytes), received and transmitted at a rate of 2.5 Gbps. By translating these figures into processing time we found that 128ns are available for the processing of each packet. On the other hand, when more elaborate processing is required on the packets, it is safely assumed that worst-case traffic conditions are not going to be observed under normal circumstances and, therefore, average traffic is considered.

In the next sections we analyse the operations existing in the IP-based protocols. Such protocols use either the IP over SONET interface or the IP over ATM interface.

3.4.1.1 IP over SONET interface

In the IP over SONET interface, the data flows and the sequence of operations are illustrated in Figure 3.4.1.1a. In this type of applications (process and forward) there are two alternative blocks for protocol processing, namely RPM1 and RPM2. In general there are three independent sequences of operations that perform in parallel and have the DMM as common reference. These are *packet reception*, *packet processing*, and *packet transmission*. Furthermore, there is *an additional operation and data path between the two CPUs* that is not controlled by the task scheduler.

The operations are initiated upon the appearance of specific events and are completed as soon as a packet is received by the destination or sent to the network. Thus the operations are initiated from the IN block (at reception of packet), from the task scheduler (send packet for processing, or send packet back to DMM after processing, or insert/extract packet) and from the traffic scheduler (send packet to the network). In other words, the IN block initiates and controls the transactions for the input interface (reception process), the task scheduler initiates and controls the transactions in the internal bus (packet processing), and the traffic scheduler initiates and controls the transactions in the outgoing network interface (transmission). In terms of bandwidth, these operations correspond to 2.5Gbps for reception and transmission processes each and 5 Gbps for packet processing (2.5 Gbps send for processing and 2.5 Gbps received after processing).

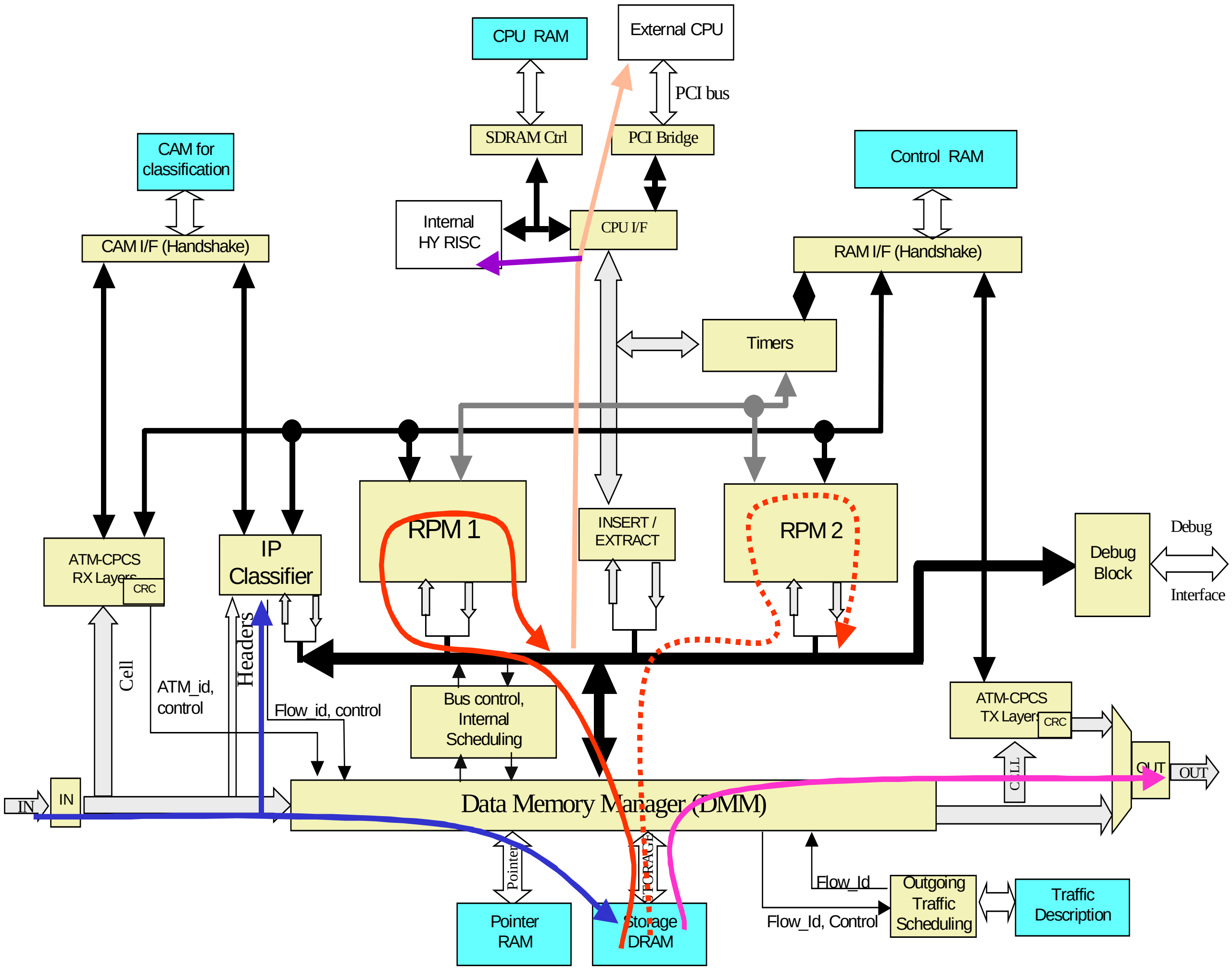


Figure 3.4.1.1a: Data flows for IP over SONET applications

A. Packet reception

IN module receives a packet and forwards it to the IP classifier and DMM. IP classifier makes preliminary check, and performs field extraction and classification. Obtains Flow_ID, Priority and Control info and sends them to the DMM. DMM segments received packet, and stores it in a temporary queue, calculates and stores packet length. It also sends to Schedulers scheduling info based on Priority, Flow_ID and Control info received.

If the packet is to be processed then update the task scheduler control info else if the packet is to be forwarded to the network without processing, then update the traffic scheduler control info.

B. Packet processing

The task scheduler initiates and controls the operation of the internal bus and sets up the inter-block communication. It receives various input signals and identifies both sender and destination blocks.

In the following the five possible senders to destination operations are identified and described. As any block, we consider one of the RPMs, the internal CPU, or the interface to the external CPU. These blocks may be used for packet processing. Packets are usually processed in RPMs, rarely in internal CPU and in special cases in external CPU.

1. DMM to any block:

   DMM sends a segment of a packet, and some additional information (classifier information, commands) through internal bus for processing.

2. Any block to DMM:

   This operation is performed when an insert (processed) packet and control command is sent from a CPU or an RPM to the DMM. In case of a processed packet DMM stores it and updates the control entry for the task scheduler. Whether the packet is ready to be transmitted to the network, it is appended to an output queue. Else if the packet is to continue processing to a higher protocol layer, then it is appended to the respective higher layer queue. It might be the case that DMM receives only a control command without data.

3. RPM1 to RPM2:

   In this case, RPM1 has to inform RPM2 to perform an action (send a packet). As soon as the task scheduler decides for this communication,

4. RPM to insert/extract:

   In this case, RPM has to send a packet to a CPU through insert/extract module. The task scheduler will enable the transaction.

5. Insert/extract to RPM:

   In this case, a CPU has to send a packet to an RPM through insert/extract module. Again the Task Scheduler will enable the transaction.

C. Packet transmission

The traffic Scheduler reads the status of the output traffic classes and selects the Flow_ID of the next packet to be transmitted, it sends a command to DMM (dedicate bus) for the next packet (Flow_ID, control) to be sent and updates its traffic queues. DMM decodes the command, reads the packet/cell to be sent and updates pointers and control.

D. CPU1 to CPU2 data path

An additional data path is foreseen between the two CPUs. This communication is resolved locally without the Task Scheduler coordination. The CPU I/F controls the operation and multiplexes it with the communication operations of the two CPUs with the insert/extract block (through it to DMM, RPM1, RPM2).

3.4.1.2 IP over ATM interface

In protocols based on the IP over ATM transfer mode, the data flows and sequence of operations are illustrated in figure 3.4.1.2a. The data flows are similar to the ones presented in the previous sections. The difference is that the protocol needs first ATM-based and then IP-based classification. The protocol processing is performed based on the Flow_ID obtained from the IP flow classification. In general in this mode the received cells are classified and queued into a number of ATM specific queues. As soon as a valid Common Part Convergence Sublayer (CPCS) packet is completed, the packet is queued for IP flow classification. Due to the non-sequential arrival of the ATM cells, it is not possible to keep information of message organization (list of cells that belong to a message) in a single queue (input queue). ATM cells are enqueued to the proper ATM flow. As soon as an ATM-CPCS message is completed, then it is forwarded for IP classification. After the IP classification, the encapsulated IP packet is appended to the appropriate IP flow. At this point all packets waiting for IP flow classification are appended to a single queue. In this mode of operation as well as in an ATM cell based interface in general, the DMM stores cell based streams. Here, the DMM must be able to extract only the cell payloads and send it for protocol processing.

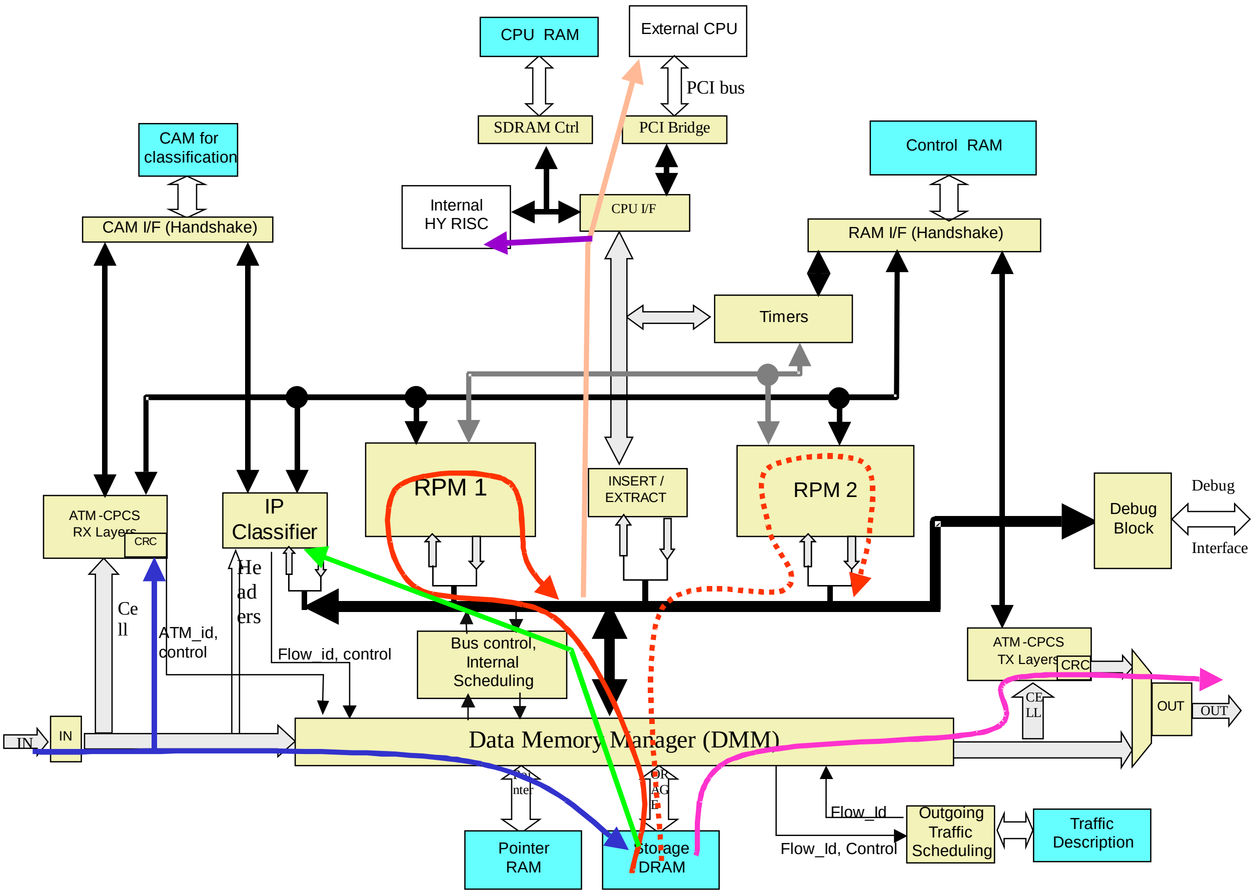


Figure 3.4.1.2a: Data flows for IP over ATM applications

3.4.2 ATM application

In this section we analyse the data paths of the PRO^3 ATM application that is typical for terminal type of applications where protocol (or stack) termination and initiation is performed. Incoming cells are directly forwarded to the DMM for storage and copied to the ATM/CPCS receive block for classification. The cells are waiting in a FIFO for the result of the classification (Flow_ID, Control, Priority) and are then appended to the right ATM queue. Since the ATM cells, which belong to an ATM message, don’t arrive sequentially, they have to wait in an on-chip buffer until a Flow_ID as well as the cell type (last or intermediate cell of a message) is defined.

In general all data flows and operations involved in the ATM application are identical with the operations described in the IP over SONET applications. The difference is that the ATM/CPCS components are involved and that the DMM stores queues of cells.

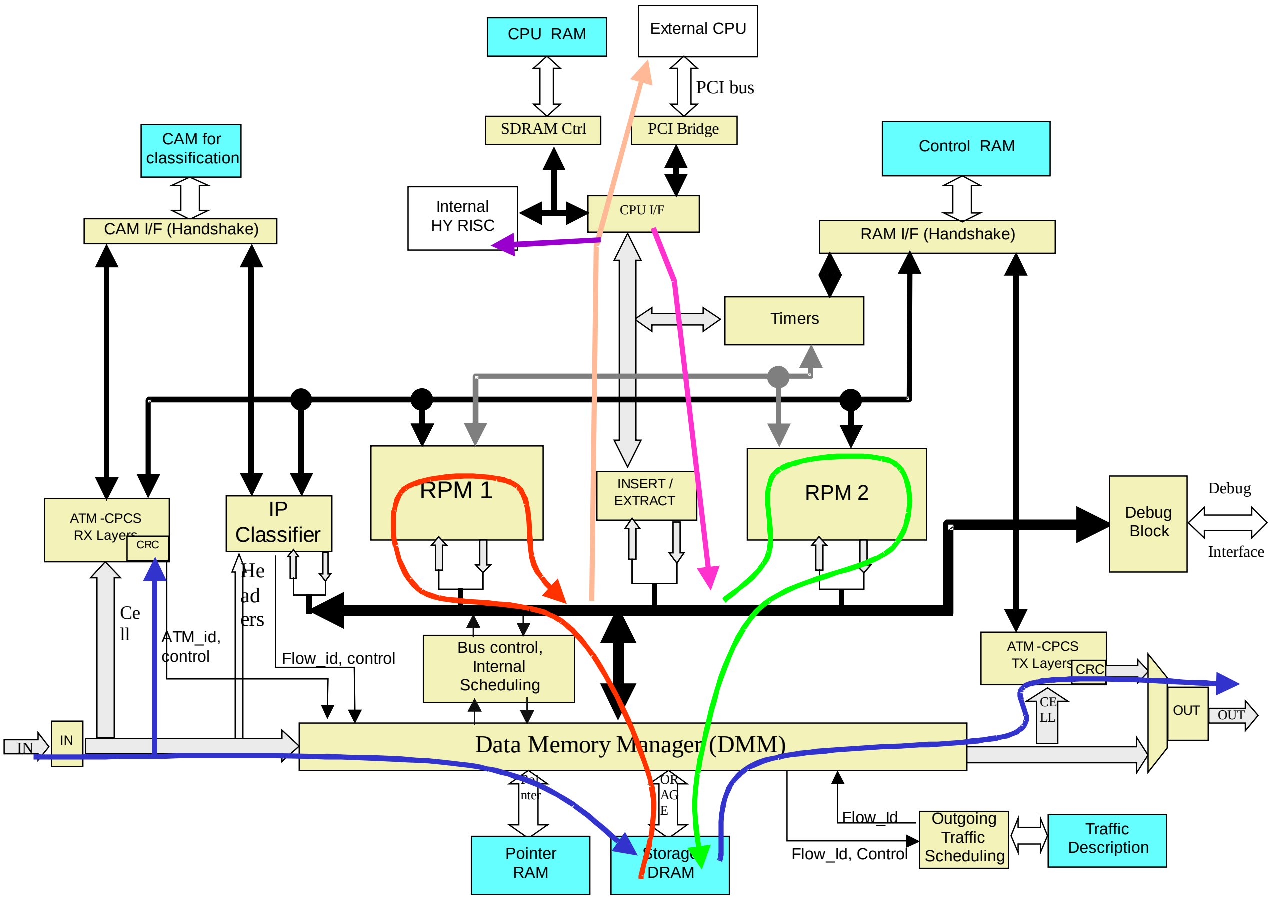


Figure 3.4.1.3a: Data flows for ATM applications

3.5 Internal interfaces

Having described the data paths traversed within the PRO^3 system, it is necessary to briefly refer to the block interfaces. Interconnections between the various blocks, together with an indication on the required bandwidth in each interconnection segment are shown in Figure 3.5a.

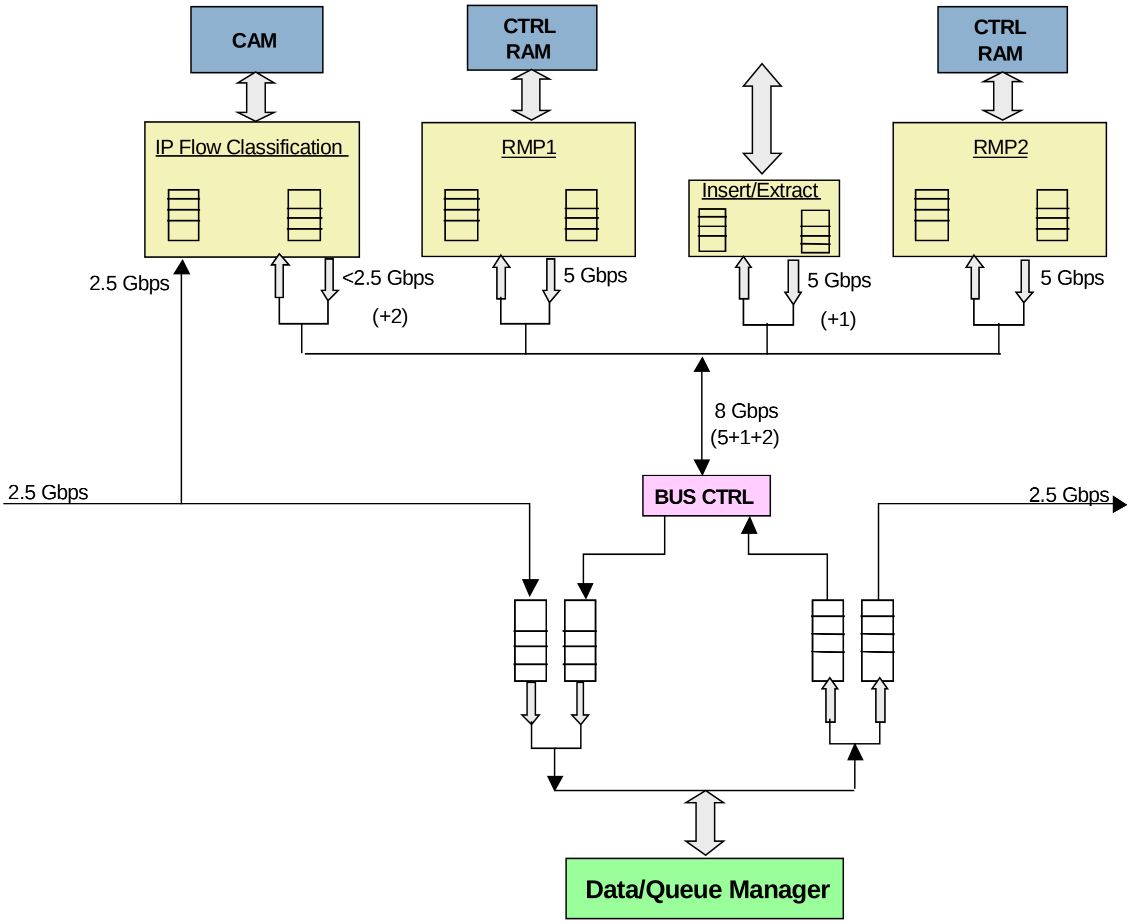


Figure 3.5a: PRO^3 internal interfaces

3.6 Re-configurable Pipelined Module (RPM)

The Re-configurable Pipelined Module (RPM) components are highlighted in figure 3.6a

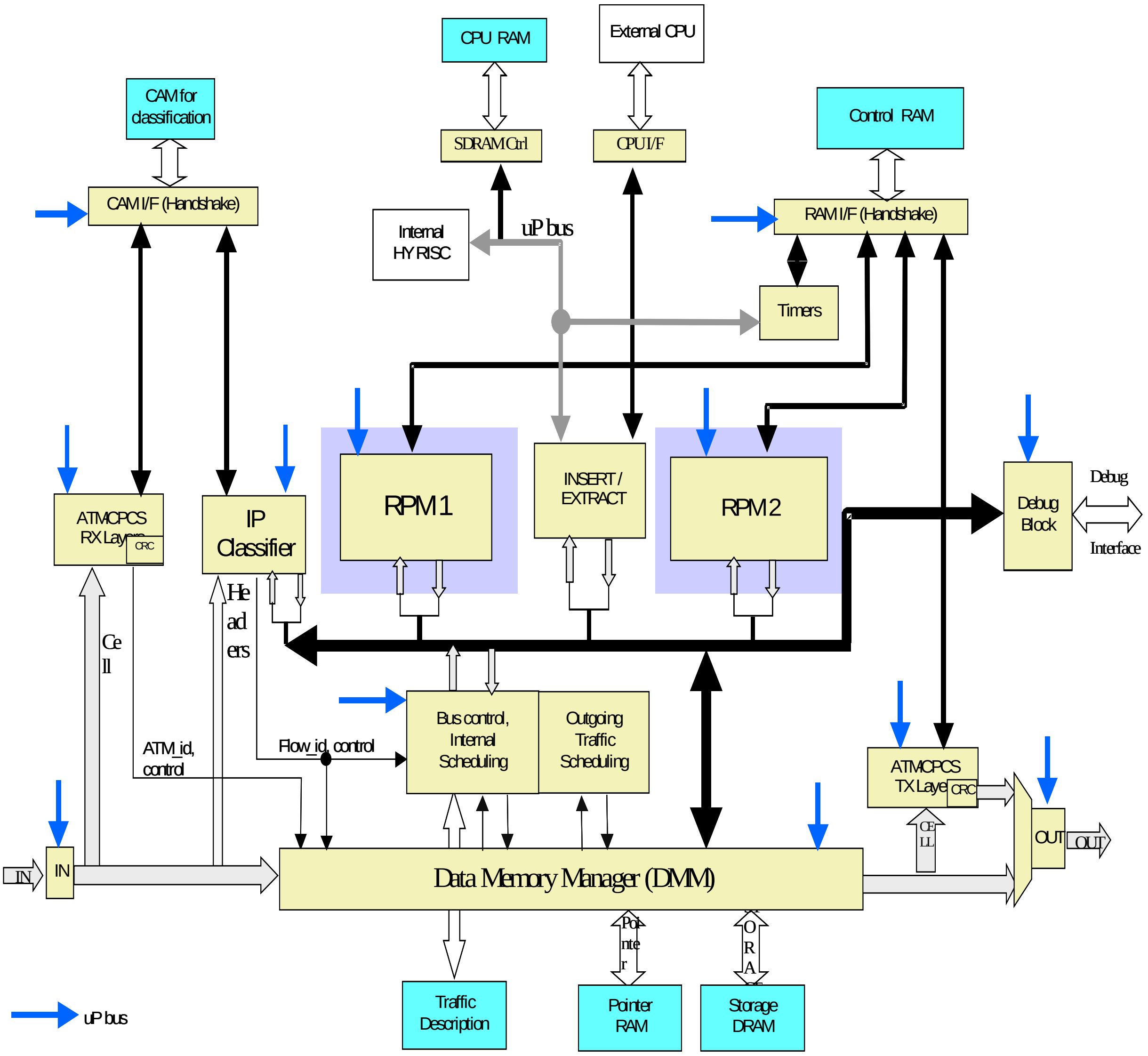


Figure 3.6a: The two RPMs in Pro^3

The following figure presents an abstract block diagram of RPM.

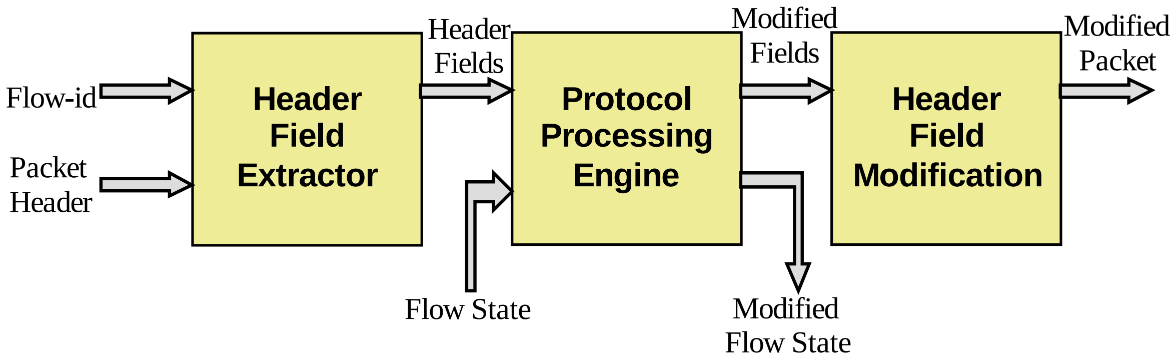


Figure 3.6b: RPM block diagram

The Re-configurable Pipelined Module (RPM) is the heart of the protocol processing of the Pro^3 system. It consists of three sub-modules, each of which performs a part of the required protocol processing. The sub-modules, shown in Figure 3.6b (detailed block diagram in figure 4a) , are Field Extractor, Protocol Processing Engine, and Packet Modifier. PPE contains a modified RISC, which is of the same architecture as the generic RISC CPU (Internal CPU Figure 3.6.2a and Figure 4a) but optimized (i.e. for efficient context switching with a set of shadow registers). For each packet, the input to the RPM module is (i) the packet's flow-id and the packet type as identified by the flow classification module, (ii) the entire flow state from the state memory, and (iii) the entire header of the packet. The filtered data are passed through the RPM pipeline to execute the protocol processing code. Between the (sub) modules, queues are used to de-couple the operations and latencies of the modules.

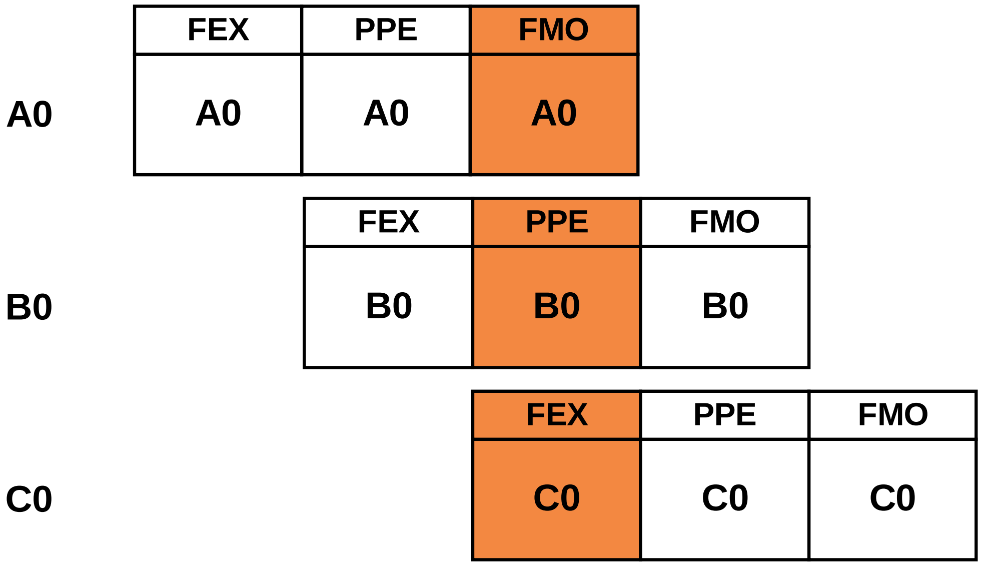


Figure 3.6d: the vertical highlighted slice shows the pipelined operation, where A0 is processed in FMO, B0 is processed in PPE and C0 is processed in FEX

3.6.1 Field Extractor

Main Features

- Extract (actually isolate) specific fields contained within a packet and forward them to the Protocol Processing Engine (PPE) of RPM for processing.
- Field extraction process can start from the packet header (or headers in case of packet encapsulation) or the trailer(s).
- Able to process data with a maximum throughput of 3.2Gbps. This is accomplished by using a 32-bit wide data path and operating clock frequency of 100MHz (half of the system clock). The aggregate throughput varies between 3.2 Gbps and 2.5 Gbps since the processing of one word may require more than one clock cycles to complete.

- The performance target is that most of the instructions will need one clock cycle to complete.

The Field Extractor (FEX) is practically a very small RISC designed for this specific task. It combines high performance and flexibility with low silicon requirements.

Specifications

The FEX operation is controlled by microcode (firmware) stored in a small internal SRAM. The instruction set comprises of simple and generic (i.e. protocol-independent) instructions that can be applied in any protocol or protocol encapsulation scheme. The internal or external CPU can download the firmware dynamically during run-time.

FEX is designed to be as much generic as possible, however it is assumed that it is integrated in PRO3 and has to decode the 64-bit control word that exists in front of each 64-byte segment sent to RPM. The FEX input is compliant to the PRO3 internal bus.

Field Extractor extracts the packet fields needed for process to PPE. The rest of the packet fields are sent to the Delay Fifo.

- The extracted fields are output in 32-bit words.
- The fields are sent to PPE through the Field FIFO, in the following order: Flow ID, PROTO, Message Type, <FIELDS>, Last word with valid Control Flags. The first three fields are required for PPE-input read requests (flow state needs flow ID, Dispatch PC needs PROTO & Message Type). The last field of the set of fields contains no valid field but valid Control Flags that delineate end of fields and end of packets (2 bits).
- Only a single header/trailer can be processed at any time. However an unlimited (theoretically) number of headers/trailers can be processed sequentially.

In case a large number of fields have to be extracted, the firmware may extract part of the 32-bit word that contains more than one field, handled as a large field. The firmware runs in the Modified HY RISC should now how to handle this case.

3.6.2 Protocol Processing Engine

The Protocol-Processing Engine (PPE) is the core of PRO^3 RPM. It consists of a modified Hyperstone RISC core, augmented with peripheral “glue” logic that implements the input and output interfaces. The input interface accepts data from the field extractor module, as well as the data (flow state information) read from the state memory (Control RAM). The Hyperstone RISC uses these data as input to the protocol processing code. The output interface sends the updated state to the state memory for writing, and informs the rest of the PRO^3 system (through the header modification -FMO) of the decisions of the protocol processing code. One such decision may be to drop or not the packet according to the firewalling policies. Figure 4a presents the detailed block diagram of RPM detailing the position and communication paths of the PPE (including the RPM Glue Logic (RPG) and the Read/Write Control RAM (RWR)).

The high-level internal structure of the Protocol Processing Engine is shown in Figure 3.6.2a. The dashed paths in MHY depict the conceptual switch achieved by changing in/out on process registers.

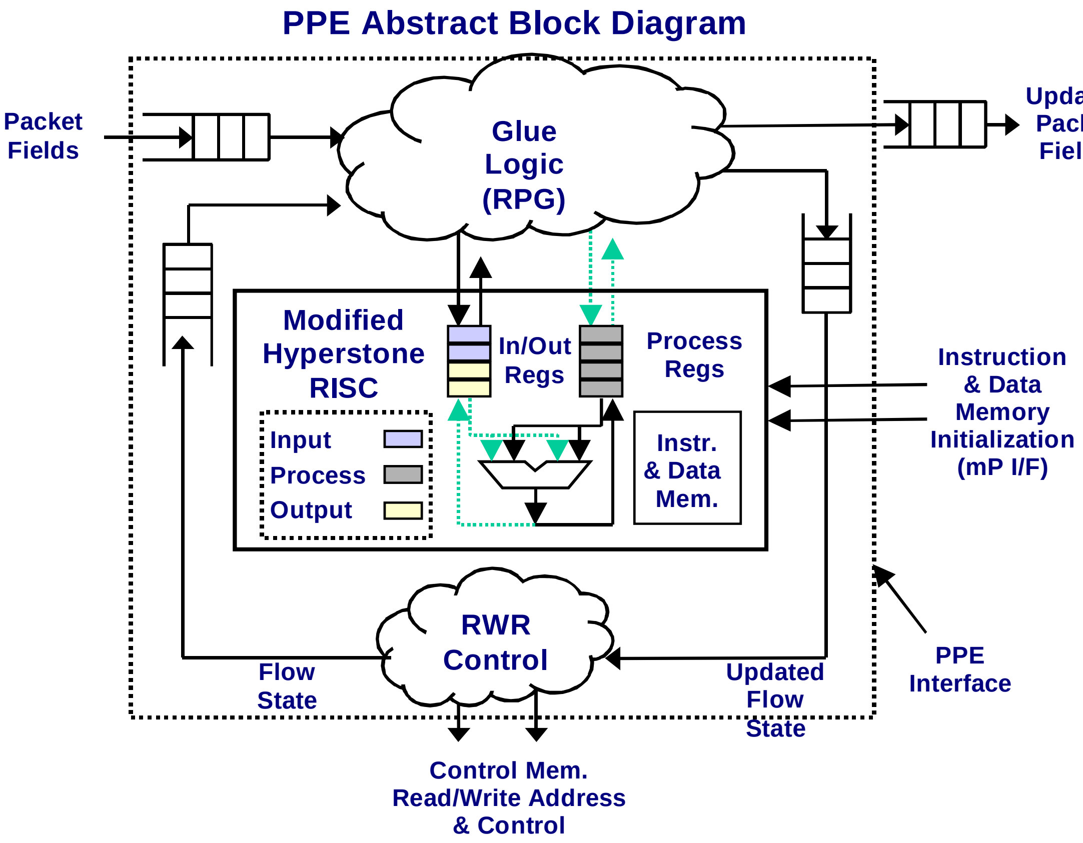


Figure 3.6.2a: The Protocol Processing Engine

Main Features

- Transfers data from Field Extractor to modified HY-RISC, and from modified HY-RISC to Field Modifier
- State information reading access from Control RAM to PPE.
- State information updating access from PPE to Control RAM.
- State information external bypass mechanism (RWR).
- Aggregate per group peak rate shaping for IP flows
- Programmable entry points for S/W packet handling according to protocol type and message type
- Programmable handling of “input” and “output” parts of flow state
- Implements bypassing of updated flow state for back-to-back processing of packets of the same flow (RPG).
- Full bandwidth input/output capacity

Specifications

PPE uses a fixed number of registers to pass the extracted data from the Field Extractor to the RISC or to read the results of processing and send them to the Packet Modifier. In case that the data to be exchanged require larger space and do not fit in the registers (e.g. the case a large message with many fields is to be processed), the glue logic is able to segment and forward them to the RISC in steps. This means that the software routine to process this message inside the RISC is segmented into several parts defined at software development phase.

The modified Hyperstone RISC includes an external read/write port to its internal register file. During the processing of each packet, the internal registers are (conceptually) partitioned in two sets: working and I/O registers. The working registers are the ones that are used by instructions (and only by instructions) for the purpose of executing the protocol processing code. The I/O registers are accessed only through the external port by the interface logic. The purpose of the I/O registers is twofold: (1) to prepare the data necessary for the processing of the *next* packet (input part), and (2) to output the protocol decision for the previous packet to the FMO and the updated state information to the state memory (output part). Once all the above actions are completed, the processing registers (usually) become I/O registers, the I/O registers become processing registers, and the processing of the next packet is initiated. At the same time the updated state for the just-processed cell is sent to the state memory and the next packet header and state fields are put in the input part of the register file.

The following figure presents the pipeline stages of RPM system,

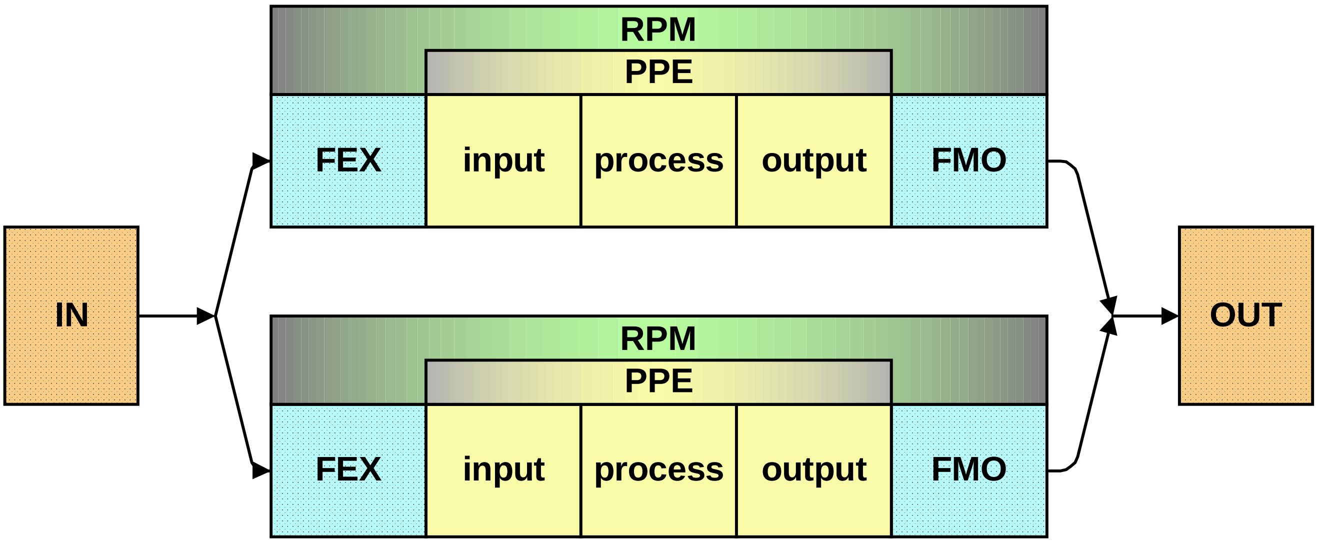


Figure 3.6.2b: RPM's pipeline stages

3.6.3 Field Modifier

Main Features

- Operation 1: Compose a packet (protocol message) according to a specific template and put in user-defined positions a number of fields received as input.
- Operation 2: Modify (actually replace) specific fields contained within a packet. As input packet an entire packet (or part of packet) can be used. This data is forwarded by FEX and delayed in the Delay FIFO of the RPM.
- Operation 3: Create an internal message for another component. This does not have the form of a protocol message but it is according to PRO3 internal control commands.
- Field Modification process can start from the packet header (or headers in case of packet encapsulation) or the trailer(s).
- Able to process data with a maximum throughput of 3.2Gbps. This is accomplished by using a 32-bit wide data path and operating clock frequency of 100MHz (Half of the System Clock). The aggregate throughput varies between 3.2Gbps and 2.5Gbps since the processing of one word may require more than one clock cycles to complete.
- The performance target is that most of the instructions will need one clock cycle to complete.

The created packet can then be forwarded to the PRO3 internal bus towards another component. The fields used to create a packet come from the RPM RISC core and can be part of the packet header (or headers in case of packet encapsulation) or trailer(s) or even control fields for internal commands.

The Field Modifier (FMO) is a small RISC that performs the dual operation of the Field Extractor. It combines high performance and flexibility with low silicon requirements.

Specifications

The FMO operation is controlled by microcode (firmware) stored in an internal SRAM. The instruction set comprises of simple and generic (i.e. protocol-independent) instructions that can be applied in any protocol or protocol encapsulation scheme. The internal or external CPU can download the firmware dynamically in run-time, through the microprocessor interface. The SRAM is dual port and is seen by the CPU as a memory peripheral.

FMO is designed to be as much generic as possible, however it is assumed that it is integrated in PRO3 and has to encode the 64-bit control word that exists in front of each 64-byte segment or control message sent to the PRO3 internal bus. The FMO output is compliant to the PRO3 internal bus specification.

FMO is assumed to comply with the overall processing architecture of PRO3. Time is distinguished into Processing Slots, where each Processing Slot can be variable in time and is able to accommodate and process either up to 7 PRO3 segments of a large packet or an entire packet of less than 7 segments. Each segment is 64-byte long.

The module operates on the packet data in the following manner:

- In general, we assume that after a message (packet) processing, PPE can generate several sets of fields for composing relevant messages as results.
    - One of these field sets will modify the packet waiting in the Delay FIFO of RPM and the modified packet will be sent to the right destination.
    - One or more set of fields can be used to construct new packets to be sent to DMM or to other PRO3 blocks.
    - One or more set of fields can be used to construct control messages to other PRO3 blocks.
    - Reject the packet waiting in the Delay FIFO.
- The above correspond to the following main modes of operation of the FMO block.
    - **Operation 1: Packet composition**: Compose a packet (protocol message) according to a specific template (known by the firmware) and putting in user defined positions a number of fields received as input.
    - **Operation 2: Packet Modification**: Modify (actually replace) specific fields contained within a packet. As input, an entire packet (or part of packet) can be used. This data has been forwarded by FEX and waited in the Delay FIFO of the RPM. This data is exactly the packet or part of packet that was sent to the RPM for processing (corresponds to one Processing Slot).
    - **Operation 3: Packet Creation**: Create an internal message for another component. This does not necessarily have the form of a protocol message but it is according to PRO3 internal control commands.

  PPE specifies the command to be performed through the CMD FIFO.
- The packet data are internally shaped and processed in 32-bit words.
- The new fields are received in 32-bit words from the PPE.
- The fields are received through the Field FIFO, in the following order: PROTO, Message Type, Subtype, <FIELDS>, Last word with valid Control Flags. The first three fields are required for FMO firmware branches. Message Type and subtype can also be used for packet composition. The last field of the set of fields contains no valid field but valid Control Flags that delineate end of fields and end of packets (2 bits).
- Only a single header/trailer can be processed at any time. However an unlimited (theoretically) number of headers/trailers can be processed sequentially (corresponding to encapsulated protocols).

In case a large number of fields have been generated as a result of a packet processing, the PPE glue logic is responsible to send all these fields to the FMO. However, in case the FMO FIFO becomes full, the PPE logic has to wait and then send the remaining fields.

Chapter 4: PPE Architecture

Position in PRO3 System

PPE is part of the two RPMs instances RPM 1 and RPM 2 as shown in figure 3.6a. PPE module consists of three modules: (i) the Hyperstone Modified RISC, (ii) the RPM Glue Logic, and (iii) the Read/Write Control RAM module (RWR). The following figure presents the block diagram of RPM detailing the position and communication paths of the MHY.

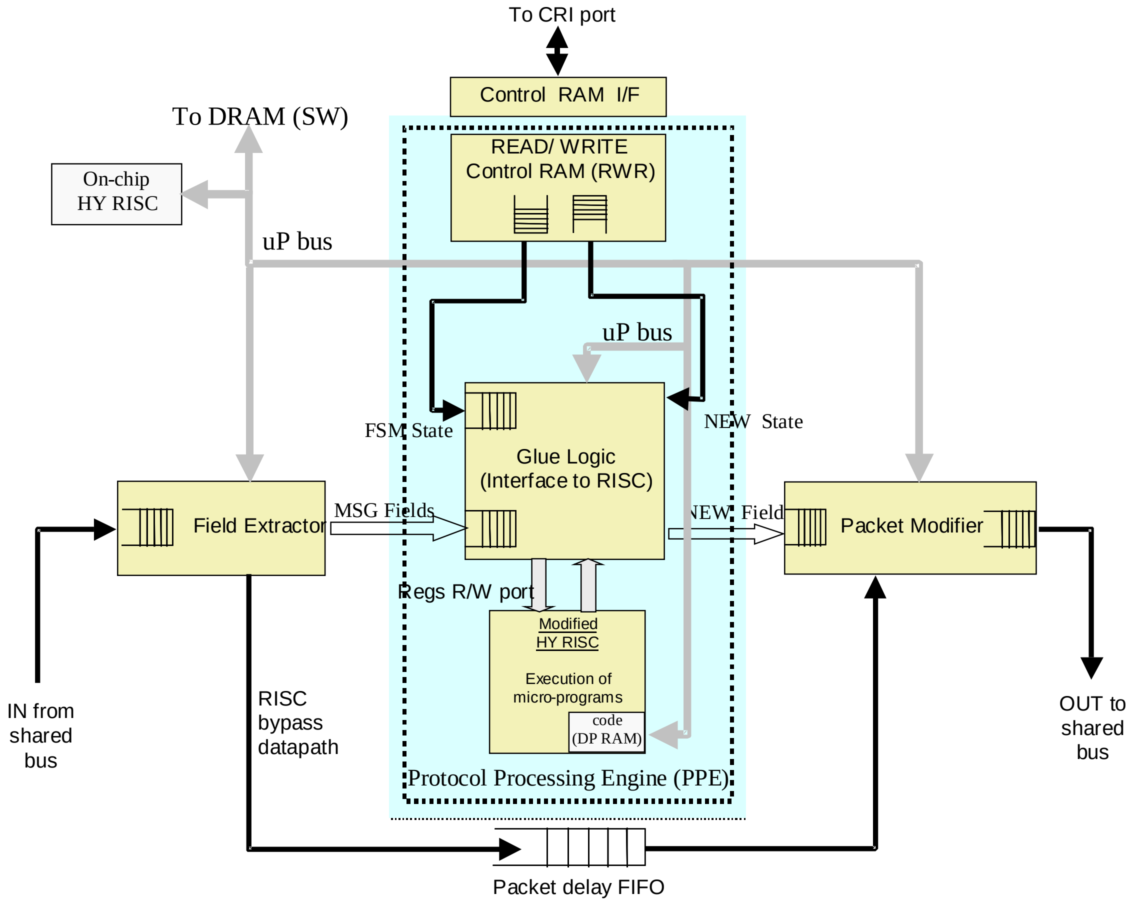


Figure 4a: PPE location inside the RPM along with its main communication paths

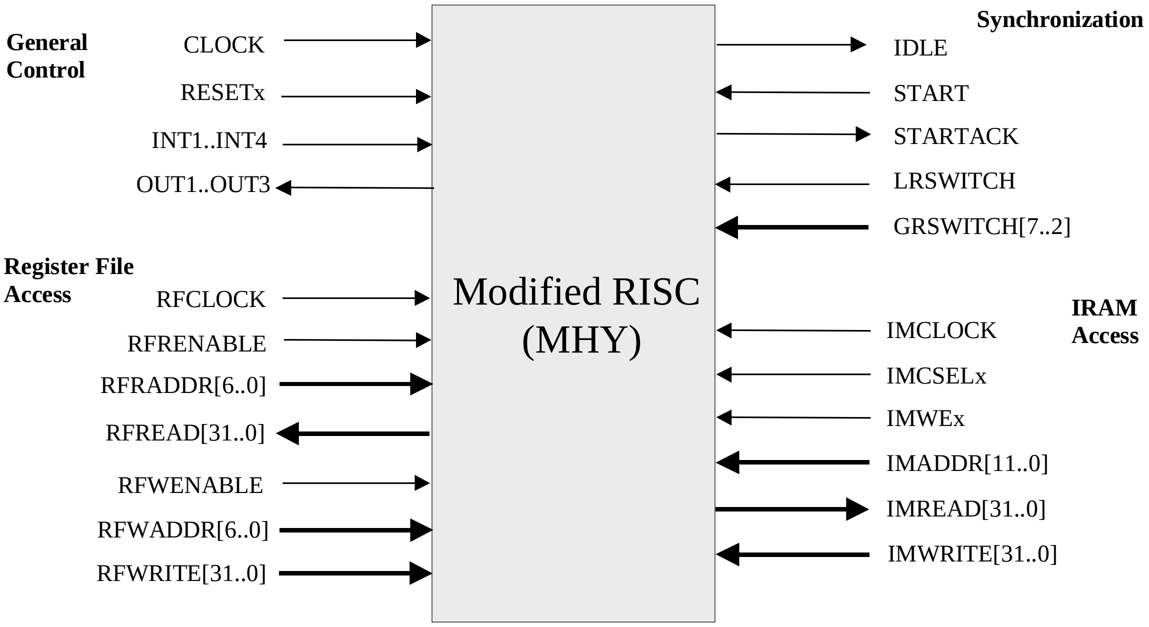
4.1 Modified Hyperstone RISC (MHY)

Main Features

- 32 global and 64 local registers of 32 bits each, 16 global and 16 local registers directly addressable
- Two sets of 14 global registers (12 of them are used for flow state) and 64 local registers (for packet fields transfer) are accessible from outside the core via a special port

- Core accesses are switchable between the two sets of 12 global registers and between the two parts of a 32+32 register partitioning of the 64 local registers
- 16 KB dual-ported internal memory (IRAM) with the second port accessible from outside the core
- Special instruction for power-down control and synchronization with RPM glue logic

Interfaces:



General Description

The Hyperstone Modified RISC (MHY) module is the central packet-processing element in the PRO3 system. It interfaces to the RPM Glue Logic through the synchronization lines, the special register file access port, and the internal memory access port.

After reset, the MHY begins executing its firmware that has been put into the internal memory by the control processor (internal Hyperstone CPU). Following the initialization, the MHY enters its idle state.

To process a packet, the following steps are executed:

- The RPM Glue Logic places extracted fields from the packet as well as state information about the flow into the MHY register file and/or internal memory. When this is complete, the LRSWITCH and GRSWITCH signals are set to indicate the correct packet data locations, and the MHY is started with the START signal.
- The MHY uses one of the input registers to dispatch to the correct routine for packet processing, and processes the data to produce updated state information and, if applicable, output packet fields. While this process is running, the RPM Glue Logic may use the

second part of the register file to already load the field and state information for the next packet.

- The MHY signals completion of the packet processing routine by asserting the IDLE signal. If new data has been loaded, a new packet processing operation can be started immediately. The RPM Glue Logic can now read the updated field and state information from the MHY register file and/or internal memory. In case of the MHY register file, *this readout may overlap with the writing of the next packet fields and state*.

Modifications compared to standard Hyperstone Core

The Modified Hyperstone RISC (MHY) is a derivative of the standard Hyperstone E1-32XS microprocessor core. Compared to the standard E1-32XS macro cell, the following things have been modified:

- An additional read and an additional write port has been added to the register file. Through the added write port, state and packet information can be put into the register file by the RPM glue logic, and through the added read port, updated state and packet information is read out by the RPM glue logic.
- The local register part of the register file has been partitioned from 64 registers into 2 × 32 registers, and the global registers G2..G15 have been duplicated. When this partitioning has been enabled by the MHY, the RPM glue logic can select which of the 32 local register partitions is used by the MHY on a local register access, and it can select in which global register block is used by the MHY on a global register access. The selection for the global registers can be done independently for G4 and G5, for G6 and G7, … G14 and G15.
- Power-down and synchronization logic has been added to start processing when a new set of state and packet information is available in the register file to process, and to signal completion of this processing. When no processing is active, the MHY switches off most of its clocks until the next packet processing starts.
- The DSP instructions and the related logic parts of the DSP execution engine have been removed in order to regain G14 and G15 as general purpose global registers. The DSP execution engine itself has been kept in order to support the ALU multiply instruction.
- Almost all of the external bus interface logic has been removed.
- For initial program load, a separate data path to the internal memory has been added so that the internal CPU can access the MHY IRAM for initialization and for status inquiries. After the MHY IRAM has been initialized, the MHY reset exception begins booting from the code in IRAM.

Examples of block Operation

In this section, signal diagrams are given showing typical operations for the power-sown synchronization, for the register file access, and for the IRAM access.

IDLE-START and Power-Down Synchronization

The following figure shows the signal forms for the IDLE, START and STARTACK signals and their relation to the internal core clock showing two different cases. Both cases assume that the sequence of instructions is:

```
...
EMUL L0, L0     ; set power-down
MOV  PC, G5     ; restart at new entry
...
Packet_type1:
...             ; packet processing commands
```

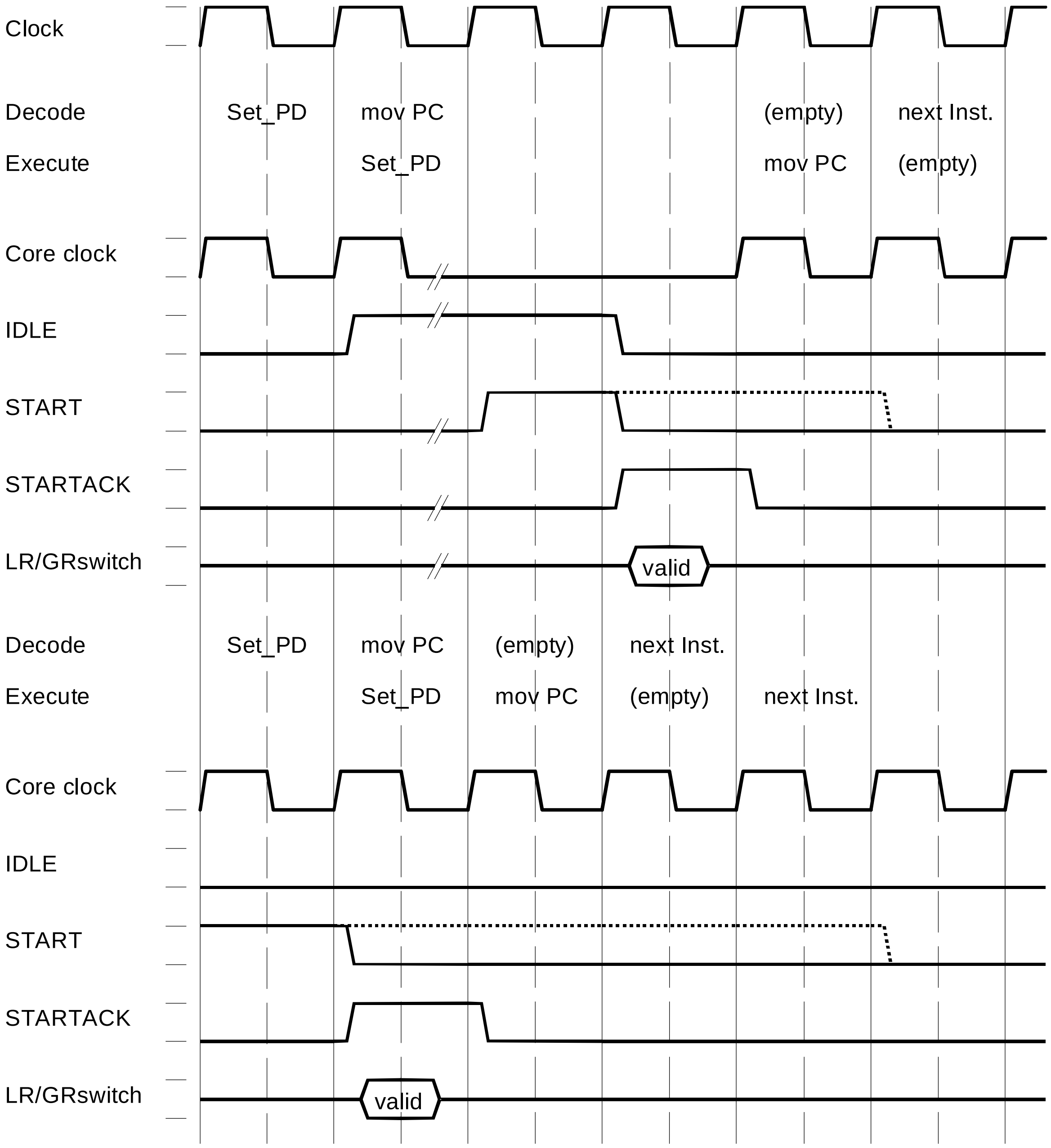


Figure 4.1a: IDLE, START, STARTACK synchronization

The first case assumes that the START signal is not asserted during the decode cycle of the Set Power-down instruction. The second case assumes that the START signal is already asserted during the decode cycle of the Set Power-down instruction.

In the first case, the core clock is suspended for at least one clock cycle. The IDLE signal indicates this fact.

When the START signal is activated (found high on a falling system clock edge), the core begins wakeup from sleep mode and performs the transition back to normal working mode. The STARTACK signal is asserted for one clock cycle in the following system clock cycle. The LRSWITCH and GRSWITCH signals are latched on the falling system clock edge in the clock cycle where STARTACK is asserted. In the system clock cycle following the assertion of STARTACK, the core resumes execution with the instruction following the Set Power-down instruction.

START may be asserted for more than one clock cycle without disturbing the functionality. This first case is also valid if the START signal is asserted during the same clock cycle when the IDLE signal is asserted.

In the second case, the core does not actually enter power-down mode since the START signal is asserted early. Thus, there is no transition of the IDLE signal. During the execution cycle of the Set Power-down instruction, the STARTACK signal is asserted to indicate that there has been a wakeup from a (not executed) power-down command, and in this same cycle also the register switch signals are latched on the falling system clock edge.

Register File Access

The MHY register file consisting of the 64 local registers and of two sets of the global registers G2..G15 are accessible through a special register file access port. For the register file access from outside, there are the LRSWITCH and GRSWITCH control signals.

The core accesses the local register stack in two partitions of 32 registers each. If the LRSwitch is set to 0, the core accesses the lower 32 local registers (0-31 L0), and the higher 32 local registers when the latched LRSwitch signal is 1(32-63 L1).

Global registers’ switch is more complicated than local switch, this will be discussed in detail in the next chapter. When RPG transfers flow state to some of the global registers (from register 15 down to 4) indicates these registers by the GRSwitch mask (GRSwitch [7..2]). Every bit of this mask corresponds to a specific couple of registers and defines whether the state is in Global part 0 or Global part 1 (figure 4.1b).

For example:
GRSwitch [7]='1' means that registers 15 and 14 of global part 1 (G1) are filled with flow state.
GRSwitch [5]='0' means that registers 11 and 10 of global part 0 (G0) are filled with flow state.

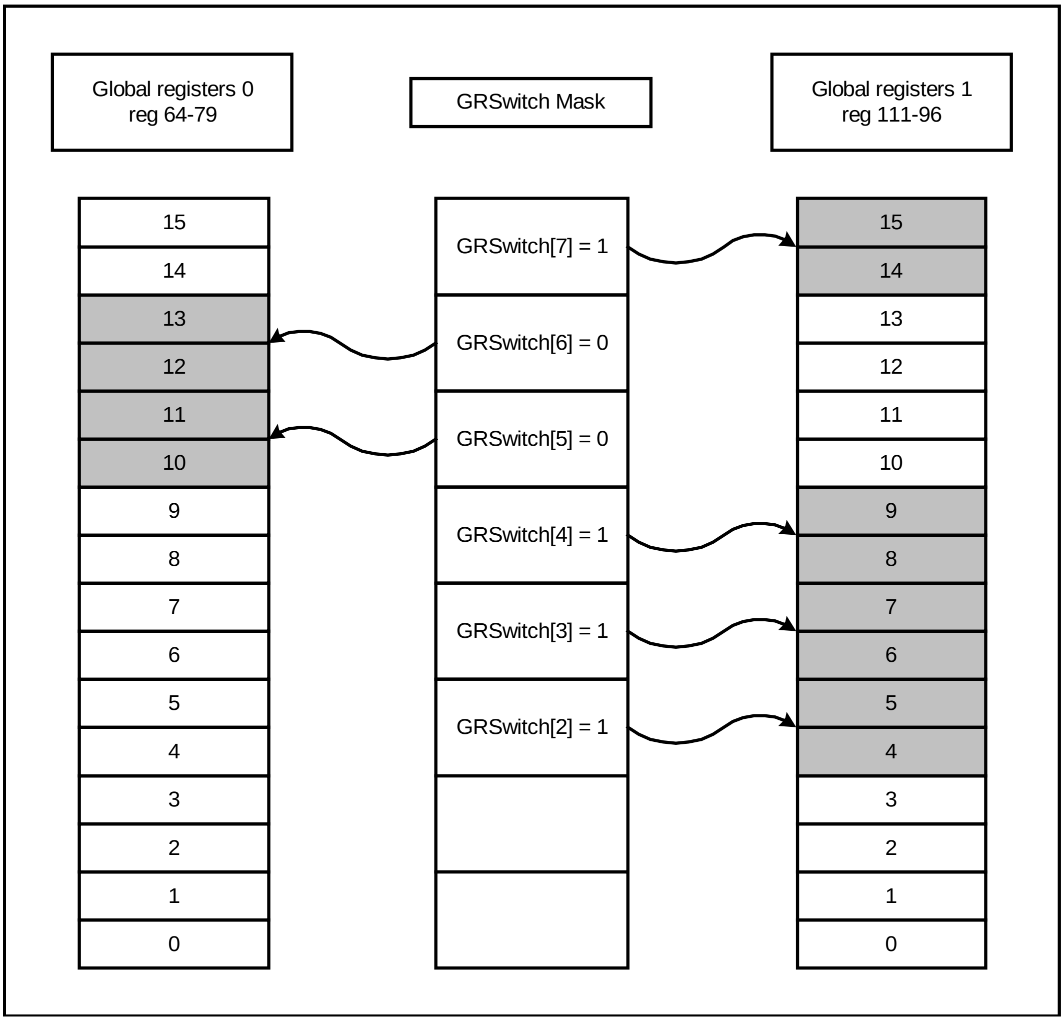


Figure 4.1b: The grey boxes show the registers that carry flow state when GRSwitch [7 down to 2] = "100111".

The signal diagram for read and write accesses through the special register file access port is given in the following figure. Simultaneous read and write accesses to the same address are valid, in this case, the read returns the old register contents while the new data is written into the register. A write access to registers currently owned by the MHY core is not allowed.

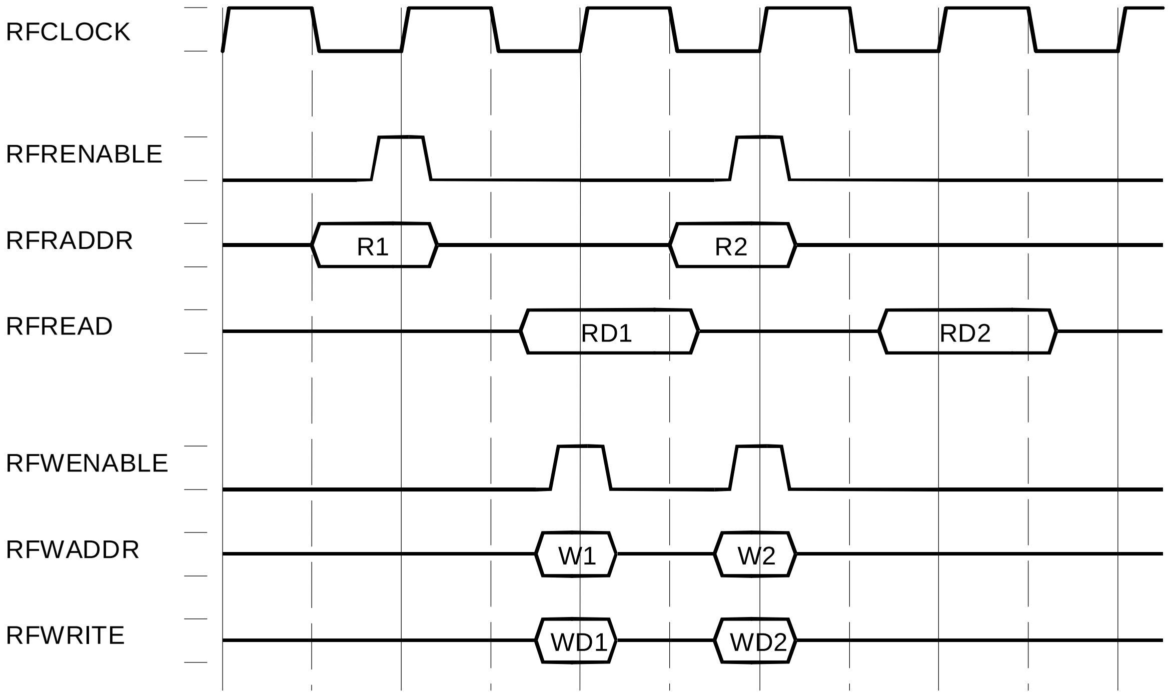


Figure 4.1c: Register File Access

In this example, two read accesses are performed to addresses R1 and R2 that are separated by one idle clock cycle. The read data RD1 and RD2 are available *after the next falling clock transition*. Two write accesses are performed to addresses W1 and W2 with write data WD1 and WD2. The case R2=W2 is allowed, in this case WD2 is written into the addressed register, and RD2 is the old contents of the register.

All input signals are referenced to the rising clock edge. Only for the read address, a larger setup time in the order of 2 ns is required. The read data output is characterized by the delay from the rising clock edge.

IRAM Access

The internal memory (IRAM) of the E1-32XS MHY core is a fully static synchronous dual-ported memory. One read/write port is used by the core to access the IRAM for code fetches and data loads or stores, the second read/write port is available to the PRO3 RPM logic to access the IRAM.

The IRAM is organized as a 4K×32-bit SRAM, addressed by 12 address lines. Read and write accesses always transfer a full 32-bit word.

4.2 RPM Glue Logic (RPG)

General Description

The RPM Glue Logic (RPG) interfaces and transfers data between the fields extractor, the modified Hyperstone RISC, the Field Modifier and the Read/Write Control RAM module. The operation of RPM for incoming packets consists of:

1. Transferring packet fields from the Field Extractor to the local portion of the registers of the modified Hyperstone core,
2. Requesting reading of the corresponding flow state information form the Read/Write Control RAM,
3. Transferring the read flow state from the Read/Write Control RAM (RWR) to the global portion of the registers of the modified Hyperstone core,
4. Initiating packet processing when the modified Hyperstone core has completed processing of the previous packet.

The operation of RPM for outgoing packets consists of:

1. Interpreting the outcome of the packet processing,
2. Transferring updated packet fields from the local portion of the registers of the modified Hyperstone core to the Field Modifier if so indicated by the S/W,
3. Transferring updated flow state information from the global portion of the registers of the modified Hyperstone core to the Read/Write Control RAM if so indicated by the S/W.

Additionally, the RPG has to:

1. Maintain consistency between the flow state information used in the packet processing, by means of bypassing. When packets of the same flow are processed back-to-back, the state flow read from the Control RAM is stale, and the RPG undertakes the responsibility to forward internally the correct flow state information,
2. Implement the necessary control and status registers for the operation of the PPE, and perform reset and support the initialisation sequences for PPE.

Specifications

The RPG consists of two major logical units, the input and the output, communicating with the Field Extractor and the Field Modifier respectively. Supporting these two logical units are interfacing units (to the internal Pro^3 processor) and to the RWR module. More specifically:

The INPUT-FSM is responsible for transferring the packet and state fields into the Hyperstone RISC’s register file (I/O part). Additionally, it inserts a “dispatch” code in the register file. This code will depend on the type of the packet and the implemented protocols, and is used by the software to vector the execution directly to the part of the code that handles the appropriate type for this packet. To achieve this functionality, the INPUT-FSM needs to maintain a table

mapping packet types (as defined by the flow classification module) to the corresponding entry point for that type of packet in the protocol processing software.
The OUTPUT-FSM is responsible for transferring the updated state fields to the state memory, and also to inform the rest of the PRO^3 system (most probably just the scheduler) of the decisions taken by the protocol processing code. Since the result of a packet processing can be more than one PRO3 internal commands/messages to respective blocks, the OUTPUT-FSM will offer a mechanism to delineate more than one messages/commands to Packet Modifier.

The synchronisation logic has to match the variable processing time of a packet to the (also variable) time of the newly arriving packets, etc. This synchronisation is mainly achieved through de-coupling input and output queues and their associated simple handshake.

There is also an interface to the Control RISC, for reasons of initialising and monitoring.
RPG is described in more detail in chapter 6.

4.3 Read/Write Control RAM (RWR)

<u>Specifications</u>

The RWR module performs 3 major tasks. It provides to the Input sub-module of the RPG, from the Control RAM, the appropriate state information for each incoming packet type. It also updates the latter with the newly processed state information and finally through a state bypass mechanism it ensures that consistent and updated state information is given to the processing core. In more detail these three basic functions are:

State Information Reading access:

The RWR module acquires the necessary FID information from the input portion of the RPG module. This information is placed in a FIFO (queue) and at the same time the Control RAM is accessed in order to receive the state information needed by the Input sub-module of the RPG. So the newly arrived packet has all the necessary state information at least 1 stage before it enters the processing stage (the Input FIFO accepts only two states). This is an ideal situation where no bypass strategy is needed or at best it is dealt with, internally into the PPE (RPG input block – Modified HY-RISC – RPG output block)

State Information Updating/Writing access:

The RWR sub-module takes the newly updated state information (more likely a portion of it) from the PPE from the output module of the RPG and places it into a FIFO. Next, the Control RAM I/F module (CRAM_IF) receives this information and updates the Control RAM.

State information Bypass mechanism:

As it is already mentioned, the RWR sub-module ensures that consistent state information is given to the PPE. But due to the nature of the PPE (and to the entire RPM) where packets are processed in a pipelined fashion, and hazardous situations are likely to occur (state information read from Control RAM before the latter is updated), there must be a bypass mechanism to ensure that this consistency is not breached.

This is the case when the updated state information has exited the processing stage of the PPE but has not been written yet to the Control RAM and at the same time the RWR accesses the Control RAM for that state type. So the Input sub-module has received "stale" (out of dated) information about a specific packet.

In order to avoid this situation and to bypass data externally, the RWR module holds the updated state data into a write buffer for a specific period of time. So when a packet (with obsolete state information) is entering the processing stage, the write buffer "feeds" it with the correct state data by multiplexing part of the state data.

The size of the buffer is a main issue to the bypass technique since it is mainly dependent on the size of the FIFO of the input sub-module. Write buffers can store state for eight flows that has not been written to Control RAM or that is written after state of the same flow has read from Control RAM. This is because RWR can keep into its FIFOs four write state requests, and three read packets.

RWR block diagram

The block diagram of the RWR is shown (as part of the block diagram of PPE) in Figure 4.3a.

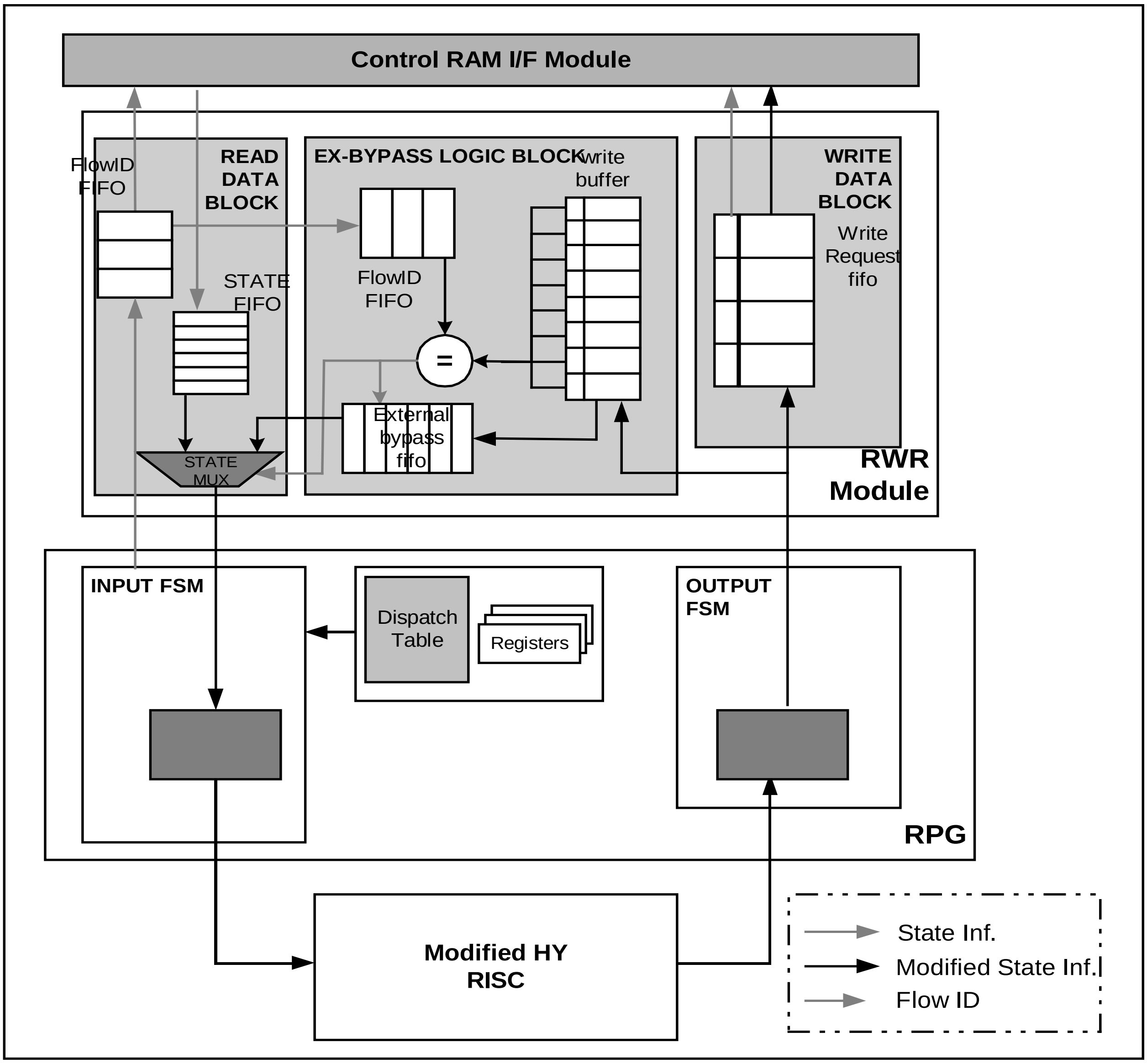


Figure 4.3a: RWR block diagram

RWR internal structure and operation

RWR supports three communication ports with the RPG and one with Control RAM Interface Module (CRIM, Control RAM stores flow state information). The communication with the RPG is mode interesting and is described in the following three subsections.

Reading Flow State Information

In order to read the flow state information, the RPG sends the FlowID to the RWR, which places it into a read request queue, and acknowledges the receipt of the FlowID to RPG. If the request queue is almost full, it asserts a “Busy” signal, which is propagated to FEX and stops the arrival of more packets. When the Control RAM is available, the RWR reads the state information and places the data in the state queue of the RPG. This operation is assumed that will *never* block, i.e. the RPG must implement enough storage to accommodate all the state fields that it has requested. If the requested flow state exists already in the State Write Buffer (see next sub-section), the state fields that are available in the State Cache overwrite the fields read from the Control RAM. The combined information is up-to-date, and is supplied to RPG.

Writing Flow State Information

In order to write the updated flow state information, the RPG sends the FlowID, the index of the changed field(s) and the modified words to the RWR, which places them in two distinct buffers. The first buffer is a *Write Queue*, and the second is a *State Cache*. The operation of the state cache will be described in the next sub-section. The write queue is monitored for entries by an FSM that dequeues the words and writes them to the Control RAM as fast as possible. The writing of the updated state is completed as soon as all the words have been sent to the Control RAM.

Bypassing Flow State Information

Bypassing occurs when a read request finds that an outstanding write request exists in the RWR. One more case of bypassing exists. If during a state read request a packet with the same FlowID is under processing, parts the flow state read from the Control RAM are stale, as they will be updated in the near future after the processing of the earlier packet. However, it is not possible to bypass at this point, as the updated state is not available yet. Therefore, the reading of the Control RAM proceeds normally, and the state is placed in the State Queue. Upon arrival of this FlowID in the Input stage of the RPG, the Input FSM checks with RWR for the existence of bypass condition. By this time, the updated state will be available to the RWR, and if this case is identified, the RWR will provide the newly updated fields, so that processing will commence with up-to-date state information.

Chapter 5: State Bypass

One of the RPG's issues is flow state transfer. Every operation of RPM Glue Logic for incoming and outgoing packets includes flow state transfer between Read/Write Control RAM module and MHY register file. Packets of the same flow id use and update the same flow state information during process. Flow state is stored in Control RAM and is read and written by RPG through RWR. RPG uses packet flow id to read and write the corresponding flow state information to the Read/Write Control RAM through RWR module. State record consists of a few 32-bit words –maximum of 12 words. The exact number of flow state words is configurable but fixed for the duration of a run. The record is split in two parts: the constant part and the updateable one - only the updateable part will be updated during process and written back to Control RAM. The updateable part of a flow state is figured out during the transfer of an incoming packet and it is specific for a flow id.

There are two groups of registers in RF that are used for flow state transfer (Global registers-G0, G1), its group consists of 16 registers (registers 15-4 are used for state- maximum of 12 registers). Input FSM transfers flow state to some of the global registers (from register 15 down to 4) and defines these registers by the GRSwitch mask (GRSwitch[7..2]). Every bit of this mask corresponds to a specific couple of registers and defines whether the state is in G0 or in G1 (§ 4.1- Register File access).

In common case (without bypass [bypass case in § 5.1,5.2,5.3]) GRSwitch mask is switched every time a new packet is transferred in RF for process (GRSwitchNew = not GRSwitchLast).

5.1 Problem Definition

In order to achieve maximum bandwidth utilization with a single flow, the processing of packets of the same flow has to occur in a pipelined fashion. However, there is a dependency between the processing of different packets via the flow state information (i.e. via the Control RAM). To achieve pipelined operation while maintaining correct operation in the presence of pipelined processing of same flow packets, it is necessary to add bypass logic to short-circuit the latest version of the flow state for the processing of later packets.

For example: assuming the sequence of packets coming from FEX to RPG A0, B, C and A1 and their corresponding flows A0, B, C and A1. Figure 5.1a shows their pass through PPE module. By the time A1 comes in RPG module, A0 has already transferred out. In this scenario there is not a data hazard and therefore, there is not a need of bypass. This is an abstract approach; the actual way of passing a sequence of packets through the RPG pipeline depends on the behavior of each stage (depends on the time that a packet spends in each stage).

A0

input	process	output
A0 Local 0 Global 0	A0 Local 0 Global 0	A0 Local 0 Global 0

B

input	process	output
B Local 1 Global 1	B Local 1 Global 1	B Local 1 Global 1

C

input	process	output
C Local 0 Global 0	C Local 0 Global 0	C Local 0 Global 0

A1

input	process	output
A1 Local 1 Global 1	A1 Local 1 Global 1	A1 Local 1 Global 1

Figure 5.1a: no bypass case

Another scenario: packets A0, B, A1, A2 of flows A, B, A, A come to RPG. Figure 5.1b shows the status of pipeline stages. The process of A0 will update state of flow A. Therefore, upon input of A1 RPG must detect the data hazard and short-circuit the updated state of flow A, and upon input of A2, RPG must detect an other data hazard.

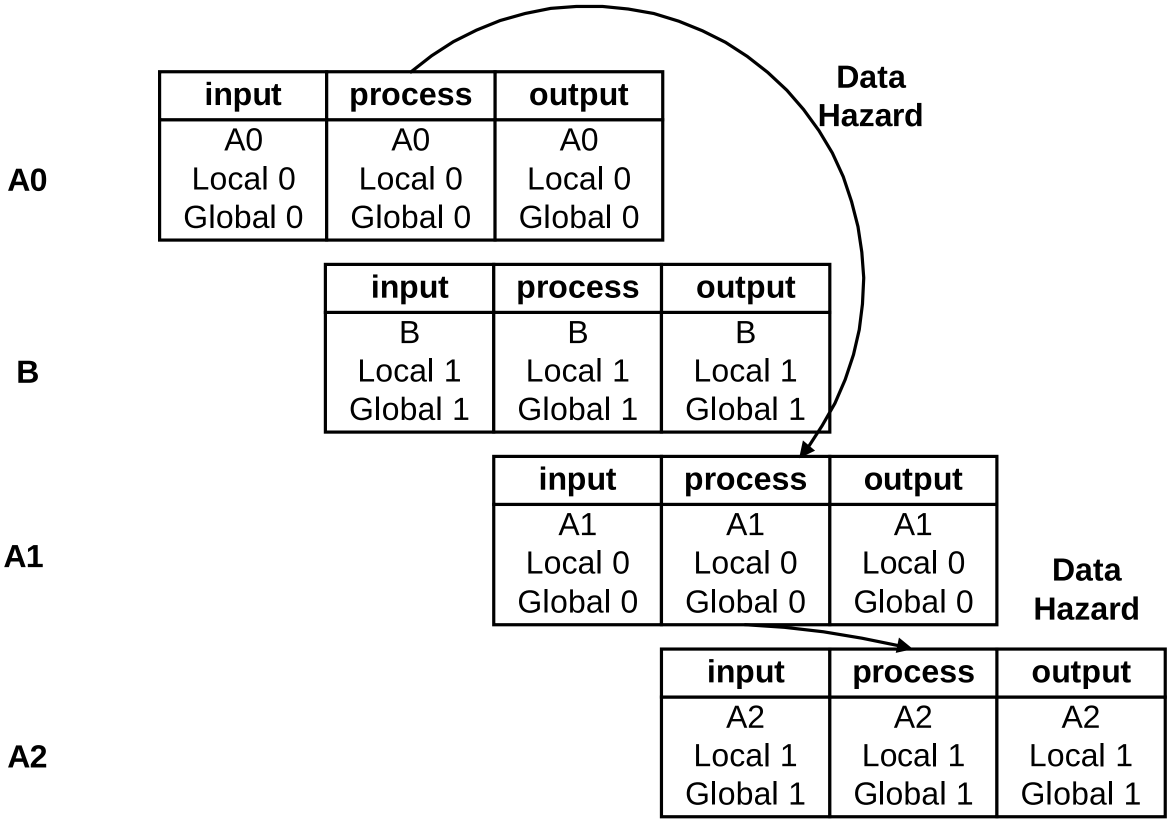


Figure 5.1b: bypass L1 – bypass L0

5.2 Bypass Logic

RPG divides PPE module in three pipeline stages: input, process and output. Each pipeline stage can be busy with a different packet at the same time (figure 5.1a-figure 5.1b). Assuming that packet ‘A1’ is transferred from FEX to MHY’s RF (input stage), its flow state “StateA_old” is already in the input state FIFO and one or more of the other two stages (process, output) is busy with another packet of the same flow ‘A0’. At the time ‘A1’ start processing, its flow state information, which was stored at input state FIFO, will not be the latest version. This will happen because the process of ‘A0’ has updated “StateA_old” into “StateA_new”, but the updated version is stored to RWR and to Control RAM after input stage has read “StateA_old” from RWR. Bypass logic of RPG is responsible of locating the latest version of state, in MHY register file, during the transfer of an incoming packet. Bypass logic compares the flow id the incoming packet with the flow id of the previous packet and the one before that. In this case RPG need a mechanism to detect and locate the latest version of flow state in MHY register file.

There are two kinds of RPG bypass, which are defined by the distance of packets of the same flow that are in the pipeline stages of RPG and MHY. Here we define the bypass distance to be the number of other packets that separate two packets of the same flow in the PPE pipeline. In case packets of the same flow are not separated by another packet there is bypass level 0 (bypass L0) and if there is a packet between them there is bypass level 1 (bypass L1). The necessary actions according to the bypass distance are as follows:

1. While the second packet 'A1' is in input, the processing of its immediately preceding packet 'A0' is producing the correct state, which will be found in the registers that packet 'A0' used during its process (bypass L0-figure 5.2a). Therefore, upon processing of the second packet, we need not to switch the mask (GRSwitchNew <= not GRSwitchLast), but instead to reuse the global registers 'A0' used for the state fields (GRSwitchNew <= GRSwitchLast). This refers, of course, only to the updateable part of flow state, for the rest of the Global registers the corresponding bits of mask will switch. The non-updateable part of state for the current PPE may be updated by the PPE of the other RPM, or by another module of PRO3 chip. Therefore, this part of flow state is transferred from input state fifo to RF either there is bypass or not.

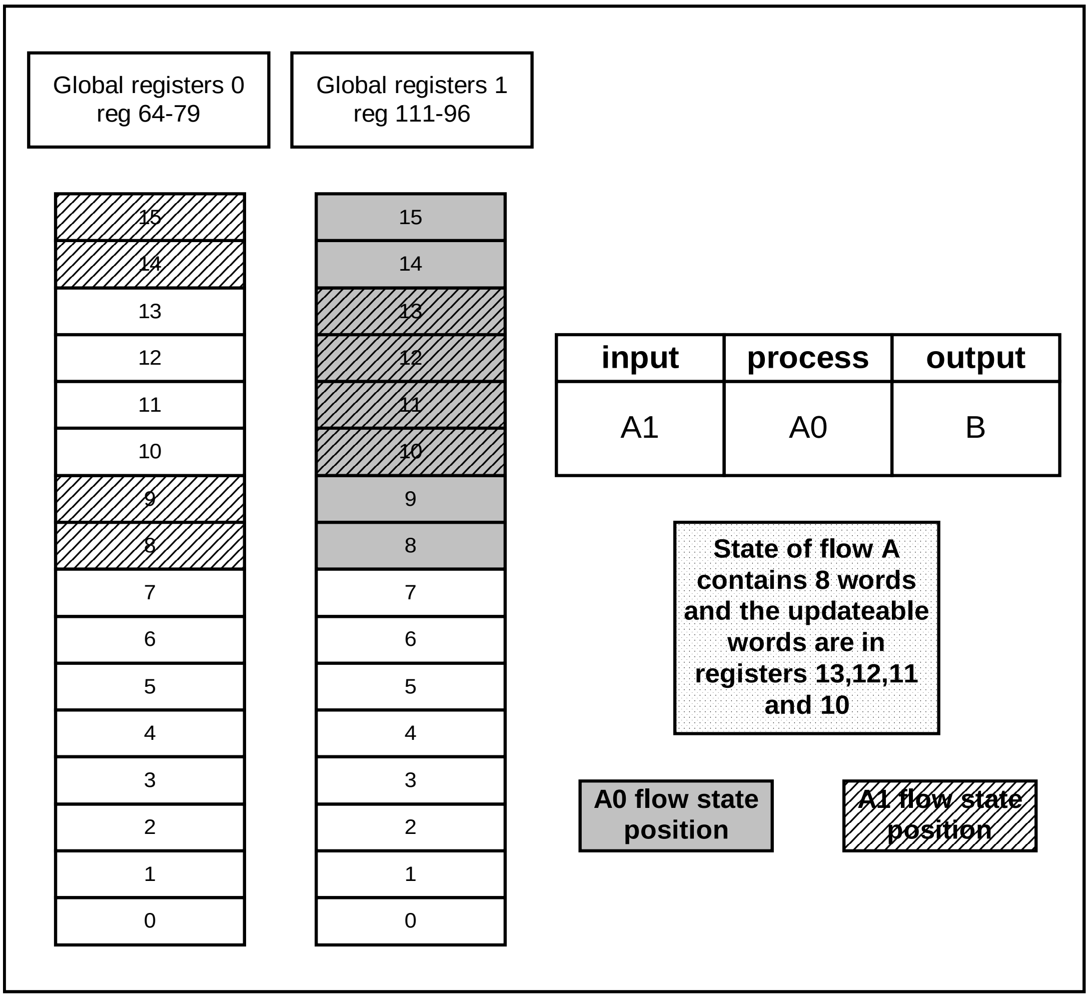


Figure 5.2a: the latest version of Flow State A already exists on G1 (registers 13,12,11 and 10)

While the second packet is in input, the output of the preceding same-flow packet is ongoing and there is a packet of another flow in ‘process’ stage (bypass L1-figure 5.2b). The Control RAM is in the middle of an update (or perhaps is not updated at all yet) with the latest information. Thus, we cannot ensure that the read Flow State is up-to-date. Therefore, a bypass path through the Control RAM interface logic of the PPE will supply the proper (up-to-date) information to the input FSM. The input FSM will then ignore the fields’ read from the Control RAM and use the bypassed fields that are up-to-date. In this scenario the necessary information is already in the IN/OUT registers (GRSwitchNew = not GRSwitchLast). The input FSM will *not* write the updateable part of stale flow state information it read from the Control RAM, and the registers will be switched as in any regular case. The only case that flow state of the outgoing packet is in the registers that process uses at the same time is when outgoing and processing packets have the same flow id [§ 6.3]. However, this is the case of bypass L0.

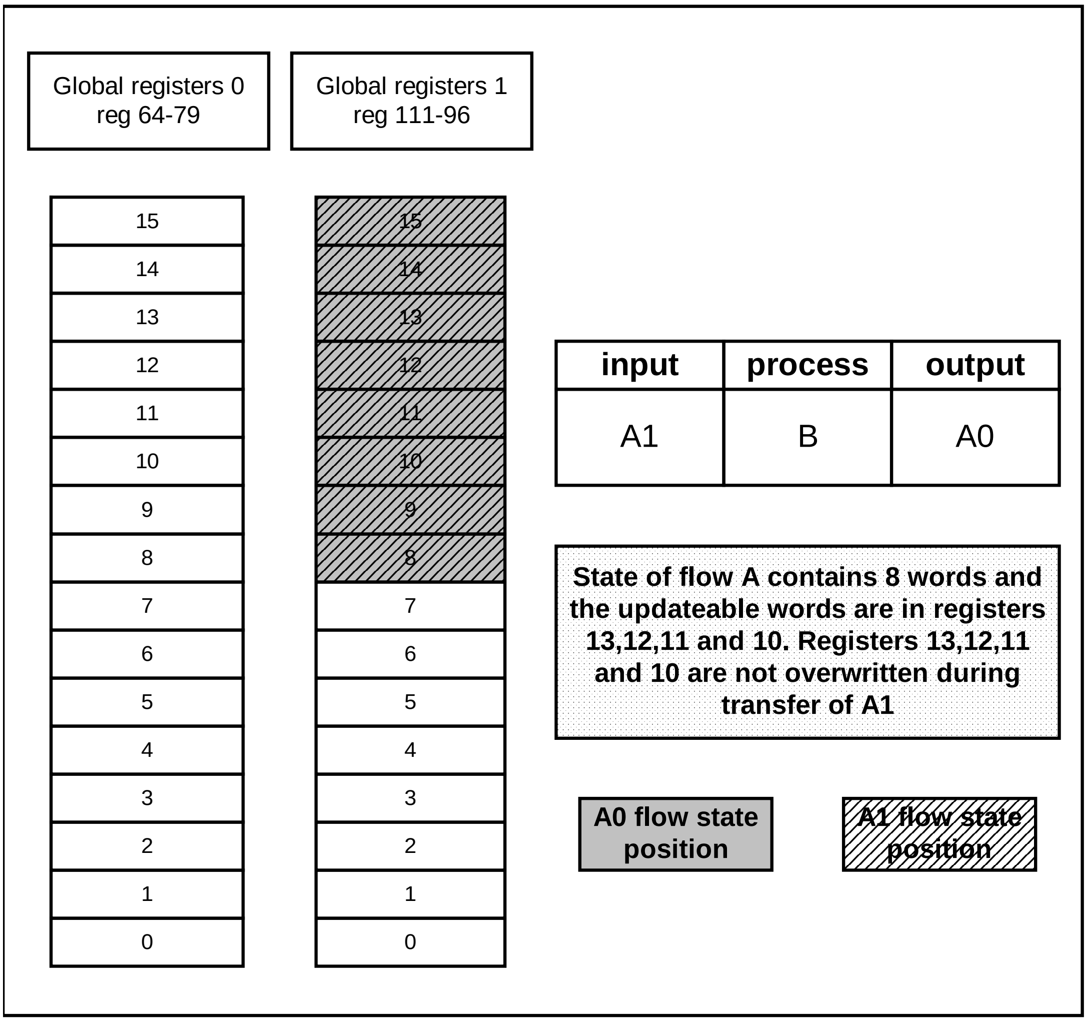


Figure 5.2b: the latest version of Flow State A already exists on G1 (registers 13,12,11 and 10)

The algorithm that detects bypass cases and decides which the part of global registers will be used for state transfer is presented next:

```
If (current_flowid = last_flowid) then
//bypassL0: the updateable part of global registers is the one
//created by the previous packet
bypass_L0;
elsif (current_flowid = before_last_flowid) then
//bypassL1: the updateable part of global registers is the one
//created by the packet before last
bypass_L1;
else
     no_bypass; // No bypass
end_if;
```

5.3 Bypass considering Pipeline

Whether 'process' or 'output' pipeline stage is busy or idle, is very important for the decision of bypass logic and must figured in. Bypass logic must take into account the status of 'process' and 'output' and figure out whether there is a packet with flow id equal to the one of the incoming packet, in these pipeline stages. If there is a 'hit', bypass logic will locate the most resent version of flow state according to the above (Table 5.3a).

Input Stage	Process Stage	Output Stage	Description
A1	A0	Busy or Idle	**Bypass L0**: A1 will find the latest version of the updateable part of flow state A in Global registers of A0 (GRSwitchNew = not GRSwitchLast)
A1	Busy processing packet of another FID	A0	**Bypass L1**: A1 will find the latest version of the updateable part of flow state A in Global registers of A0 (GRSwitchNew = GRSwitchLast)
A1	Idle	A0	**Bypass L0**: A1 will find the latest version of the updateable part of flow state A in Global registers of A0 (GRSwitchNew = not GRSwitchLast)
A1	Idle	Idle	**No bypass**: because process and output pipeline stages are idle and therefore the last version of flow state A is written back to RWR or to Control RAM. (Possible case of RWR bypass through RWR write buffers, if state A has not been written to Control RAM by the time RWR reads it from Control RAM)

Table 5.3a: bypass considering Pipeline

The full algorithm that detects bypass cases according to the status of pipeline is presented next:

```
If (current_flowid = last_flowid and (process = ”busy” or output =
“busy”)) then     //bypass L0
      bypass_L0;
elsif(current_flowid = before_last_flowid and process = ”busy” and
output = ”busy”) then
//    bypass L1
      bypass_L1;
else
// No bypass
      no_bypass;
end_if;
```

Chapter 6: RPG Microarchitecture

The block diagram of the RPG is shown (as part of the block diagram of PPE) in Figure 6a.

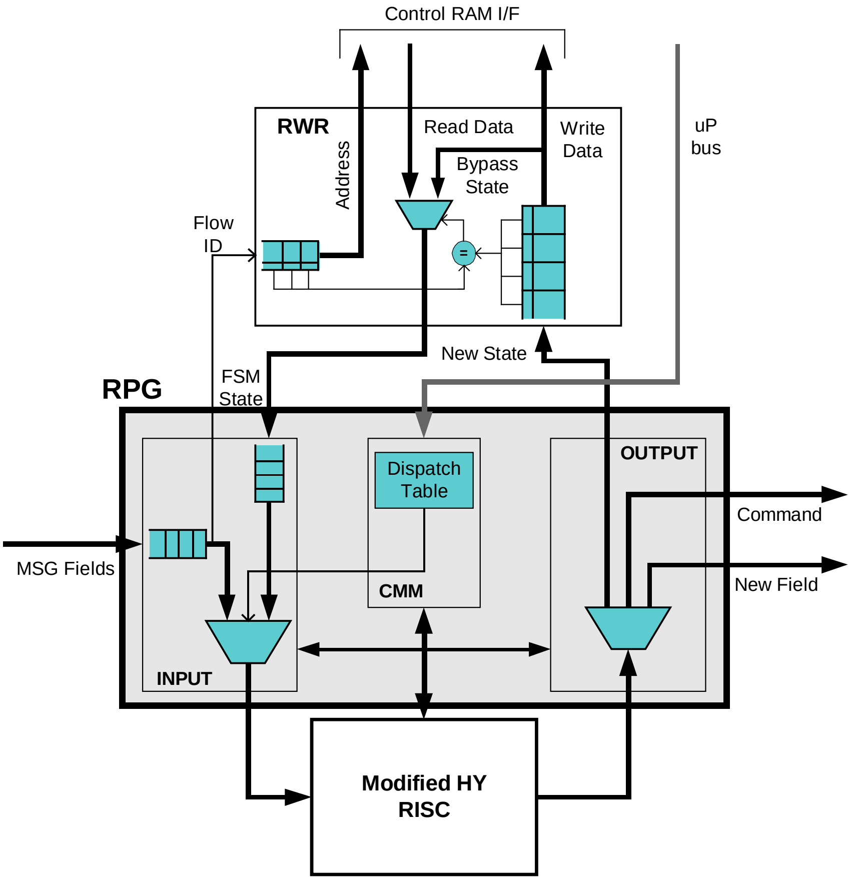


Figure 6a: RPG Block Diagram

RPM Glue Logic (RPG) consists of three units: input module, output module and Control and Monitoring Module (CMM). Input module transfers packets into MHY's register file for process. Output module transfers out process results. CMModule is responsible of the initialization of Dispatch Table, MHY internal RAM, status and result registers.

6.1 Data path

Input Module

Input module transfers all the necessary data needed for a packet process into MHY's register file. It transfers:

- Incoming packet fields from field extractor into RF,
- Dispatch PC from dispatch Table. MHY needs dispatch PC to dispatch to the correct routine for packet processing,

- Flow state information from RWR into RF.

Input uses packet flow id to request state from RWR. This request must take place as soon as possible in order to reduce the time of RWR's response. For this purpose, a module that detects packet FlowIDs is adapted to the input of field FIFO, it is quite simple because flow id information is always in the first word of every packet. The same module is responsible for internal packet fragmentation. In case packet field information doesn't fit in RF, packet is fragmented into more than one segment, which will be processed individually.

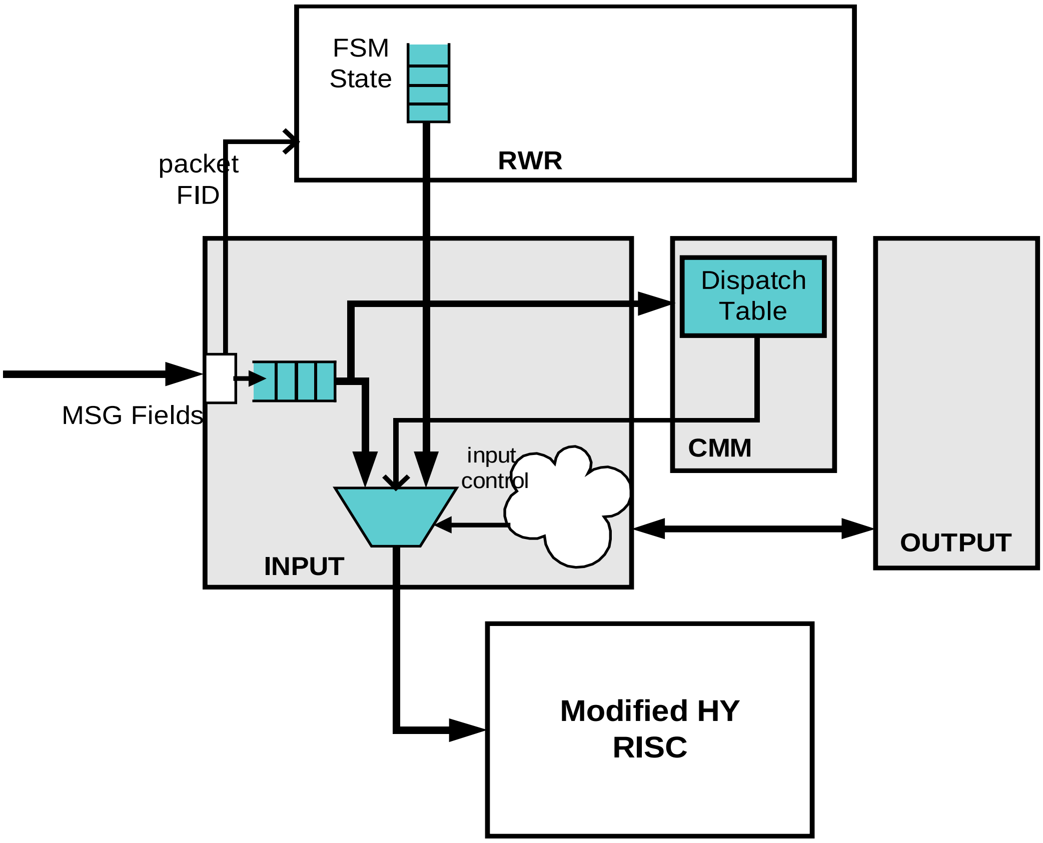


Figure 6.1a: input module datapath

Output Module

Output module transfers process results from MHY's RF or internal RAM to FMO and RWR. It transfers:

- Packet fields and commands from register file or the internal RAM of Hyperstone to FMO.
- Updated flow state information from RF to RWR.

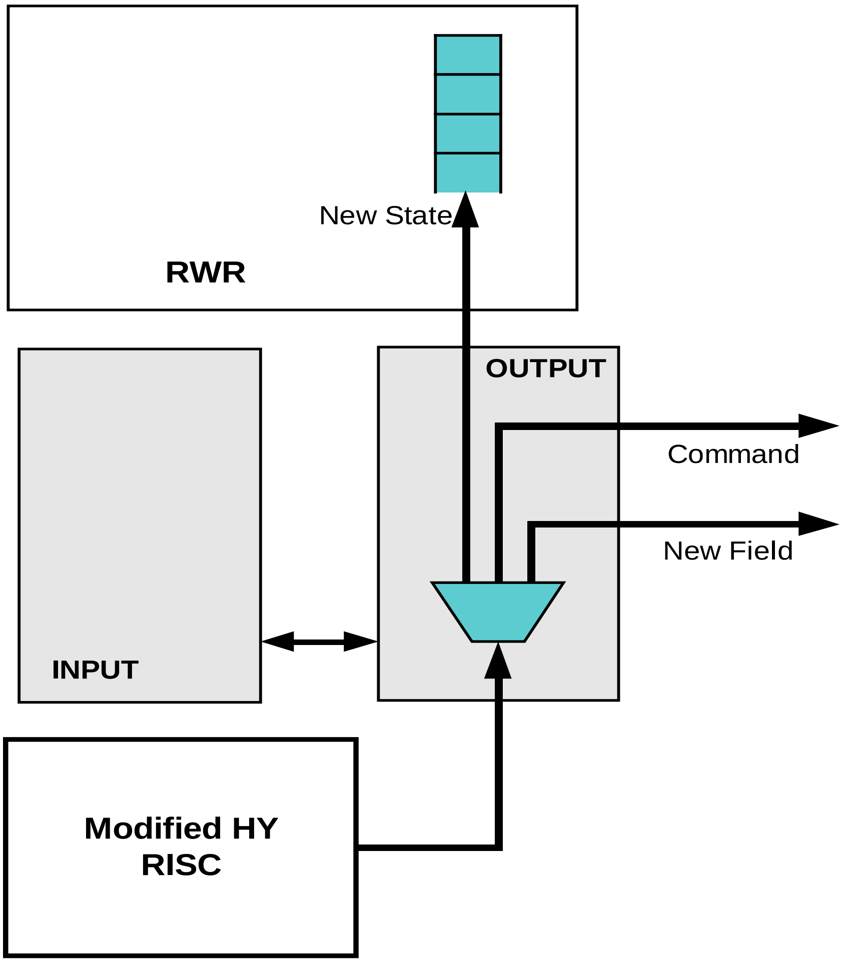


Figure 6.1a: input module datapath

6.2 Control path

Input Module

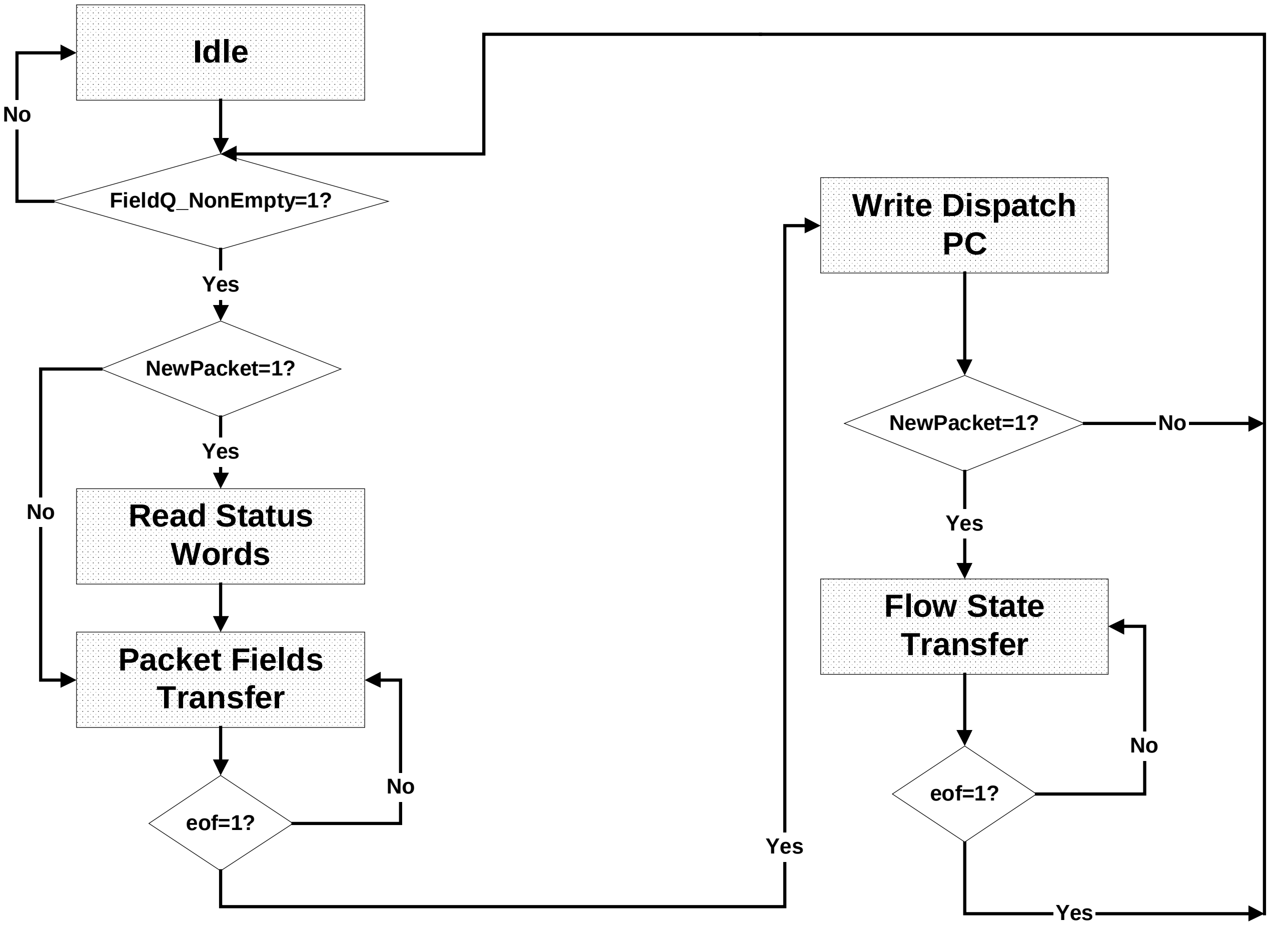


Figure 6.1a: input FSM

Input FSM (figure 6.1a) is designed in order not to waste any clock cycles during packet transfer. For example if input FSM needs to read and write 10 32-bit words, this transfer will last 11c.c.

Input FSM Idle

The input FSM is in this state when there is no new packet to transfer to the register file of the CPU for processing.

Read Status Words

The FSM goes to this state when the previous state was Idle or when the previous packet transfer has finished, there is new data in input Field FIFO and this data is the beginning of a new Packet. It lasts three c.c.

This state does the following actions:

1. Reads first word of the input Field FIFO (Flow id). Enables Bypass module that detects whether there is a bypass. Bypass module needs packet flow ID to check for bypass.
2. Reads second word of the input Field FIFO (Protocol Type)
3. Reads Third word of the input Field FIFO (Message Type)
4. Reads Dispatch Table and uses as an address Protocol and Message Type.

Bypass (detector) module

It takes as input the flow id of the packet and it detects data hazards and the kind of bypass (Bypass L0, Bypass L1) that is needed. This happens by checking whether current packet flow ID is equal with the previous one or the one before previous, it also figures out which local and global part of the RF will be used to current transfer.

Packet Field Transfer

1. It checks whether can transfer the next word:
    - Checks if input field FIFO is not empty.
    - If address comparator "allows" to write this word in the specific address (see § 6.3) of the RF (comparator compares the write address with the read address of output module).
2. If it's ok to write to the RF:
    - It reads the Header Field FIFO
    - Writes field word in RF (in the calculated address)
3. The state completes when there is an End of Segment flag (indicates that the transfer of segment or packet fields has been completed).

Write Dispatch PC

It starts after Packet Field Transfer is complete and it lasts one clock cycle.

It writes to a specific RF address (#66) the Dispatch PC, which Read Status Words state has already asked for.

Flow State Transfer

Starts right after the Write Dispatch Table Data only when FSM transfers the forst segment of a packet.

1. It checks whether can transfer the next word:
    - Checks if State FIFO is not empty.
    - If address comparator "allows" to write this word (see § 6.3) in the specific address of the RF (comparator compares the write address with the read address of the output part of glue logic –if it exists).

- Checks whether the number of incoming state words is greater than the defined flow state word number. If it is the process stops and the rest of the Data remaining in the State Fifo is cleared off. Internal Control CPU is informed for the error occurred.
- Checks whether there is bypass, In case of bypass it won't overwrite useful information.

2. If it's ok to write to the RF:
   - It reads the State FIFO.
   - It increases a counter that helps the configuration of the RF address every time. (The counter is the same counter that counts the Header Field address and is initialised every time is needed depending on its use).
   - Writes state word in RF (in the calculated address)
3. State completes when there is an End of State flag (indicates that the transfer of flow state information has been completed).

The last state of the Input FSM depending on the circumstances can be the Write Dispatch Table Data or Write State, but the Transfer of the next Packet can not start even thought there is data available, if CPU has not accepted last incoming packet (StartAck = 1, § 4.1).

The order of FSM's states is the one mentioned above in order not to waste any clock cycles waiting for data. During the first state "Read Status Words", FSM reads Dispatch Table and detects data hazards. After "Read Status Words" state FSM transfers packet fields, "Packet Field Transfer" state**.** By the time "Packet Field Transfer" state completes FSM has read Dispatch Table and can write Dispatch PC to register #66, "Write Dispatch PC" state. Finally, at the last state "Flow State Transfer", FSM has already detected any eventual bypass case and hopefully RWR would have read flow state from Control RAM. A packet may have more than one segment. The above sequence refers to the transfer of packets that have only one segment, or to the transfer of the first segment of a packet. In case of a multiple fragmented packet, the transfer of the segments, that are not the first one, includes only "Packet Field Transfer" and "Write Dispatch PC" state. Flow state information has already been placed in RF during the first segment transfer.

Output Module

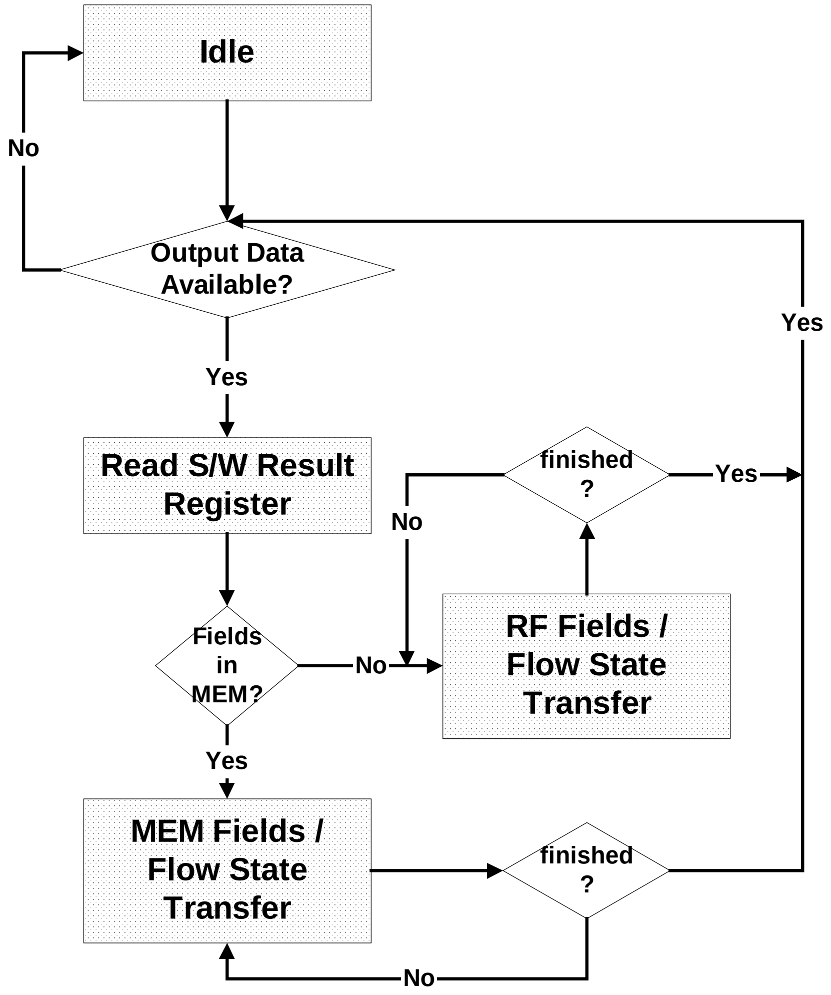


Figure 6.1b: output FSM

Output FSM Idle

Output FSM is in this state when there is not any processed packet to transfer out from MHY.

Read S/W Result Register

When a packet process completes and Output FSM is idle, a new transfer starts and the software result register (#67) is been read.

MEM Fields/ Flow State Transfer

According to software result register modified packet(s) may be in MHY's internal memory. In this case packet fields are transferred out from internal memory to FMO and in parallel updated flow state is transferred out from register file to RWR (there are two different port in RPG/MHY interface, register file port and internal memory port- see § 4.1).

RF Fields/ Flow State Transfer

According to software result register modified packet(s) may be in MHY’s register file. In this case packet fields are transferred out from register file to FMO and after that updated flow state is transferred out from register file to RWR. In case that for a number of clock cycles -during filed transfer- FSM does not read from register file packet fields, it reads and transfers flow state. Finally, sometimes flow state must be transferred first (see § 6.3 Read and Process), and then FSM gives immediately priority to flow state over packet fields transfer.

Output FSM is responsible of transferring out the Modified packet fields and the updated flow state after every process. A process may produce more than one packet. When the process of a packet finishes, output FSM reads a software result register (#67). This register indicates:

- Whether modified fields are in RF or in internal memory of MHY,
- The number of field words that are going to be transferred,
- Whether there is updated flow state,
- Whether output FSM will make or read from RF a command and sent it to FMO,
- Whether process has produced more than one packet.

The software result registers format (one per produced packet) is as follows:

Bit #	**Meaning**
31	more set of fields to follow after this one
30	fields reside in memory
29	SW_Ctrl_bits. A 0 indicates that the control fields are generated based on hardware values under software control, while a 1 indicates that the first X words in the register file/memory are the control words for fields and CommandQ.
28	Update Control Flow State Information (0 => do not update state)
27	Have command
26-22	5 bits: #words: for RF fields #regs to read, for fields in memory the first memory word specifies number of words.
21-0	Control Word creation bits, part of FMO command. If the fields are in memory (indicated by bit 30), these bits encode the *32-bit word* address where the actual control word and the packet fields reside in the MHY memory.

Table 6.1c: software result register

For every produced packet after a process, output FSM transfers out Modified fields and commands, but it transfers out only one updated flow state. If modified fields are in internal memory of MHY then FSM transfers fields from memory and flow state from RF in parallel. In case fields are in RF, it transfers first fields and after state (except of bypass L0 case see § 6.3)

Output FSM is designed so that won't be wasted any c.c. during modified packet transfer. During field transfer, read RF port may not be used for one or more clock cycles. During these number of clock cycles output FSM decides to transfer state.

6.3 Read, Write and Process to RF

RPG is able to read or write any register any time even if this register is used for process. This is of course a very critical issue. RPG needs mechanisms to ensure that output FSM reads before input FSM or process overwrites a register. Three differed possibilities can take place at the same part of registers, in the same time: read and write, write and process or read and process.

Read and Write

While a packet is in input and another one is in output stage input FSM will try to write packet fields and flow state at the same Local and Global part of register file, which output FSM uses to read the updated fields and state (figure 6.3a). This problem is solved by comparing the read and write addresses during the input of an incoming packet and stalling input FSM every time it tries to write to a register that is about to be read by output FSM, but is not read jet. Especially about flow state registers, the comparator compares read and write global addresses ignoring the bit that indicates the global part of RF (G0 or G1). This happens because is practically unnecessary and very expensive, regarding logic complexity, to figure out every bit of GRSwitch mask the registers that are about to be read by output and compare them with every write address.

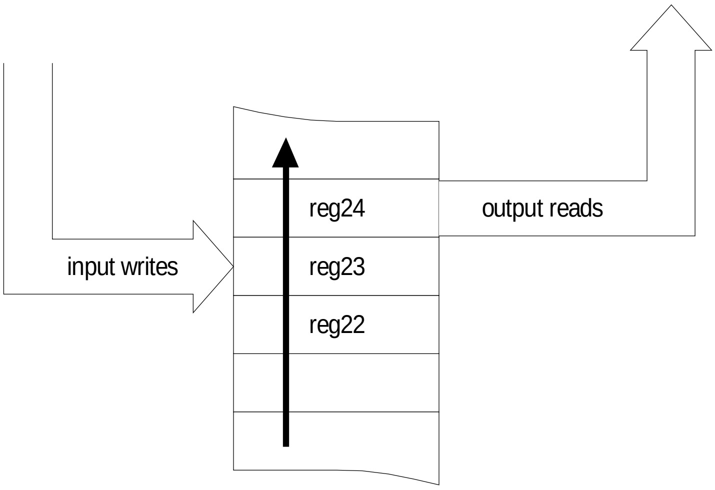


Figure 6.3a: Read and Write to the same part of RF. Input FSM cannot write to reg24 before output FSM read it

Write and Process

When a packet is in input and another one of the same flow is in process (bypass L0), input FSM points (using the GRSwitch mask) to the global part of process. In this case, there is not a problem because the common registers that contain the bypassed flow state are not going to be overwritten by input.

Read and Process

Finally, the most complicated case is detected at global registers when bypass L0 occurs by the time the second packet starts processing and the first packet of the same flow is in output (figures 5.1b and 5.2.a). In this case read and process take place at the same registers and there is a chance for process to modify registers that are about to be read by output. A way to avoid this is software to guaranty that the values of global registers won't be modified until output read them. On the other hand, hardware must try to detect this case. Output FSM will switch to a special mode, which gives priority to flow state transfer (§ 6.2). Thereby, these registers are going to be read as soon as possible. The restriction for software is not to start modifying global registers for 5 clock cycles after start processing. When input FSM detects a bypass L0 case during the transfer of second packet (A1), the first packet (A0) may be in process stage or in output stage. In the first case, by the time 'A0' start output the output FSM will decide to transfer flow state first. In second case, an interrupt will occur on output FSM and the updated flow state that has not transferred yet will be read immediately. Notice that the second case is very critical. If 'A0' is in output, output FSM may be in the state that decides the steps of transferring outgoing packet.

6.4 Synchronization between Input, Output and MHY modules

There is not a time slot in which every pipeline stage will start and finish a task. This makes synchronization between the three modules a complicated issue.

Input-Output

Incoming Packet information from input to output

While Input FSM transfers a packet to register file it writes some useful information about the incoming packet to a queue. This information includes:

- Source flow id, destination flow id,
- Local and global part of RF that was used during this transfer
- Protocol Type, Message Type, Subtype.

When this packet is processed, Output FSM reads this from this queue and uses the information according to the software result register.

Compare of read and write addresses

This issue is described in more detail on section 6.3. Additionally, can be mentioned that output FSM won’t permit input to write to any register until it figures out the registers it needs to read.

Input-MHY

During a packet transfer to RF input transfers fields, flow state, Dispatch PC and some additional information about the incoming packet. When the transfer is complete, it set start signal that indicates that there is a packet in RF available for process. Input FSM will wait until MHY set StartAck signal, which indicates that MHY started the process for this packet. Input can start new transfer only after StartAck is set (figure 6.4a).

Output-MHY

Output FSM figures out that there is a processed packet in MHY when CpuIdle signal, which indicates (when set) that MHY is idle, was zero and goes high. Output first of all reads the software result register to figure out what to do (see §6.2, Output Module) and then transfers process results.

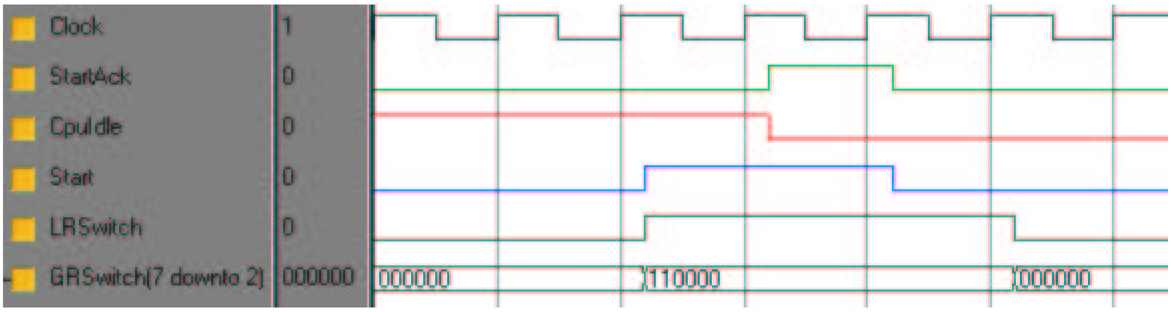


Figure 6.4a: Input-Output-MHY synchronization

6.5 Pipeline Stall Mechanism - Backpressure

There is not a top RPG FSM that controls the operation of input, process and output stage. Therefore, the pipeline stalls with a backpressure mechanism. Assuming that output transfers packet A out of RPG, MHY processes packet B and input transfers in RF packet C. When output stage stalls (output writes data to a FIFO that after a while gets full) and the transfer of A is not complete there are two possibilities:

1. Input FSM needs to write data about C in a register that Output FSM needs to read. In this case Input waits until Output reads this register. Input will stall until output starts transferring data again and read the particular register.

2. Input FSM manages to complete the transfer of packet C (there was not a register that input needed to write and Output wanted to read), but packet B, even if process has been completed, cannot be transferred out yet because output FSM has stuck during A transfer. In this case, (when a process is completed and output is stalled or busy transferring another packet) input FSM cannot set Start signal, which indicates to MHY that there is a packet available for process. Packet C can start processing only when Output FSM transfer out packet A and start transferring B.

Chapter 7: Performance Evaluation

RPG performance cannot be evaluated without looking at the performance of the relevant modules. Therefore, performance of FEX, FMO, process (MHY), RWR and Control RAM is very important too.

7.1 Metrics

To understand and evaluate the performance of PPE we model the operation analytically. First there is need to create equations, which calculate the clock cycles needed for a packet to pass through PPE module. The performance of the nearby modules (CRIM, FEX, FMO) must also be considered, because they affect PPE's performance.

Process Input & Output Delay

Process is one of the pipeline stages (input, process, output). If process delay is greater than input and output delay, then the bottleneck of PPE pipeline is process. Process delay depends on packet size, protocol code (packet protocol), Modified Hyperstone performance and process overheads (hardware/software interface i.e. extra data creation and picking over extra data needed for protocol process).

Input delay equation calculates the number of clock cycles needed for a packet transfer to MHY's register file (includes field, flow state and other information transfer, needed for packet process) is:

InputDelay = {4 + (3 + State) * NewPack + Fields + RWRreadReqDelay
+ FEXReadReqDelay} c.c.

Where:
State: number of State words
Field: number of packet fields' words
NewPack: it is '1' in case of the first segment of a packet, else is '0'
RWRreadReqDelay: number of c.c. that input waits for state although is ready to transfer state.

FEXReadReqDelay: number of c.c. that input waits for fields although is ready to transfer fields.

Output delay equation calculates the number of clock cycles needed for a processed packet transfer out (from MHY to FMO):

OutputDelay = {3+ UpState + [(ModField1 +1) +…+ (ModFieldn +1)] +
[2*S/Wcmd1 +…+ 2*S/Wcmdn] + RWRWriteReqDelay + FMOWriteReqDelay} c.c.

Where:

RWRWriteReqDelay: number of c.c. that output waits (full FIFO) although is ready to write state.

FMOWriteReqDelay: number of c.c. that output waits (full FIFO) although is ready to write field or commands.

UpState: the number of updated state words

ModField_n: the number of field words

S/Wcmd_n: is ‘1’ when software is producing commands for FMO command fifo, else is’0’

N= the number of packets produced by one process

Control RAM Performance

Control RAM response time effects RPG’s performance. As soon RPG reads and writes flow state information to Control RAM the better. PPE uses a specialized module that detects packets flow ids to request flow state from Control RAM (through RWR and CRIM) as soon as possible (§ 6.1 input module figure 6.1.a). This module reduces the response time of Control RAM and improves PPE performance. There are also read and write request fifos in RWR, which can store state for more than one flow ids.

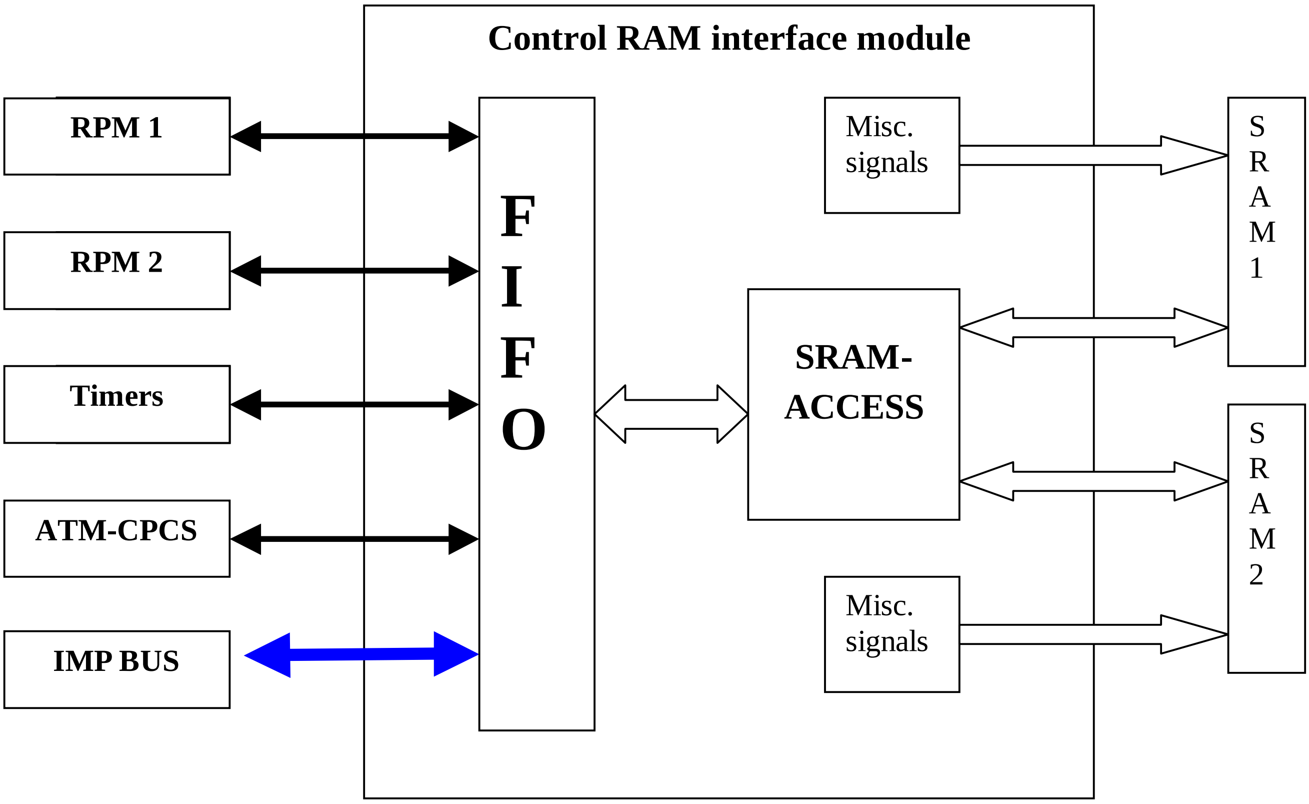


Figure 7.1a: Control RAM Interface Module (CRIM) block diagram

Control RAM Interface Module (CRIM, figure 7.1a) has five read/write ports (for of them have 64-bit wide data in & data out ports and one 32-bit) and only one 64-bit wide port to Control RAM. It performs at 200 MHz.

These are four proprietary Control RAM request interfaces which will access the Control RAM the most Additionally, there is also IMP bus port, which is 32-bit wide. The accesses to the Control RAM will be in chunks of 64-bit words (for the four ports) - maximum of eight words (i.e. maximum of 512 bits) in one shot (burst). On an interface, read and write requests can occur simultaneously. In such a case of simultaneous requests on the same interface, the write request is given a higher priority over the read request. One of the interface requests is serviced at a higher priority than the other three. This is needed for quick servicing of ATM CPCS module requests. Other three interface requests are served in a round-robin fashion (i.e. without any priority between them). Note that servicing of any interface request is never interrupted i.e. all memory words are accessed without any interruption even if there is a request from a higher priority request interfaces. In such a case, CRIM first completes the present request fully before servicing the high priority one.

FEX-FMO performance

Both FEX and FMO operate on 100MHz clock frequency, and the aggregate throughput varies between 3.2Gbps and 2.5Gbps. This is generally satisfactory. 2.5Gbps = 12.5bits/c.c.
and 3.2Gbps = 16 bits/c.c. . PPE would prefer a throughput near 32bits/c.c., but this would be worthy only if Control RAM performance accommodates PPE.

7.2 Performance Results

Following Table presents performance results of PPE. 100.000 packets fragmented in 139,701 segments passed through PPE and produced 159.915 segments. Control RAM response varies between 5 – 10 c.c. Supposed that FEX can satisfy PPE’s throughput and FMO’s FIFOs will never get full. Process duration varies between 20 to 40 c.c. There are 10 different flows and every packet randomly belongs to one of these flows. 20% of the packets produce 2 packets after process and the rest of them produce one. 80% of the packets are fragmented –randomly- in 1, 2 or 3 segments. Every packet contains 6 to 32 field words (32-bit). Every packet updates flow state. In case of a multiple segmented packet, updated flow state will be produced and updated only during the process of the last segment of the packet. Flow state contains 8 words (32-bit) and 4 of them are updateable.

Packet Segments	Number of Processes	Process c.c.	Total c.c.	Process c.c. / Total c.c.
139.701	139.701	4.191.030	5.960.358	70,315%

Table 7.2a: Performance Results of 100.000 packets through PPE

Similar results of 10.000 packets passing through PPE and process duration 20, 30, 40 and 50 clock cycles are presented next. The parameters of 100K packets are valid for these vectors of 10K packets too. The value of {Process c.c. / Total c.c.} can never become 100%, because PPE pipeline can never be prefect. Even if input and output stages would last less c.c. than process, there is always a chance to wait for fields (coming from FEX), flow state (coming from Control RAM) or to get FMO or RWR FIFO full. Besides, idle pipeline stages cannot be avoided.

Packet Segments	Number of Processes	Process duration	Process c.c.	Total c.c.	Process c.c. / Total c.c.
13.988	13.988	20	279.760	594.041	47,094%
13.988	13.988	30	419.640	590.051	71,119%
13.988	13.988	40	559.520	626.349	89,330%
13.988	13.988	50	699.400	742.174	94,236%

Table 7.2b: Performance Results of 10.000 packets through PPE (changing process duration)

The following figure shows the way process duration effects PPE performance.

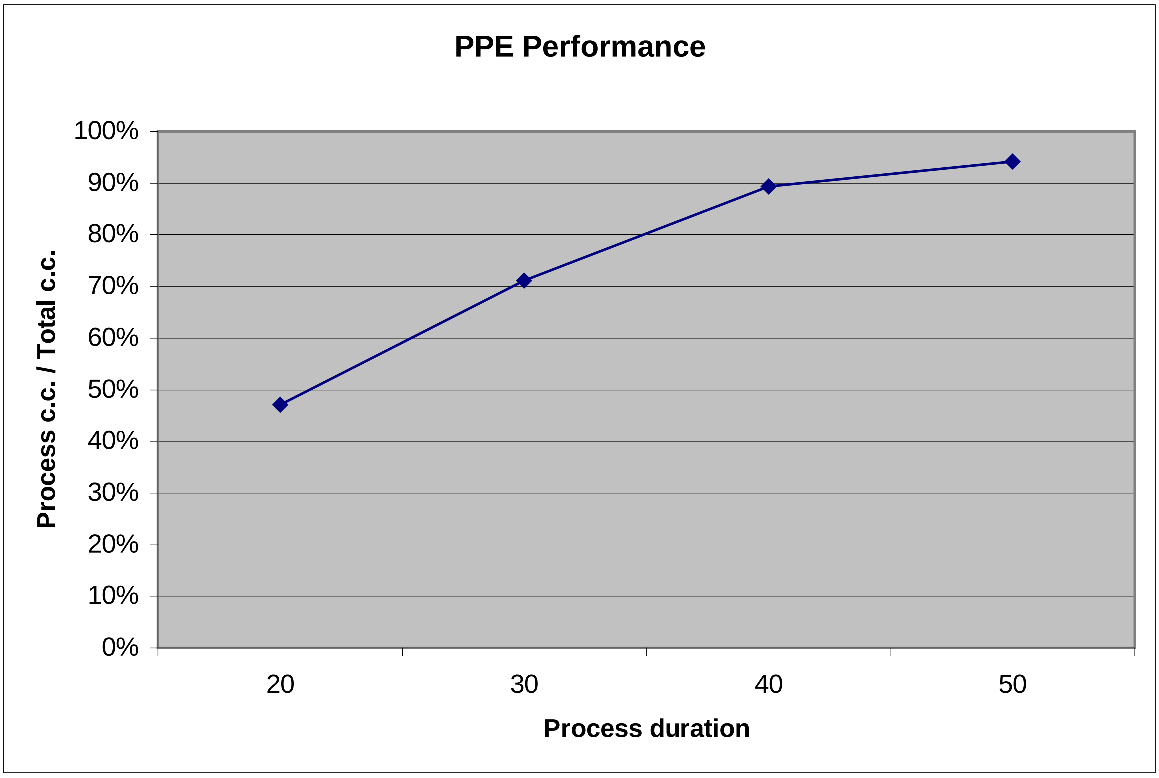


Table 7.2c: PPE Performance

7.3 Overheads

In Pro3 design hardware is chosen to handle the input and output of packets and additional information in MHY core. PPE is designed to support parallel input, output and process of packets. PPE's performance is fairly better than another solution, which would serially input, process and output packets. On the other hand, this solution has some overheads. Extra clock cycles are needed during protocol processing to handle software/hardware interface. Finally, except packet fields extra information has to be transferred in/out MHY register file during packet input and output (Dispatch PC, flow state, s/w result word).

Process Overhead

RPG solution creates some process overheads. Every protocol processing routine needs extra instructions (shifting and logic operations to analyze and edit packet data fields). Software needs extra instructions to handle software/hardware interface: read from RF information about every incoming packet (word number, Proto, Subtype…) and compose software result word for output module.

Extra data Transfer

There is also extra data transfer such as flow state, dispatch PC and s/w result word. This data is needed for every protocol process. Therefore, extra clock cycles are spent by Input and Output FSM to transfer this data.

Chapter 8: Implementation and Verification

PPE is a complicated module and includes complicated FSMs. There is not a top-level FSM to controls sub modules; therefore, their synchronization is a critical issue. The need of exhaustive testing is obvious. The key in debugging this design was the systematic test using software. Input files, that include different possible cases of incoming packets, were created using software. An input feeder reads input file and gives to PPE module incoming packets.

Monitors (figure 8.1a) record data that end up at output ports or pass through critical blocks of the design. At the beginning, these tests were performed with the use of mock modules that substitute real RPM modules and have similar response with the real ones (figure 8.1b). After that, the same tests were performed with the real modules of RPM (FEX, MHY, FMO, Control RAM – figure 8.1c) and finally, there were chip level tests. Software evaluated processes' results of every test.

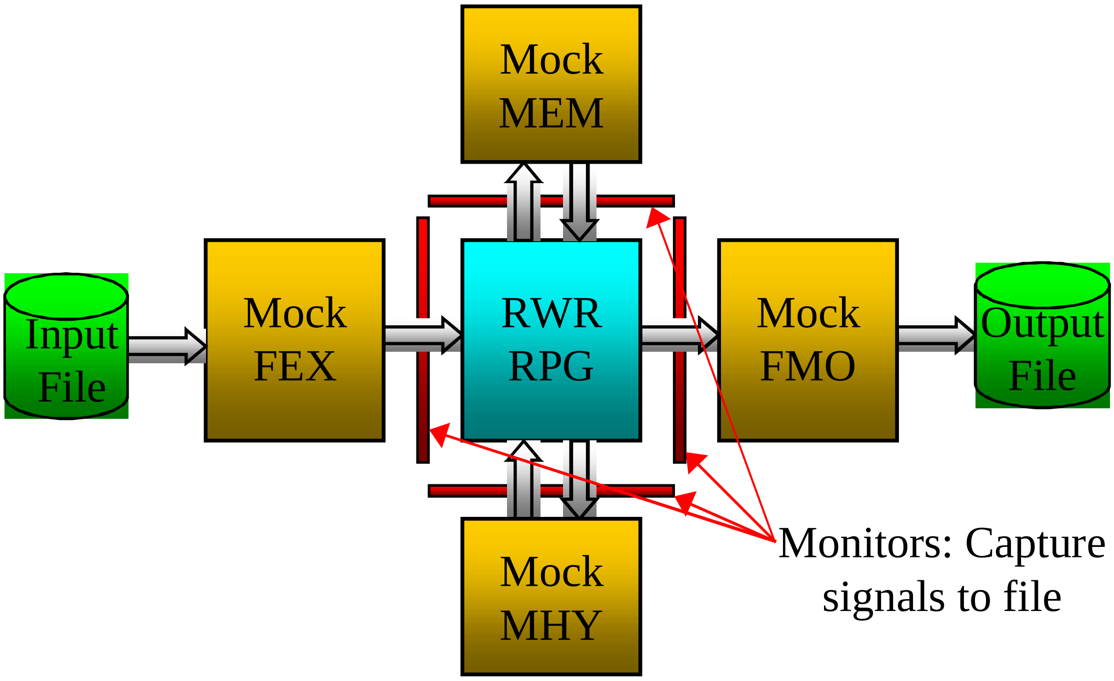


Figure 8.1a: Test-bench using mock modules and monitors

8.1 Techniques

Mock Modules

Mock modules are very important for debug (figure 8.1a). During the first steps of verification, nearby RPM modules were not completed (neither PPE). Therefore, was very important to create modules, which would substitute the nearby actual ones (FEX, FMO, Control RAM).

These modules have similar response with the real modules and a “dummy” functionality - sometimes they just have reduced functionality:

i.e.
Mock Control RAM and mock Control RAM interface Module:

- Has only one read and write port instead of five (for the needs of one PPE module), and
- Can store flow state for 256 different flows, instead of 500K.

Another example of mock module is mock CPU, which has a register file and an internal memory just like Modified Hyperstone. After every “dummy” process, mock CPU needs to give to output FSM a software result word, which specifies what output FSM, will do. Mock CPU uses as software result word a specific word, which input FSM puts in register file (figure 8.1b). “Dummy” process also updates (ADDI state_word(i),1) flow state, this is necessary for bypass debug.

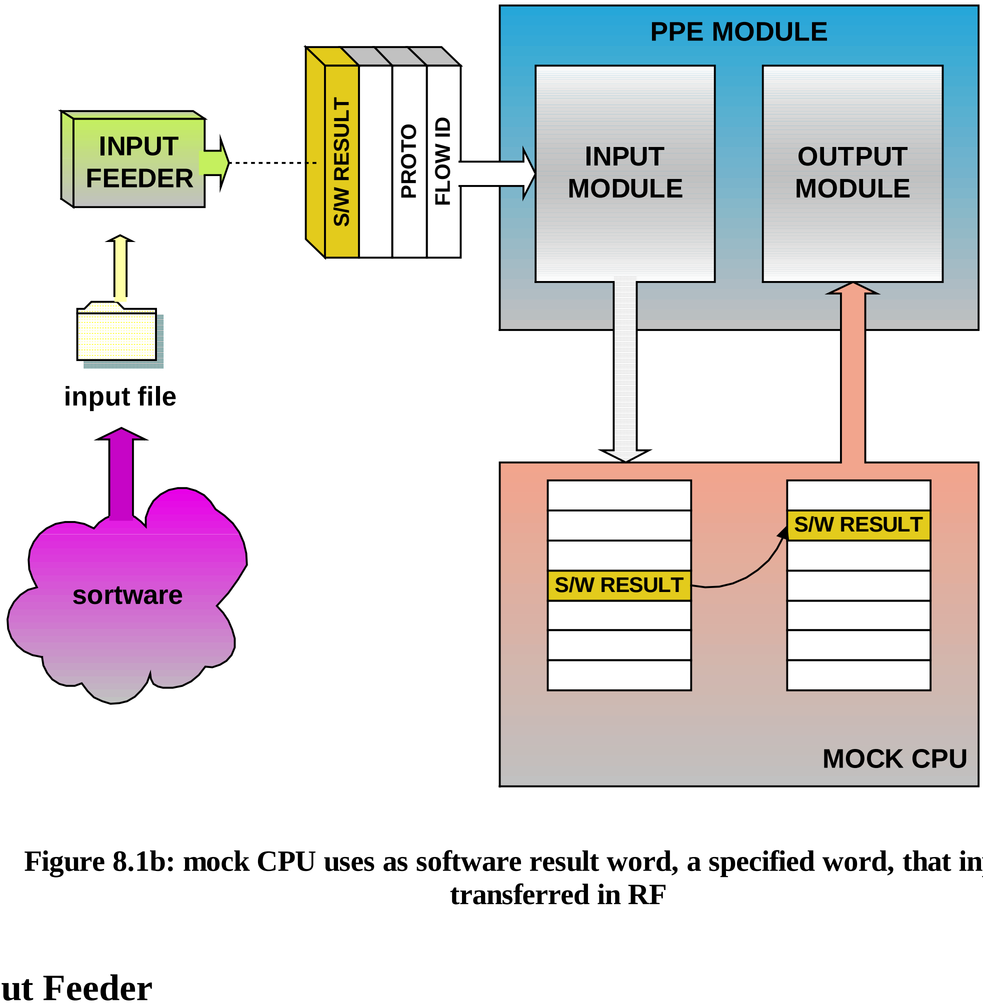


Figure 8.1b: mock CPU uses as software result word, a specified word, that input FSM transferred in RF

Input Feeder

For stand-alone tests, PPE needs a module to substitute FEX. This module is input feeder, it reads an input file and writes packet fields in PPE input FIFO (figure 8.1b). Input files are created using software. Modifying software parameters can create different scenarios of incoming packets with specific flow id sequences or process results (process result is specified by software result word, which is –only for “dummy” processes- part of input file). Input feeder reads input file and enqueues packet fields to input field FIFO. In case input FIFO is full, input feeder waits for a specific number of c.c. and then tries again to enqueue data. The number of clock cycles that input feeder waits for writing data, can be different for every packet and is defined in input file.

The format of an input file that is produced by software and looks like:

```
# Packet number 1
 140 ns
 0000000005
 0000002ae6
 0000000442
 0000001bcd
 00000044bb
 000000190f
 00000059a4
 0000007996
 001b800000
 0000001524
 00000036a1
 1000000000
```

Monitors

Systematic tests require monitors (figure 8.1a, 8.1c). Monitors are able to capture signals and store their values. They are HDL modules that monitor a particular interface or module and record specific events into text files (monitor files). Monitor files are evaluated by software and every bug of the design can be tracked. The monitor modules could either monitor an external interface or an internal interface or an internal HDL module. For example, a monitor can store data that is written or read to/from a FIFO and the time of every enqueue or dequeue. Two monitors are foreseen, one for the interface between FMO and PPE (modified fields) and one for the interface between PPE and RWR (updated state).

The format of the monitor file that is produced by the monitor between PPE and FMO (records modified packet fields), and looks like:

```
# EOPacket No: 1
00000464B
000005C29
000000E96
0BA000000
000004839
```

```
000002C18
000006F32
0000021D8
200000000
# EOPacket No: 2
```

The format of the monitor file that is produced by the monitor between PPE and RWR (records updated state), and looks like:

```
# New state packet No: 1
# Flow ID: 000005
# offset          state_data
    0               05000000
    0               05010000
    0               05020000
    0               05030000
#====================================
```

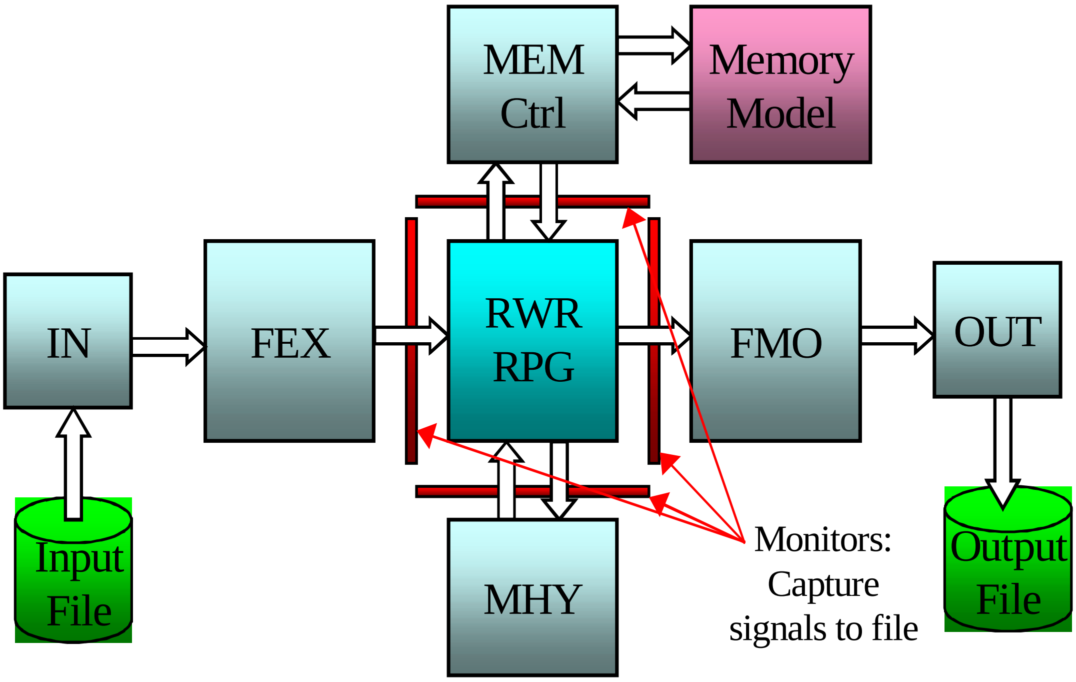


Figure 8.1c: Test-bench using real RPM modules and monitors

Verification using Software

Either using mock modules or real RPM modules (figures 8.1a, 8.1c), software verifies the results of every run. Either using mock CPU or Hyperstone, there is a "dummy" packet process as shown in figure 8.1b. Dummy process does the following things:

- Increments (“dummy” update) flow state words by one.
- Gives incoming packet fields to output FSM as modified packet(s), software result word defines details (there is not any field modification).

First of all, software verifies whether every incoming packet field, which should result in output, ends up in output monitor file according to S/W result word (§ 6.2 Output Module).

About flow state verification, state word format (created only for debug reasons) is shown in figure 8.1d. This state format is also presented in the example of “updated state” monitor file above.

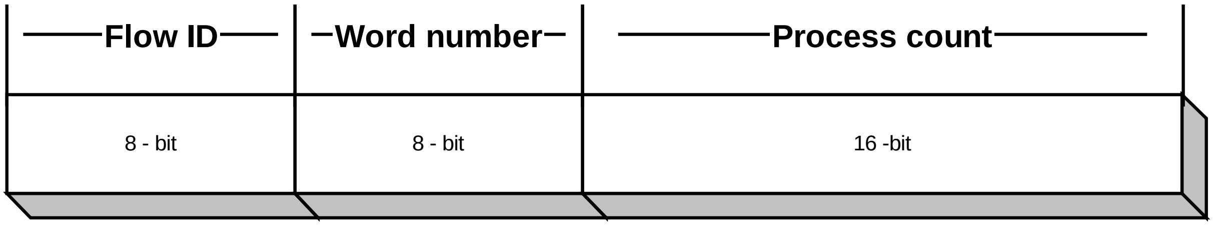


Figure 8.1d: Flow State Word format (only for debug)

1. Flow Id is needed to check whether state belongs to the correct flow id.
2. Word number is the number of every state word, i.e. if state contains six words, every word will have a different word number (1, 2, 3…).
3. “Process count” part is the part that “dummy” process increases after every process. (“dummy” update)

Software verifies whether updated flow state is the correct one. It checks for every flow state word whether flow id, word number and “process count” values are correct. Bypass correctness can be verified by checking whether last process counter value of every flow id is equal to the number of incoming packets of the same flow. When they are equal, means that flow state has been updated correctly during *every* packet process. Whether they are not equal, software tracks the bugs by figuring out which “dummy” packet process did not update its flow state.

Chip-level Verification

The verification methodology followed in chip-level simulations is based on the following elements:

- Drivers: HDL modules that drive particular interfaces of the PRO3 chip (PPE’s input feeder is a Driver). In general these modules read values from a text file and drive those values on the appropriate pins of the interface. We call these text files command files. Two driver modules are foreseen, one for the input interface and one for the external CPU interface.

The format of the command file to be used by the input-interface driver (input_drv) looks like:

```
# this is a comment line
packet_start
    # packet header
3456 7879  abcd
# packet data
aaaa bbbb cccc 1234 5678
1234 12aa
packet_end
#
    idle 10 # wait for 10 clock cycles
packet_start
    222
    packet_end
```

While the format of the command file to be used by the external CPU interface driver (ecpu_drv) looks like:

```
# this is a comment line
read  34ABC
write 34ABC 1234_567B
idle 10 # wait for 10 clock cycle
write 34ABC 1234567C
```

- Monitors: monitor modules were also used in chip-level simulations to monitor external or internal interfaces or internal HDL modules.

  The format of the monitor file that is produced by the output-interface monitor (output_mon) and looks like:

```
Time Data       EOP
--------------------------------------
120  12345678   -
140  789A       1
220  11223344   -
240  22334455   -
260  AA         1
```

  The format of the monitor file that is produced by the external-CPU-interface monitor (ecpu_mon) and looks like:

```
Time Operation  Addr       Data
----------------------------------------------------------
------
123  RD         34ABC      1234567A
170  WR         34ABC      1234567B
190  WR         34ABC      1234567C
```

- Testbench. This is a top-level VHDL testbench that instantiates the PRO3 chip along with all external devices (RAMs and CAM), drivers and monitors. The same testbench will be used for the validation of all models of the PRO3 chip. At first the testbench will be used for verifying the behavioral/RTL model, then the synthesized gate-level model and finally the back-annotated gate-level model.
- Pre-load files. In general it is required that some internal or external memories need to be loaded in order for the system to start functioning (i.e. field extraction microcode memory, MHY internal RAM). Also this may be required in order to set the system to a particular state that it would otherwise require a lot of simulation time to reach. The loading of these memories could be achieved through normal operation, usually through the external CPU interface. However, in order to save time it is preferred that incorporated memories have the ability to initialize themselves through the use of pre-load text files.
- Test vectors. The PRO3 chip will be verified by running several test scenarios against it. A particular test scenario is described by the command and pre-load files used to achieve this scenario. This set of files is referred to as test vector. A number of test vectors need to be identified in order to achieve full coverage of the functionality of PRO3. A way to produce a complete set of test vectors would be to identify the functionality of each module and write vectors in order to check all this functionality.

  Since it would be very time consuming to run the whole test vector set against the gate-level model of PRO3, it has been identified a number of indicative vectors to run against it. In order to have a neat verification environment each test vector and all input and output files associated with are kept in distinct subdirectories. After running a test vector the monitor files can be used to verify whether the behaviour is correct or not.

The above elements define the actual verification methodology followed in chip-level simulations.

8.2 Fallback module

In case the original PPE path does not work properly there is another datapath (figure 8.2b) to make sure that packets, even not processed, will be transferred from FEX to FMO (for chip debug). An Internal CPU (Control CPU) signal activates this “fall-back” path. In this case Fields are been transferred from Input Field Fifo to Output Modified Field Fifo. The Command FIFO of FMO is filled with a default FMO command and a default IBUS control word (64 bits) every time a new packet or another segment of a packet (followed by an end of segment flag) arises.

FEX packets and FMO packets do not have the same format (Table 8.2a), therefore, fallback module must change every incoming packet before it will be send to FMO:

Pack wrds	Incoming Packet Format (From FEX)	Outgoing Packet Format (To FMO)
1st word	Source FID	Proto Type
2nd word	Proto Type	Message Type
3rd word	Message Type	Sub-Type
. . .	(Fields)	(Fields)
Last word	End of packet/end of segment word (eop/eos)	End of packet/end of segment word (eop/eos)

Table 8.2a: format of incoming and outgoing packets

- The Source FID that comes from FEX can be used to fill the Source Flow ID field of IBUS data.
- The FEX Proto Type can be used as the FMO Proto Type.
- The FEX Message Type can be used as the FMO Message Type.
- The FMO Sub-Type is the number of current packet part and can be calculated.
- The rest of the FEX Fields are been transferred to FMO Modified Field Fifo.
- The FEX end of packet/end of segment (eop/eos) word can be used as the FMO eop/eos word.

In this mode PPE does not process the FEX packets. Therefore, there is no need to update flow state information and to read or write from/to Control RAM.

When this mode is activated the original PPE datapath is not used and “fall-back ” mode is responsible for reading from Input Field Fifo and writing to FMO FIFO. Reading and writing State from/to RWR’s FIFOs is disabled.
Whether Internal CPU (control CPU) decides to change mode then RPM module should be reset.

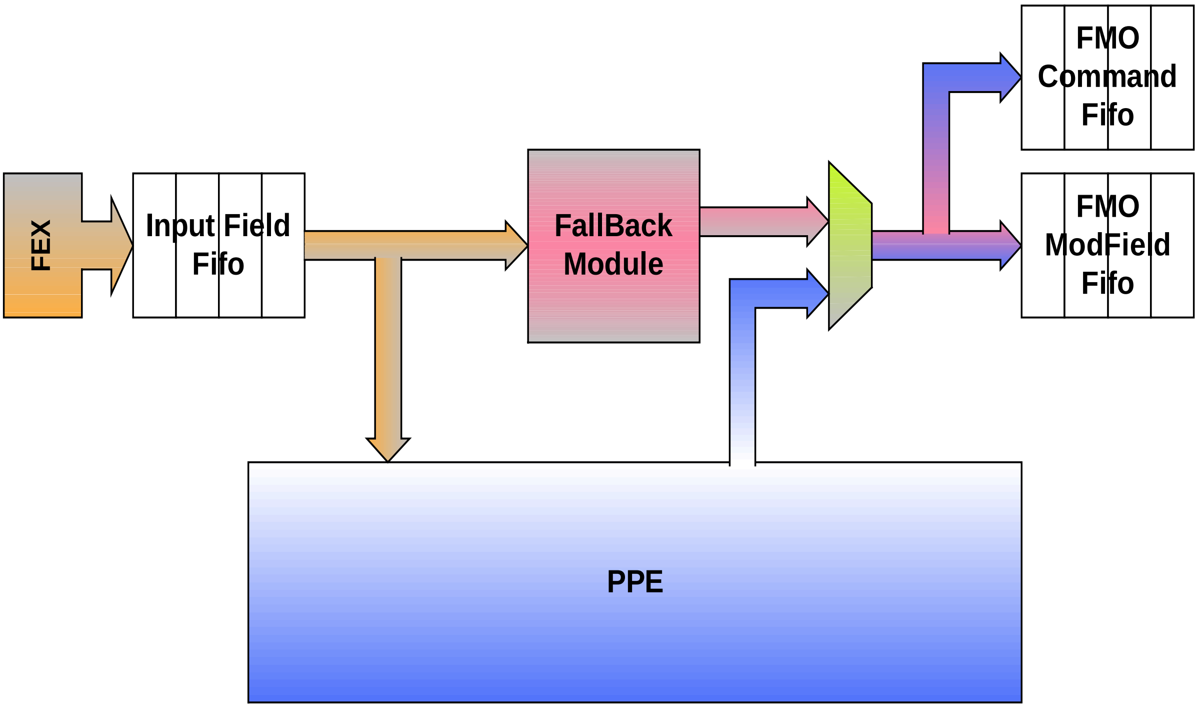


Figure 8.2b: fallback module in PPE block diagram

8.3 Synthesis-Gatelevel Simulation

The Synopsys Design Compiler (DC) was used for the synthesis process. Because of the high processing and memory requirements of a monolithic synthesis run for the entire chip, it was decided that a mixed top-down / bottom-up approach will be followed, as described in the following.

First of all, each major PRO3 block was synthesized with a simple top-down run of DC. PPE synthesis yielded a netlist, together with appropriate area and timing reports. For this first run, the environment and timing constraints applied to PPE were approximate. Environment constraints consist of output capacitive loads and input drive strengths, while timing constraints define the external delays of inputs and outputs. The constraints arising from on-chip block-to-block interfaces were not known at this point, and were approximated with typical values for on-chip interconnections. Therefore, the initial netlists and reports generated were not entirely accurate.

After stand-alone synthesis of each block, next step is the generation and characterization of the full-chip netlist. The VHDL source code of the PRO3 top level was compiled and linked to the netlists of the individual blocks. Then, a timing characterization was carried out, that records the timing across the major PRO3 blocks (and PPE), as it results from the current block netlists. Some timing violations came up at this point, since for some signals, the sum of the delays in the source and sink blocks, which were synthesized independently of each other, were larger than the clock cycle. The timing information, together with actual capacitive loading and drive strength information, was recorded and output as a DC command script for each PRO3 block. Then, inter-block environment and timing constraints were more accurately recorded. This concludes the first pass.

The second pass started by re-synthesizing all the major blocks individually. This time, the real timing constraints from the automatically generated scripts are used. The netlists and reports generated in this second pass were more accurate. The full-chip netlist was generated again and another characterization was carried out. This time, most of the timing violations were cleared, although some new violations came up and fixed.

Since these are only statistical estimates, the timing resulting from this process is not entirely accurate. Therefore, we have applied a *10% margin* in all timing; during synthesis, the system clock and memory clock periods are assumed to be 4.5ns and 6.75ns respectively. Finally, clock skew is 0,3 ns.

The design is mapped to the UMCL18U250 0.18μm standard cell library from UMC. In addition, the UMCL18U300 staggered pad library is used for the PRO3 pad ring. For each major block and the PRO3 top level, netlists are generated in VHDL format for gate-level simulation and Verilog for signoff to Europractice, as well as the internal Synopsys db format. All netlists are accompanied by area and timing reports. Standard Delay Format (SDF) files are also generated during synthesis, to be used in timing simulation. These SDF files contain delay information due to net fanout and expected wire loading.

Gatelevel simulations (with and without SDF files) were performed after synthesis and had the same (correct) process results like the functional tests. These tests (including functional ones) were performed (using Modelsim 5.5e) on testbenches, which included mock modules and real RPM modules. Some problems occurred during gatelevel tests: initialization problems and different response on memory models and latches.

- **Initialization Problems.** In functional simulations signals that are not initialized, take as initial value zero. However, in gatelevel simulations these signals become undefined and cause functional problems.
- **Memory Models:** In functional simulations data read from memory models were latched, however, in gatelevel simulations were not. Therefore, was necessary to put latches in every memory output.
- **Latches.** Latches did not function properly in gatelevel simulation. Therefore, was decided to replace latches with modules that included registers and multiplexers 2x1 (figure 8.3a).

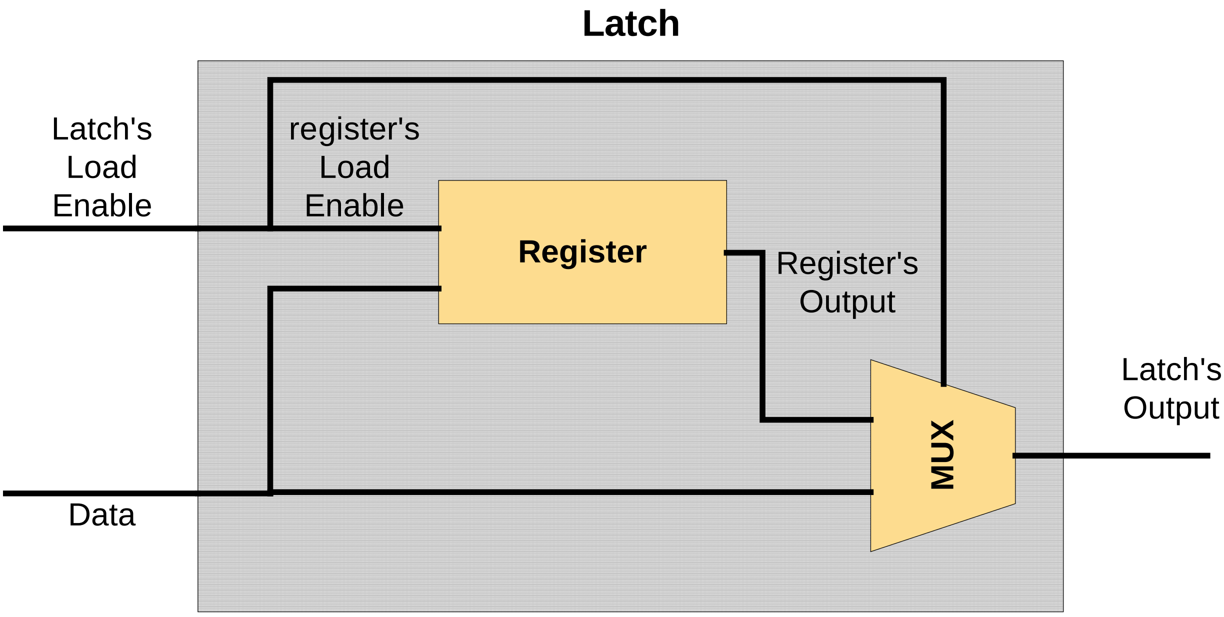


Figure 8.3a: Latch using Register and Multiplexer 2x1.

Pro3 target frequency (200 MHz) has been accomplished. There were also some synthesis problems: false paths, timing loops and unnecessary delay of data paths. All of them were detected and corrected.

- **False paths** are datapaths that do not actually used. DC may locate violations in false paths. These paths must be located in VHDL design and fixed.
- **Timing Loops** are loops made by signals that affect each other asynchronously (i.e. A<= B and C; B<= D or A;). These loops must be detected and removed. This problem is more difficult when timing loops include signals, which are in more than one module.
- **Unnecessary Delay of data paths.** This happens when datapath is unnecessarily delayed by control and causes violation problems. In these cases separation of datapath and control reduces delay and hopefully fixes violation. However, sometimes is necessary to cut off a datapath, which causes violation, using registers.

The following table shows the difference between functional and gatelevel simulation speed. Simulations ran at a PC with a 1.2 GHz processor and 512 MByte DDR.

	Functional PPE	Functional Pro3	Gatelevel PPE	Gatelevel PPE with SDF
Clock cycles/minute	15.359	1.088	3.109	2.940
Comparison Results	14,1 * a	a	2,9 * a	2,7 * a

Table 8.3b: simulation speed

Synthesis results about PPE area, number of cells, ports and nets are shown in table 8.3c. In area results, net area is not contained.

Modules	Port#	Net#	Cell#	Combinational Area	NonCombinational Area	Total Area
RPG	**538**	**6,105**	**5,724**	**74,351**	**539,772**	**614,123**
RWR	**353**	**6,115**	**5,531**	**77,453**	**403,944**	**481,398**
PPE	**541**	**10,870**	**10,028**	**136,974**	**933,235**	**1,070,206**

Table 8.3c: synthesis results

Chapter 9: Conclusions

This work includes design and implementation of an input/output sub module that inputs data into the RF of a RISC processor, allows simultaneous packet process and output of processed packet.

The key of testing and debugging this design was the systematic tests using software. The complexity of PPE predicated systematic testing. It was practically impossible to substantially debug PPE module without generating large input files and check automatically (i.e. using software) whether output files were the expected ones. Large input files give the facility to check the response of PPE system for thousands of input packets. Monitors give the opportunity to store in files the results of packet processing or internal signals' values and using software can be checked whether they are correct.

Using the above methodology, the next steps were followed for the debug of design:

- First, run stand-alone tests using mock modules on test benches, which replaced the nearby Pro3 modules.
- Then, integrate the PPE with the actual nearby modules of RPM (FEX, MHY, FMO and Control RAM).
- After synthesis, perform stand-alone gatelevel simulations (using mock modules)
- Then perform simulations including in testbench the entire Pro3 design.
- Finally, perform gatelevel simulations, using actual nearby modules.

Design analysis is also very important for the success of a design. Whenever there was insufficient analysis, problems occurred during implementation. For example, RWR module came up behind schedule when interfaces had already been frozen. Therefore, design and implementation of RWR was much more difficult and complicated than it would be if RWR had been included in PPE analysis from the beginning. Figure 9a shows the actual PPE design and figure 9b shows the one that should be designed. In case of Figure 9b when RWR would detect an external bypass case (RWR bypass), it would short circuit immediately the up-to-date state. On the other hand, at Figure 9a block diagram, on bypass case, RWR must read Control RAM first and give to RPG flow state information from control RAM *and* write buffers (bypassed state). This means that the block diagram of figure 9a is more complicated than the one of figure 9b.

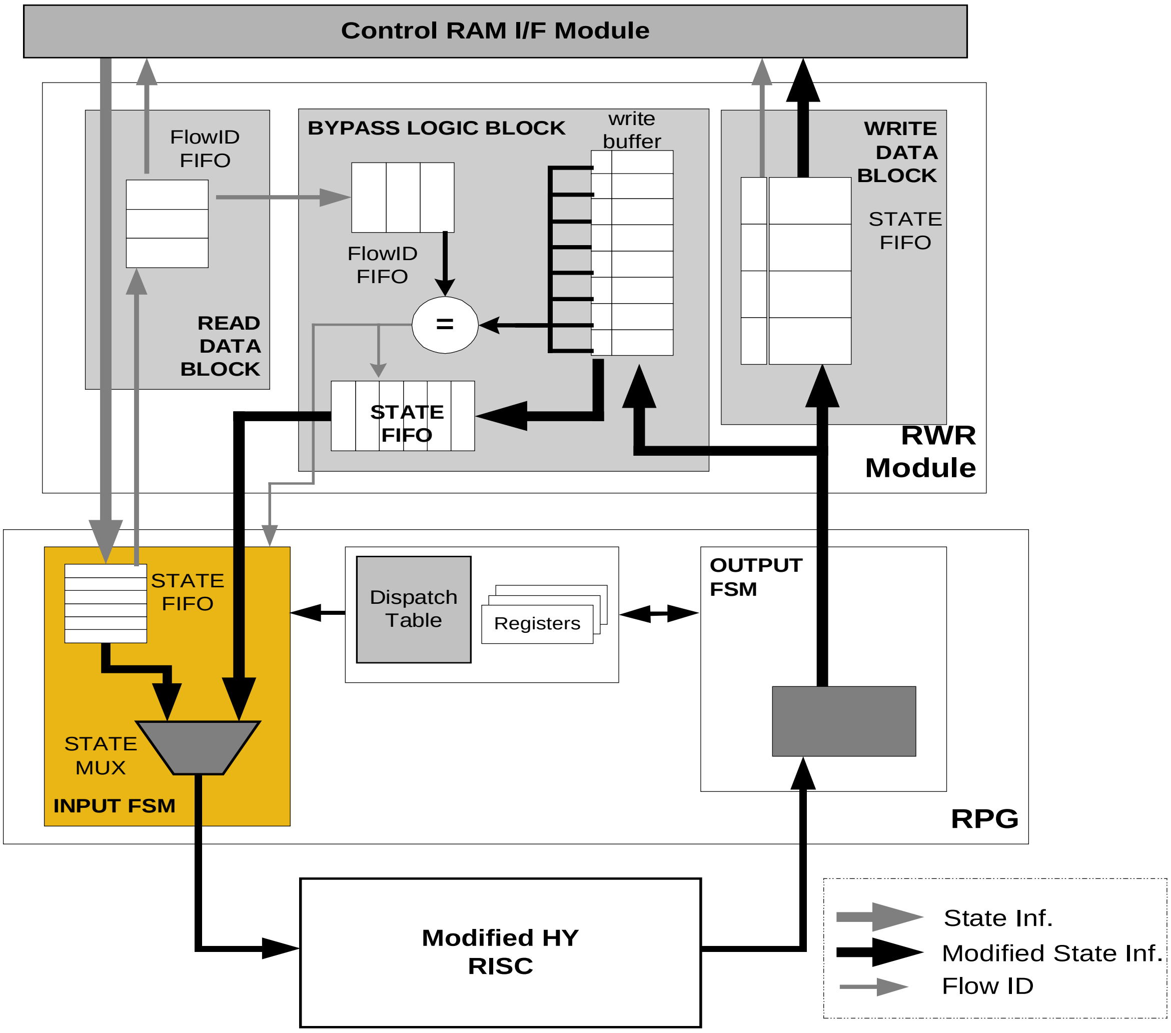


Figure 9a: Actual PPE block diagram

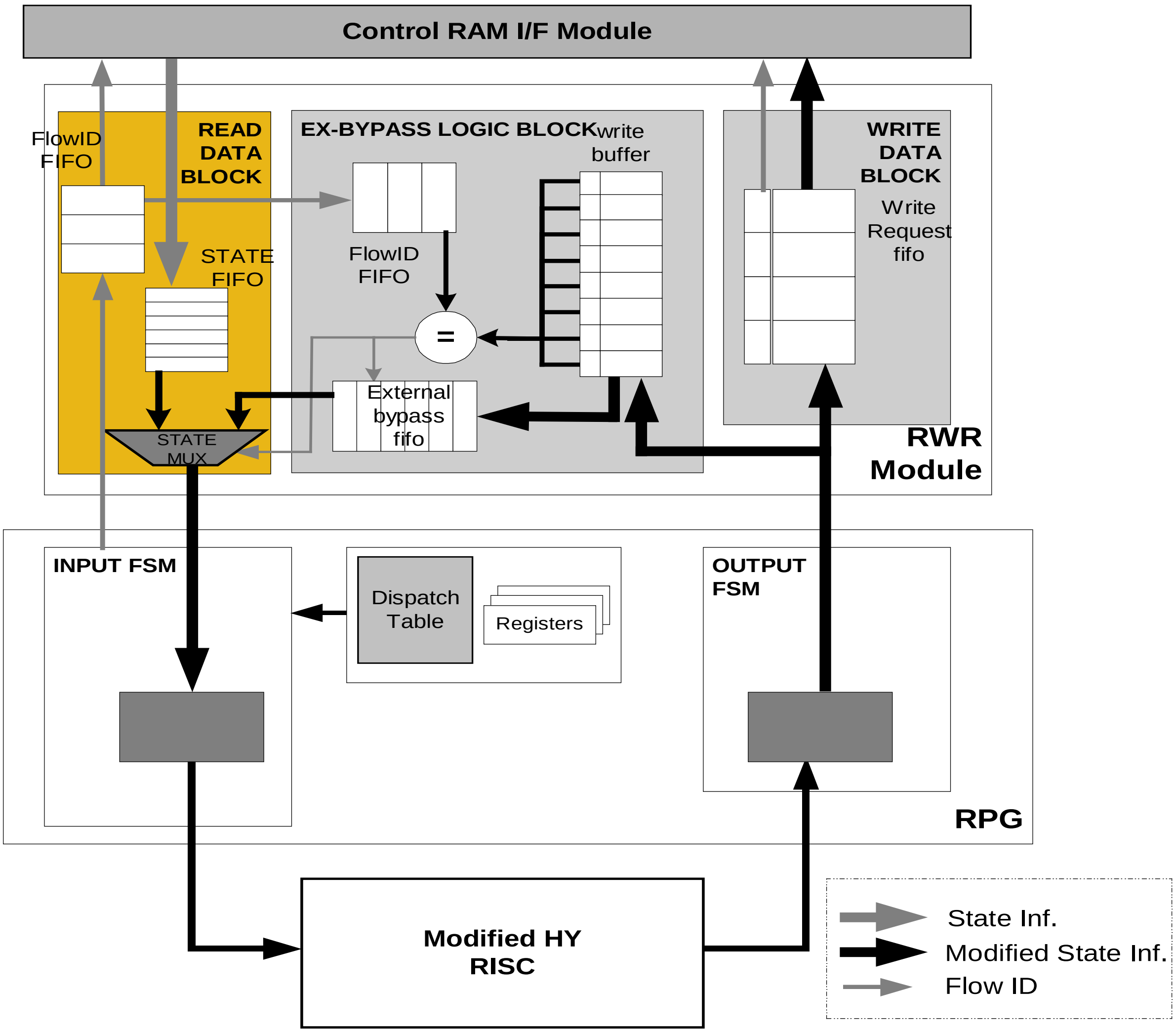


Figure 9b: PPE block diagram that should be designed.

References

[1] G. Konstadoulakis, Ch. Georgopoulos, Th. Orphanoudakis, N. Nikolaou, M. Steck, D. Verkest, G. Doumenis, D. Reisis, J. -A. Sanches and N. Zervos, "A Novel Architecture for Efficient Protocol Processing in High Speed Communication Environments", presented in the IEEE European Conference on Universal Multiservice Network (ECUMN'2000/0), Colmar, France, October 2-4, 2000.

[2] N. Nikolaou, J. Sanchez-P., T. Orphanoudakis, D. Pollatos, N. Zervos "Application Decomposition for High-Speed Network Processing Platforms", to appear in the proc. of the 2nd European Conference on Universal Multiservice Networks (ECUMN'2002), Colmar, France, April 8-10, 2002.

[3] C. Georgopoulos, G. Konstadoulakis, T. Orphanoudakis, N. Nikolaou, J.-A. Sanches, N. Mouratidis, K. Pramataris, N. Zervos "A Protocol Processing Architecture Backing TCP/IP-based Security Applications in High Speed Networks", INTERWORKING'2000, Bergen, Norway, October 2000.

[4] D. Liu, U. Nordqvist, C. Svensson, "Configuration based architecture for high speed and general purpose protocol processing", IEEE Workshop on Signal Processing Systems, October 1999.

[5] PRO3, IST project 99-11449 presentation,
http://www.cordis.lu/

[6] Hyperstone Electronics E1-32X RISC/DSP,
http://www.hyperstone-electronics.com

[7] T. Spalink, S. Karlin, L. Peterson, "Evaluating Network Processors in IP Forwarding", Technical Report TR-626-00, November 15, 2000.

[8] Level One " IXP1200 Network Processor", Advance Datasheet.

[9] Intel web site:
http://developer.intel.com/design/network/products/npfamily/ixp1200.htm

[10] Motorola, C-Port, http://e-www.motorola.com/

[11] Solidum Systems Corp., PAX. port 1100, www.solidium.com

[12] Agere, NPFPP, NPRSP, NPASI, FPL, IDS:
http://www.agere.com/enterprise_metro_access/network_processors.html

[13] MMC web site for nP3400 and other network processors of nP7000 family:

[14] IBM web site:
http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/Networking_Technology

[15] MMC web sites for AnyFlow 5000 family:
http://www.mmcnet.com/Solutions/anyflow5400.asp
http://www.mmcnet.com/Solutions/anyflow5500.asp

[16] Z. Haas, "A Communication Architecture for High-speed Networking", from the Proceedings of IEEE INFOCOM 1990, San Francisco, CA, USA. IEEE, 1990, June 3-7, 1990,

[17] D. Niehaus, A. Battou, A. McFaralnd, B. Decina, H. Dardy, V. Sirkay, B. Edwards, "Performance benchmarking of signalling in ATM networks", IEEE Comm. Mag., August 1997.

[18] C. Maeda, B. Bershad "Protocol service decomposition for high performance networks", Proceedings of he 14th ACM symposium on Operating Systems Principles, December 1993.

[19] D. Feldmeier, A. McAuley, J. Smith, D. Bakin, W. Marcus, T. Raleigh, "Protocol boosters", IEEE Journal on Selected Areas in Comm., vol. 16, No. 3, April 1998.

[20] Peter N. Glaskowsky, "Network Processors Mature in 2001", Microprocessor Report (http://www.mpronline.com/), February 19, 2002.

[21] Samuel J. Barnett, "When Shopping For Network Processors, One Size Does Not Fit all", Electronic Designs magazine, April 2, 2001.