**RC 21865 (98398) 27 October 2000** **Computer Science**

# IBM Research Report

## Design of an Instruction Set for Modular Network Processors

**Tilman Wolf**
IBM Research Division
Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich**

# Design of an Instruction Set for Modular Network Processors

Tilman Wolf
IBM T. J. Watson Research Center

**Abstract**

The demand for more flexibility and functionality in networks has lead to a new router generation that is equipped with software controlled processors in the datapath. These network processors range from configurable protocol processing units to general-purpose processor cores. Being able to combine various modules that implement different packet manipulation functions in a single architecture is key to an modular, extensible, and scalable network processor. We introduce an instruction set architecture that allows to centrally control a number of modules that can work on packets in a pipelined, parallel fashion. We show how the instruction set can be used on an example and compare its performance to a traditional RISC processor system.

## 1 Background

Flexibility has become an increasingly important aspect of networking hardware. Traditional packet forwarding engines implemented as special purpose logic cannot adapt to the numerous new protocols that are introduced networking environment, like IPv6, multicast, and various quality of service schemes. Additionally, research in "active networks" or "programmable switches" has promoted the idea of a store-process-forward paradigm in routers that allows custom processing of data streams [13], [2]. As a result, it is desirable to have the ability to control the data path processing by software rather than hard-wired logic.

This application pull is met by technology push on the integrated circuit side. Application specific integrated circuits (ASIC) have reached the point that it has become possible to combine multiple processing engines, memory, and I/O components on a single piece of silicon. These systems-on-a-chip (SOC) have the advantage of much smaller communication delays between components, higher clock rates, and lower manufacturing costs. In a more recent development, field-programmable gate arrays (FPGA) with a size of ten million gates are becoming an alternative to ASICs for SOC implementations.

While network processors have to be able to do general purpose processing, there is also a large amount of processing that is generic to the network environment and can be done more efficiently with specialized components. Thus, many network processor architectures contain standard RISC-like cores and specialized hardware accelerators. We believe that an architecture that integrates such components in a modular fashion provides the necessary flexibility and scalability. In this paper, we propose an instruction set architecture (ISA) that can be used to program such a modular network processor. We define the instruction set, give an example architecture, and compare its properties to a traditional RISC ISA.

Sections 2 and 3 motivate the use of a modular architecture. Section 4 introduces the basic system architecture. Section 5 describes the instruction set in detail. An example of IP processing is shown in Section 6. Section 7 addresses performance evaluation issues and Section 8 shows related work.

## 2 Modularity

The key to a flexible, scalable network processor architecture is the ability to adapt the configurations to different tasks. Network processors can be used on end-systems, edge routers, and in the backbone. The location determines if the processing requirements are focused on protocol conversion, QoS routing, payload transcoding, or other custom processing.

Another important aspect of a scalable architecture is to span the range from high-performance backbone equipment to low-cost end-point equipment. Modularity is an ideal concept for that since different modules can be implemented in various grades of performance and cost. A target system can be build by combining the interchangeable modules with the appropriate performance and cost. Thus, the main advantages of modular network processors are:

- ability to specialize modules to common or computationally intense task,
- achieve scalability by using various module configurations.

Using specialized hardware ("hardware accelerators"), like digital signal processors (DSP) [7] or customized logic, is useful for common and computationally intense tasks. These modules are tradeoffs between silicon real-estate and processing speedup for packets that can make use of the specialized module. Thus, it is important to use the accelerators for common tasks (Amdahl's Law). Compared to workstation processors, network processors operate in a very limited domain. While packet forwarding can be seen as "general-purpose processing," it is still restricted to processing data packets. Thus, a lot of tasks performed on a packet are the same – or at least very similar to – tasks performed on other packets. As a result, acceleration of these tasks has a performance impact on almost all packets. Examples of such common tasks are CRC and checksum computations or packet classifications.

Scalability is also an important property that can be achieved with modularity. Network processors can be used in all levels of the network, from LAN routers to edge systems to backbone routers. The resulting processing demands are different and cannot be accommodated efficiently by a single configuration. For backbone network processor it makes sense to include high-end, specialized components to achieve maximum performance. On high-volume, low margin components it might be more suitable to aggregate modules in a single more general processor module and have software perform certain functions. This will yield lower performance, but smaller and cheaper processor chips. As the integration level increases over time and gates become cheaper, more and more specialized modules can be introduced to take over the functions performed by software.

Finally, modularity also helps to isolate processing steps to components, which can be used to efficiently parallelize and pipeline processing of multiple packets at the same time. This is explained in more detail the following section.

## 3 Parallelism and Pipelining

Network traffic typically is an aggregate of many connections (flows) from a multitude of independent end-systems. On the low end, a LAN router might encounter only a few dozen active connections at any point in time. Back-bone routers, on the high end, are designed to handle several thousands of active flows at a time. The independence between different flows can be exploited by parallelizing the packet processing tasks. The only order that typically has to be maintained is the sequence of packets within any given flow (actually, the IP does not even require that, but it is commonly assumed). Additionally, there is a basic sequence of steps that packets go through: demultiplexing, processing, forwarding. While the detailed processing sequence might vary, it still leads to possibilities of pipelining the process. The ISA described here is designed to support parallelism as well as pipelining.

Today's network protocols are based on Open Systems Interconnection (OSI) seven layer model and the Department of Defense (DoD) four layer model. Each layer encapsulates the functions of a specific communication abstraction (e.g., link layer for point-to-point communication, network layer for end-system to end-system communication). In general, layers only pass information to the next upper or lower layer. The layered abstraction makes it possible that heterogeneous networking components can communicate together if they implement the same layer interfaces.

Packets that are forwarded by a router, are typically demultiplexed header-by-header to the layer where enough information is available to forward the packet. Plain routers only need to demultiplex packets to layer 3 (network layer) to obtain the destination address of a packet. Routers that perform QoS or context-sensitive routing, though, need to demultiplex packets to layer 4 (transport layer) or even higher to be able to discriminate packets. The significant performance bottleneck with a layered model is that one cannot 'jump' over layers. Each header needs to be 'parsed' before the next header can be decoded. This is due to variable header lengths and arbitrary protocol combinations. This is described in more detail in [3].

As a packet is demultiplexed and passed on to higher layers, there is more information available about the packet. After processing the network layer header, the source and destination of a packet is known. When processing the transport layer, the source and destination ports become available and the packet can be assigned to a flow. Going further into application headers (like HTTP), one can determine the actual content of a packet. This information can be used then to process/forward the packet. Based on how much information on the packet is available, different levels of "customized processing" is possible. There are four levels of processing that we can distinguish:

- Interface Level: only the data link protocol is known, possible functions: framing, integrity check, bridging
- Protocol Level: set of used protocols is know, possible functions: routing
- Flow Level: connection to which packet belongs is known, possible functions: QoS, firewalling
- Packet Level: packet is treated as individual datagram: "active processing"

As the packet goes through the various levels, the system has to make the instruction code available that processes the packet in its individual way.

# 4 System Architecture

To illustrate the network processor instruction set (NPIS), we introduce a basic system architecture.

## 4.1 Control

The network processor instruction set (NPIS) is the control interface to the network processor. The goal is to provide one common control architecture to the various modules. While different modules can be implemented in a variety of ways, like custom logic or DSPs, they are programmed through on single instruction set. This requires a lot of flexibility in the instruction set, since each module might use a different programming abstraction. This flexibility is achieved by having the individual module translate the instruction into microcode.

In a traditional RISC processor, a program is compiled to binary code. Each instruction then is dynamically decoded and the respective microcode controls all the hardware components of the processor (see Figure 1). In the modular approach, the control component should be fairly independent from the particular modules that are used in a configuration. Thus, the individual

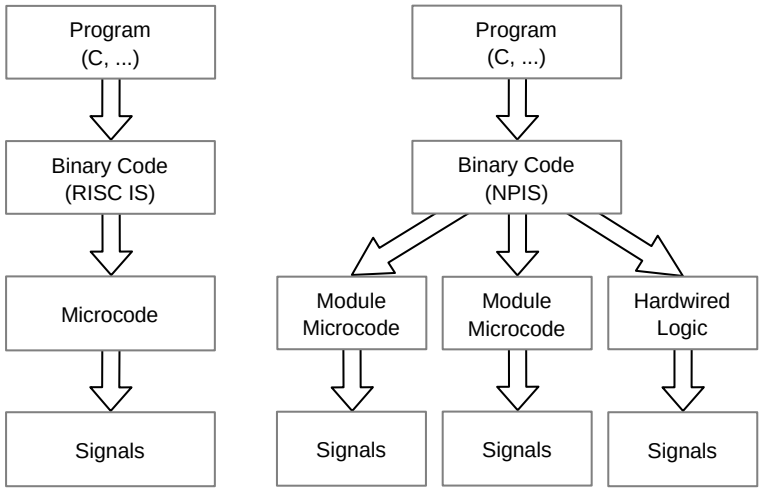


Figure 1: Per-Module Microcontrol.

module control should be contained in the module itself. In this case, NPIS instructions are decoded by the central control only to the point where it is clear to which module they refer. Then they are passed to the module where they are translated into microcode or directly trigger customized logic.

### 4.2 Network Processor

A high-level, functional illustration of the network processor architecture is shown in Figure 2. There are five main components: the I/O system that receives and transmits packets, the memory that stores packets, classification tables, and processing instructions, the various modules, a set of context mappers that make the proper registers available to the modules, and a centralized system control.

Packets that are received on the I/O interface are placed in a free memory location (the system controls provides the I/O interface with a list of free memory locations). For the processing of a packet, a context mapper and a control modules is assigned to the packet. The control module executes the basic packet processing code. The context mapper makes a set of registers, memory locations, and special purpose registers (instruction counter, status register) available to the control module and any other module that is assigned to the packet. The control module can dynamically acquire other modules that are necessary for the packet processing. These modules then get access to the packet data over the context mapper. The processing control always stays with the control module. When processing has finished, the packet is sent out and the context mapper is used to process another packet.

As long as there is no conflict in accessing modules or memory locations, several packets can be processed independently in parallel. By logically segmenting the memory and making enough special purpose modules available, these conflicts can be avoided.

## 5 Network Processor Instruction Set

The network processor instruction set (NPIS) provides the programmability of the various modules that are available in the system. As described in the previous section, modules are controlled by control modules that execute the processing code.

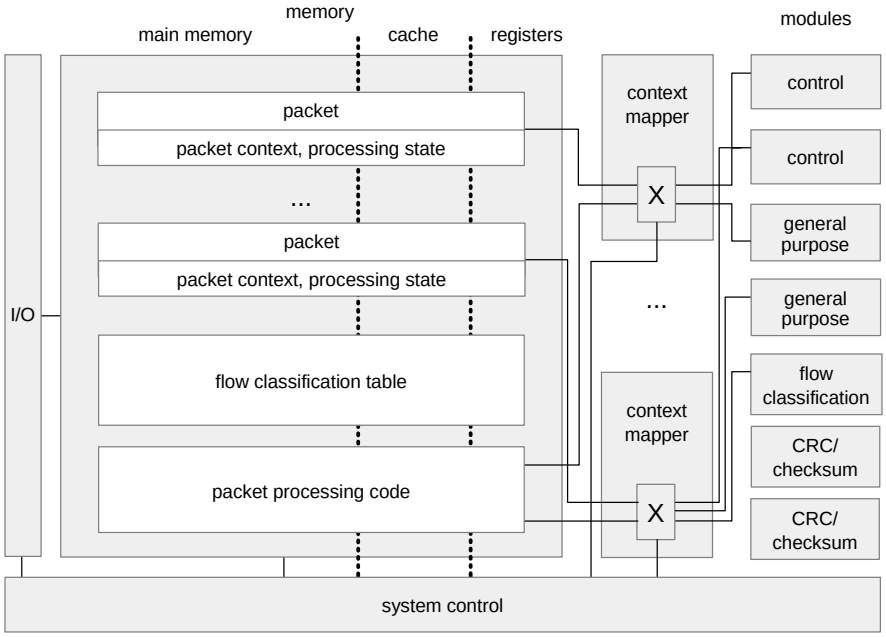


Figure 2: Architecture.

## 5.1 Instruction Layout

The instruction set has to provide the extensibility to support many different modules. One basic observation, though, is that, although there might be many different modules in a system, a single packet requires only a handful of modules for its processing on a router. This leads to the idea to augment instructions by a short 'tag' that identifies for which module the instruction is destined.

The basic process of executing instructions on a module follow the following sequence:

- The control module acquires another module $M$ and defines a tag $T$ that is used to represent $M$.
- All instruction opcodes $O$ that are to be executed by $M$ are tagged with $T$: $O@T$.
- Once a module is not needed for the processing anymore, it can be released and the tag $T$ reused for another module.

While the explicit acquisition of modules represents an overhead over direct addressing of modules, it provides the following benefits:

- Small tag space. Tags are used on a per-packet basis and only have to be unique within the code fragment where they are used.
- Indirect mapping of physical modules to tags. Tags allow the usage of the same code fragment in parallel on different packets. One packet can acquire module $M_1$ for tag $T$ and another packet can use module $M_2$ for tag $T$. If both packets execute $O@T$, there is no conflict, because different modules execute the instruction in parallel.
- Resource control and load balancing. The explicit acquisition and release of modules makes it possible to determine which modules are used in which code fragment. This can be used to determine how processing can be parallelized and pipelined.

Except for the bit field of the module tag, there is no specification on how the remaining bits are used for the instructions. This allows to define instructions that are suited to the individual

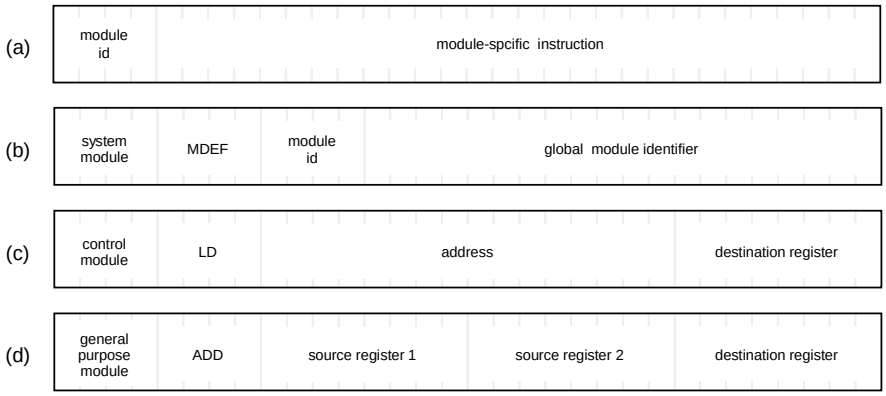


Figure 3: Instruction Set.

module. It is the job of the compiler to generate the appropriate sub-instructions for the modules used. Of course, there are two pre-defined tag-module combinations for our system. The system module that is responsible for receiving, transmitting, and dropping the packet is referred to by tag 0. The control module that is necessary to run basic load/store and control flow operations has tag 1.

Figure 3 shows four examples for NPIS instructions. It is assumed that the instruction size is 32 bit, where 4 bit are used for module tags. Instruction (a) shows the basic instruction layout. The first 4 bit represent the module tag (i.e., 0000 for the system module, or 0001 for the control module). The remaining 28 bit are defined for the individual module. Instruction (b) shows the MDEF instruction of the system module that defines a tag for a module. The 4 bit module id is the tag that is to be used for a module of the type given by the global identifier. It is the job of the system to assign an available module of this type to the packet. The context mapper has to keep track of the mapping $tag \rightarrow module$ and translate instructions accordingly (thus, the name "context mapper"). Instruction (c) is an example for a traditional memory load instruction of the control module. Except for the additional tag, there are no differences to RISC. This can also be seen in instruction (d) that is an add instruction from the general purpose computing module.

## 5.2 Registers, Immediates, and Addressing Modes

While instructions of modules can be formed arbitrarily, they have to agree on how to use the registers that are shared with the other modules. For now, we assume that an 8 bit space is used for one argument. To make the instruction set more flexible and also to consider immediate values, these 8 bits are split between register identifiers and immediates. With the most significant bit set, the remaining 7 bit are interpreted as referring to a register, with the most significant bit not set, they are interpreted as an immediate value. In case larger immediates are required, they can be explicitly loaded with LDIL and LDIH instructions.

For addresses, a 16 bit field is used. This field is logically divided into two 8 bit fields that represent two register/immediate values. The address is obtained by adding the values of the two registers/immediates. With this general scheme, it is possible to do direct, displacement, register deferred, and indexed addressing.

## 5.3 Instruction Set

A list of instructions for the system, control, general purpose processing, classification and checksum modules is shown in Table 1. For each module, the individual instructions are shown, their

arguments, the size of all components in bits, and a description of the functionality.

# 6 Example

In this section, we present an example of a modular network processor and show the instruction set in use. As system configuration, we assume the network processor shown in Figure 2. We assume that the system has received a packet from the I/O module and put it into memory. The control modules then starts processing the packet with the code shown in Figure 4. We assume that the packet is an Ethernet/IP/TCP packet.

The following processing steps are performed on the packet:

- check Ethernet CRC and extract LLC/SNAP value,
- check IP header correctness and checksum,
- extract IP source, destination, and layer 4 port numbers for classification,
- forward (or drop) packet based on classification table entry,
- adjust IP TTL field and recompute header checksum,
- generate Ethernet header and packet CRC,
- enqueue packet to be sent out on outgoing port.

One can see that the code is 'linear' ad does not contain any loops. Additionally, one can determine which portions of the code require which module just by looking at the instructions. This can be used to determine how the this code can be parallelized and pipelined.

# 7 Performance Evaluation

The performance of the instruction set can be evaluated in multiple ways. We look at it under two aspects. One is a comparison of a single processor system. The other is the comparison of a multi-module system with a specialized components to a system with multiple RISC cores. The results are based on the example shown in Figure 4. Since we do not have a compiler for the network processor instruction set, we limit our comparison to this one example and try to extrapolate results to be generally meaningful.

## 7.1 Single Processor

The efficiency of the instruction set can be observed when the packet processing code is compared to code for a traditional RISC processor. Efficiency can be expressed in terms of code size, execution cycles, and number of memory references.

### 7.1.1 Code Size

Both NPIS and RISC are based on 32 bit instruction words. Also, there are many instructions that have a one-to-one correspondence in the instruction sets. Thus, no significant differences in code size for these instructions can be expected. Minor differences are caused by a few instructions that increase the NPIS code size compared to RISC code size:

- Module acquisition and release instructions. The MDEF and MUDEF instruction is used to assign modules to a packet and release them after they had been used. Depending on the packet data path, this is usually limited to a handful of modules.

| Opcode | Arguments | Size in bits | Description |
|---|---|---|---|
| System module: | | | |
| DQUE | | 4+4 | dequeue packet from queue |
| ENQUE | R1 | 4+4+8 | enqueue packet into queue R1 |
| DROP | | 4+4 | drop packet |
| MDEF | M, n | 4+4+4+20 | acquire module of type n as M |
| MUDEF | M | 4+4+4 | release module M |
| Control module: | | | |
| LD | R1, A | 4+4+8+16 | load |
| ST | A, R1 | 4+4+16+8 | store |
| LDIL | R1, n | 4+4+8+16 | load immediate into lower half of word (clear rest) |
| LDIH | R1, n | 4+4+8+16 | load immediate into higher half of word |
| RBITS | R1, R2, n1, n2 | 4+2+8+8+5+5 | read n1 bits into R2 from R1 starting at the n2$^{nd}$ bit |
| WBITS | R1, R2, n1, n2 | 4+2+8+8+5+5 | write n1 bits from R2 to R1 starting at the n2$^{nd}$ bit |
| CMP | R1, R2 | 4+8+8+8 | compare and set status bits |
| BR | A | 4+8+20 | unconditional branch |
| BREQ | A | 4+8+20 | branch if equal |
| BRL | A | 4+8+20 | branch if less |
| BRLE | A | 4+8+20 | branch if less or equal |
| NOP | | 4+8 | no operation |
| General purpose module: | | | |
| ADD | R1, R2, R3 | 4+4+8+8+8 | add |
| SUB | R1, R2, R3 | 4+4+8+8+8 | subtract |
| MULT | R1, R2, R3 | 4+4+8+8+8 | multiply |
| AND | R1, R2, R3 | 4+4+8+8+8 | logic and |
| OR | R1, R2, R3 | 4+4+8+8+8 | logic or |
| XOR | R1, R2, R3 | 4+4+8+8+8 | logic xor |
| NOT | R1, R2 | 4+4+8+8 | logic not |
| SHL | R1, R2, n | 4+4+8+8+8 | shift left by n bit positions |
| INC | R1 | 4+4+8 | increment |
| DEC | R1 | 4+4+8 | decrement |
| Classification module: | | | |
| TABLE | R1 | 4+4+8 | choose table pointed to by R1 |
| LOOKUP | R1, n | 4+4+8+8 | classification for n entries (return value in R1) |
| Checksum module: | | | |
| CRC | R1, R2, R3 | 4+4+8+8+8 | CRC starting at R2 for R3 bytes (result in R1) |
| CSUM | R1, R2, R3 | 4+4+8+8+8 | checksum starting at R2 for R3 bytes (result in R1) |

Table 1: Network Processor Instruction Set. Arguments to instructions can be registers (Rn), addresses (A), immediate values (n), or module identifiers (M).

```
    MDEF 0, 0x0000          # define control module
start:
    DQUE R1                 # get packet from queue
    LD@0 R2, R1+4           # load packet length into R2
    LD@0 R3, (R1)           # load actual packet start into R3
    MDEF 1, 0xcafe          # define CRC/checksum module
    CRC@1 R4, R3, R2        # compute CRC on packet
    MUDEF 1                 # release CRC/checksum module
    CMP@0 R4, R0            # check if result is 0
    BREQ@0 label1
    BR@0 bad                # else drop
label1:
    LD@0 R4, R3+12          # load ethernet type word
    RBITS@0 R5, R4, 0, 16   # extract actual 16 bit
    MDEF 2, 0xbeef          # acquire general purpose module
    ADD@2 R9, R3, 16        # adjust beginning of packet pointer
                            # by ethernet header size
    SUB@2 R10, R2, 16       # adjust packet length
    LDIL@0 R6, 0x0800       # IP identifier
    CMP@0 R6, R5            # compare ethernet type with IP identifier
    BREQ@0 ip_processing
    LDIL@0 R6, 0x0806       # ARP identifier
    CMP@0 R6, R5            # compare ethernet type with ARP identifier
    BREQ@0 arp_processing
    ...
    BR@0 bad
ip_procesing:
    RBITS@0 R5, R9, 0, 4    # extract IP version
    CMP@0 R5, 4             # compare IP version to 4
    BREQ@0 label2
    BR@0 bad                # drop if not version 4
label2:
    RBITS@0 R5, R9, 4, 4    # extract header length
    SHL@2 R5, R5, 2         # convert length from words to bytes
    CMP@0 R5, 20            # compare header length to minimum
    BRL@0 bad               # drop is header too short
    MDEF 1, 0xcafe          # define CRC/checksum module
    CSUM@1 R7, R9, R5       # compute checksum over header
    CMP@0 R7, R0            # check if result is 0
    BREQ@0 label3
    BR@0 bad                # else drop
label3:
    RBITS@0 R5, R9, 16, 16  # get packet length
    CMP@0 R5, R10           # compare length field value with
    BREQ@0 classification   # actual packet size
    BD@0 bad                # drop if unequal
classification:
    MDEF 3, 0xabba          # acquire classification unit
    LDIL@0 R11, IPTABLE     # load address of IPTABLE used for
                            # IP classification lookups
    TABLE@3 R11             # set table to be used
    RBITS@0 R18, R9, 8, 8   # extract TOS field
    LD@0 R19, R9+3          # get word with protocol field
    RBITS@0 R19, R19, 8, 8  # extract protocol field
    LD@0 R16, R9+4          # load IP source address
    LD@0 R17, R9+5          # load IP destination address
    CMP@0 R19, 6            # check for TCP protocol type as layer 4
    BREQ@0 tcpudp
    CMP@0 R19, 17           # check for UDP protocol type as layer 4
    BREQ@0 tcpudp
    LOOKUP@3 R8, 4          # neither TCP nor UDP, do 4-tuple lookup
    MUDEF 3                 # release classification unit
    BR@0 action
tcpudp:
    ADD@2 R11, R9, R6       # adjust beginning of packet pointer
                            # by IP header length
    SUB@2 R12, R10, R6      # adjust packet length
    RBITS@0 R20, R11, 0, 16 # extract layer 4 source port
    RBITS@0 R21, R11, 16, 0 # extract layer 4 destination port
    LOOKUP@3 R8, 6          # do 6-tuple lookup
    MUDEF 3                 # release classification unit
action:
    LD@0 R11, R8+0          # get action field from table
    CMP@0 R11, 1            # 1=forward
    BREQ@0 forward
    CMP@0 R11, 2            # 2=control
    BREQ@0 control
    BR@0 bad                # 0=drop, unknown=drop
forward:
    LD@0 R4, R9+3           # get TTL word
    RBITS@0 R11, R4, 0, 8   # get TTL bits
    DEC@2 R11               # decrease TTL
    WBITS@0 R4, R11, 0, 8   # put TTL back
    WBITS@0 R4, R0, 16, 16  # set checksum to 0
    ST@0 R9+3, R4           # store new word
    MDEF 1, 0xcafe          # define CRC/checksum module
    CSUM@1 R7, R9, R5       # compute checksum over header
    MUDEF 1
    OR@2 R4, R4, R7         # add checksum to word
    ST@0 R9+3, R4           # store new word
etheroutput:
    LD@0 R4, R9+8           # get first 32 bit of l2 addresses
    ST@0 R3+0, R4           # store in packet
    LD@0 R4, R9+12          # next 32 bit
    ST@0 R3+4, R4
    LD@0 R4, R9+16          # and final 32 bits
    ST@0 R3+8, R4
    SUB R4, R2, 4           # packet length without CRC
    MDEF 1, 0xcafe          # define CRC/checksum module
    CRC@1 R5, R3, R4        # compute CRC on packet
    MUDEF 1                 # release CRC/checksum module
    ADD@2 R4, R3, R4        # compute CRC address
    ST@0 R4+0, R5           # store CRC
output:
    LD@0 R5, R8+4           # get output queue
    ENQUE R5, R3, R2        # enqueue packet
    MUDEF 2                 # release general purpose unit
    BR@0 start              # that'a all, next packet
control:
    ...
arp_processing:
    ...
bad:
    DROP                    # drop packet and clear all state
    MUDEF 2                 # release general purpose unit
    BR start                # start over
```

Figure 4: Example NP IS code for IP packet forwarding.

- Loading of large immediate values. Due to the compactness of NPIS, the size of immediate values is limited to typically 7 *bit* or 16 *bit* for the load instruction. Due to the high spatial locality of network processing code, this should not have a negative effect on the code. If a program uses many large immediate values, though, it can lead to an increase in code size, because a RISC load has to be replaced by several NPIS instructions.

Since NPIS is adapted to the networking environment, it also contains several instructions that reduce the code size, because they implement common functions that require multiple RISC instructions. These instructions are:

- Bit read and write instruction. Due to the compactness of protocol headers, often several parameters are packet in a 32 bit word. To process the protocol header, it is necessary to extract the bits corresponding to a single value. The specialized RBITS instruction and the corresponding writing instruction WBITS implement this functionality. To achieve the same functionality in a RISC instruction set, typically two operations are needed (two shifts or a load immediate and a logic AND) for RBITS and three to four instructions for WBITS. This difference might have a noticeable effect on code size, since in the less than 100 instructions of the example, RBITS was used 9 times and WBITS was used twice.
- System instructions. These basic packet operations, DQUE, ENQUE, and DROP, are implemented in a single instruction in NPIS. RISC requires multiple instructions depending on how the enqueue and dequeue operations are implemented. These instructions occur exactly twice for each packet (DQUE and ENQUE or DQUE and DROP).
- Special module instructions. These instructions aggregate much functionality in a single instruction. The reduction in code size can be significant for those instructions. For example, checksum computation is represented by a single CSUM instruction, whereas in a RISC instruction set it has to be implemented with many basic instructions. In particular if loops are unrolled for efficiency, the NPIS representation is significantly shorter.

### 7.1.2 Memory References

A component that significantly affects the overall performance of a system, is the memory system. Off-chip memory accesses due to cache misses are costly and cause processor stalls. It is difficult to compare the effects of different instruction sets on the memory system, since locality in access patterns plays an important role. In general, though, the NPIS uses the same type of load and store instructions as a RISC processor. Also, it accesses the same data (packet header, classification tables, ...) roughly in the same order. Thus, one can assume that the overall performance with respect to the memory system is comparable to that of a traditional RISC.

## 7.2 Multiple Modules

The introduction of specialized modules require additional chip area. In general, a specialized module pays off when it increases the speedup or throughput of the chip with little additional chip area. To quantify this property, we define the following simple performance metric:

$$performance = \frac{throughput}{area} \tag{1}$$

The performance is higher when higher throughput is achieved and it is lower if more chip area is required. For the RISC processor, the throughput can be defined by the processor clock $c_{RISC}$, the average number of instruction that are required for packet processing, $n_{RISC}$, and the average clocks per instruction, $CPI_{RISC}$. The area of a RISC processor is $a_{RISC}$. The area for memory is not considered here, since it can be assumed to be the same for different configurations. Using basic evaluation techniques from [4], the performance of a RISC processor can be computed as

$$performance_{RISC} = \frac{\frac{c_{RISC}}{n_{RISC} \cdot CPI_{RISC}}}{a_{RISC}}. \quad (2)$$

If we look at a modular system that uses two modules $A$ and $B$, one which does general computations similar to a RISC and one that is a hardware accelerator for a certain function, then we can describe the performance as

$$performance_{NPIS} = \frac{\frac{c_{NPIS}}{n_{NPIS} \cdot CPI_{NPIS}}}{a_A + a_B}. \quad (3)$$

The hardware accelerator replaces some of the RISC instructions, so in general on can assume $n_{NPIS} < n_{RISC}$, but since it might do more complex processing in each instruction $CPI_{NPIS} > CPI_{RISC}$. This formula can of course be extended to more than two modules.

As an example, we look at a 200 MHz RISC processor which executed 150 instructions per packet at a CPI of 1.2 and uses 3 mm$^2$ or chip area. This is compared with a two-module system that has one module that is similar to the RISC (control module) and a module that does a hardware accelerated function (checksum module). Due to the accelerator, only 80 instructions need to be executed, but the CPI of 1.5 is slightly higher. The clock is also 200 MHz, and the modules require 2 mm$^2$ each. Thus, the performance of the RISC system is

$$performance_{RISC} = \frac{\frac{200 \cdot 10^6}{150 \cdot 1.2}}{3} = 370370. \quad (4)$$

The performance of the modular system is

$$performance_{NPIS} = \frac{\frac{200 \cdot 10^6}{80 \cdot 1.5}}{2 + 2} = 416666. \quad (5)$$

For this example, the performance of the modular system is higher than the standard RISC system. It is important to note that the performance metric cannot be used to determine the speedup obtained by using one configuration over another. To obtain speedup, other components, like memory and I/O have to be included.

# 8 Related Work

There are a number of commercial network processors available or soon to be available. They are typically single-chip solutions designed to process upward from OC-48 speeds (2.4 Gb/s). The general architecture is a processing cluster of RISC-like processing units. In some cases these processors are controlled by a central processor. Packets are processed in parallel by those processors. The following list is a rough comparison of the network processors for which public information is available:

- IBM IBM32NPR161EPXCAC133 (Rainier) [5]: 16 processing units, one control processor, 133 MHz clock rate, 1.6 GB/s DRAM bandwidth, 8 Gb/s line speed, 2 threads per processor.
- Intel IPX1200 [6]: 6 processing units, one control processor, 200 MHz clock rate, 0.8 GB/s DRAM bandwidth, 2.6 Gb/s line speed, 4 threads per processor.
- Lexra NetVortex [8]: 16 processing units, 427 MHz clock rate, over 20 Gb/s line speed, 4 threads per processor.
- Lucent Fast Pattern Processor [9]: 3 VLIW processing units, one control processor, 133 MHz clock rate, 1.1 GB/s DRAM bandwidth, 5 Gb/s link rate, 64 threads per processor.
- MMC nP3400 [10]: 2 processing units, 220 MHz clock rate, 0.5 GB/s DRAM bandwidth, 5 Gb/s link speed, 8 threads per processor.

- Motorola C-5 [1]: 16 processing units, one control processor, 200 MHz clock rate, 1.6 GB/s DRAM bandwidth, 5 Gb/s line speed, 4 threads per processor.
- Tsqware TS704 [14]: 4 processing units.
- Vitesse Prism IQ2000 [12]: 4 processing units, 200 MHz clock rate, 1.6 GB/s DRAM bandwidth, 6.4 Gb/s line speed, 5 threads per processor.

Another approach is to use SIMD processors, which were mainly used for media applications, and adapt them to the network environment. PixelFusion uses their Fuzion 150 [11] for this purpose. It has 1,500 simple processing units clocked at 200 MHz with 24 MB on-chip memory and 6.4 Gb/s external bandwidth. It is not clear, though, that network processing can be parallelized enough to make use of this large number of processors.

One commercial network processor that uses a specialized instruction set, is the Inter IPX1200 [6]. In addition to basic RISC ALU and branch instructions, the instruction set contains a set of specialized read/write and load/store instructions that explicitly specify the memory that they address (registers, SRAM, or SDRAM). It also contains two instructions that perform 48 and 64 bit hash functions.

# 9 Summary

Modularity in network processors provides the flexibility to adapt one basic system architecture to different tasks and performance requirements. Modularity also supports the parallel and pipelined processing of packets, which is the basis to achieving high throughput. The network processor instruction set pushes the decoding of instructions into the individual modules. This gives the instruction set the extensibility that is required for this environment. Instructions are tagged with the identifier of the module that they are meant for. A context mapping unit provides the modules with the appropriate register set and data of the packet that they are processing. The example code for processing an IP packet shows how the instruction set can be used. We also defined a performance metric that can be used to compare modular system configurations.

# References


[1] C-Port Corporation. *C-5$^{TM}$ Digital Communications Processor*, 1999. http://www.cportcorp.com/solutions/docs/c5brief.pdf.

[2] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vincente, and D. Villela. A survey of programmable networks. *Computer Communication Review*, 29(2):7–23, Apr. 1999.

[3] D. Decasper. *A Software Architecture for Next Generation Routers.* PhD thesis, ETH Zurich, 1999.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, second edition, 1995.

[5] IBM Corp. *IBM Power Network Processors*, 2000. http://www.chips.ibm.com:80/products/wired/communications/network_processors.html.

[6] Intel Corp. *Intel IXP1200 Network Processor*, 2000. http://developer.intel.com/design/network/ixp1200.htm.

[7] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals: Architectures and Features.* IEEE Press Series on Signal Processing, Jan. 1997.

[8] Lexra Inc. *NetVortex Network Communications System Multiprocessor NPU*, 2000. http://www.lexra.com/products.html.

[9] Lucent Technologies Inc. *PayloadPlus$^{TM}$ Fast Pattern Processor*, Apr. 2000. http://www.agere.com/support/non-nda/docs/FPPProductBrief.pdf.

[10] MMC Networks, Inc. *nP3400*, 2000. http://www.mmcnet.com/.

[11] PixelFusion, Ltd. *Fuzion 150 Product Overview*, 2000. http://www.pixelfusion.com/products/FUZION150A4_Product_Overview.pdf.

[12] Sitera Inc. *Prism IQ2000 Network Processor Family*, 2000. http://www.sitera.com/products/prism.pdf.

[13] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[14] T.sqware Inc. *TS704 Edge Processor Product Brief*, 1999. http://www.tsqware.com/.