CHAPTER 3

PACKET PROCESSING FUNCTIONS OF NETWORK PROCESSOR

3

PACKET PROCESSING FUNCTIONS OF NETWORK PROCESSOR

3.1 INTRODUCTION

In this chapter, we discuss implementation of two packet processing functions of Network Processor namely: the packet encryption and packet classification. Instead of using DES, 3DES and AES algorithms, [43] we suggest implementation of IDEA (International Data Encryption Algorithm) encryption algorithm as an integrated cryptography core in network processor. Packet classification is one of the main task of NPU. We realize packet classification in software by C program. We realize packet classification for router using hardware and packet classification for terminal using hardware software co-design. This is an effort to understand and evaluate above methods for packet encryption and packet classification.

3.2 PACKET ENCRYPTION FUNCTION IMPLEMENTATION

3.2.1 IDEA Encryption Algorithm and its Interface with Network Processor

Data Communication and data transaction security is must in new emerging electronic business and commercial applications. Data Communication security is performed at Network layer and Data transaction security is performed at application layer. A collection of cryptography applications such as secure IP (IPsec) and Virtual Private Networks (VPNs) has been widely deployed in both routers and end systems. Security related applications are all computational intensive applications that can consume as much as 95 percent of an application server's processing capacity [103]. As demands for communication security grow, cryptographic processing becomes another type of application domain. The fastest security processor can achieve the speed of few Gbps.

Currently available NPU and Cryptographic processors use DES, 3DES and AES algorithms. The IDEA is a symmetric block cipher, developed in 1990-92 by Xuejia Lai and James Massey of the Swiss Federal Institute of Technology. The input and output block of the IDEA are 64 bit long while the key is 128 bit long (as compared to 56-bit key in the DES). We prefer IDEA over other algorithms due to the following reasons:

1. IDEA clearly excels over AES and 3DES in Embedded Systems Applications and in hardware solutions because it does not use substitution boxes (s-box), thus eliminating additional memory access.
2. An architectural analysis of different cryptographic algorithms suggest that IDEA is one of the best and more secure block ciphers available today [103].
3. IDEA performance fact sheet is shown in table 3.1 [46].

Table 3.1 Embedded Systems Performance Comparison

Performance Comparison		IDEA [Mbit/sec]	AES [Mbit/sec]	3DES [Mbit/sec]	Source
Embedded Systems	C167/20MHz	0.22	0.06	0.03	Ascom Systec 2002,CH
	ARM7/20MHz	0.42	0.16	0.10	
	MCF5397/90MHz	1.47	0.65	0.54	

Thus IDEA is the best option among other algorithms for incorporation in NPU.

3.2.2. Issues for Incorporating Cryptographic Application for NPU

When designing a network security product, one must consider both the packet-processing requirements and the security requirements. Adding security functionality into the silicon area of network processor while maintaining wire rate and minimizing new silicon area is a challenge.

The securing of network traffic is portioned into two partitions: protocol processing and cryptographic algorithm processing [101].

a) Protocol processing would include Encapsulating Security Protocol (ESP), Authentication Header (AH), Secure Sockets Layer (SSL), Transport Layer Security (TLS), and other non security protocols such as Transmission Control Protocol (TCP) and the Internet Protocol (IP) processing.
b) Cryptographic algorithm processing includes the data manipulation that would need to be done on the entirety of the payloads, such as confidentiality and integrity. The cryptography unit may be comprised of several algorithms that in conjunction must provide data confidentiality and data integrity. Each algorithm has its own set of trade-offs and challenges, in terms of silicon area, parallelism, and symmetry.

3.2.3. Implementation of IDEA

We have implemented encryption of IDEA (International Data Encryption Algorithm) using ALTERA's Quartus Development tools. IDEA is a block cipher that uses a 128-bit key to encrypt data in blocks of 64 bits, while DES uses only 56-bit key.

Design principles for hardware implementation are as follows:

(i) Similarity of encryption and decryption: Encryption and decryption should differ only in the way of using the key so that the same device can be used for both encryption and decryption. IDEA has a structure that can satisfy this requirement. (ii) Regular structure: The cipher should have a regular modular structure to facilitate VLSI implementation. IDEA is constructed from two basic modular building blocks repeated multiple times.

IDEA encryption: The single round of IDEA encryption is shown in the Figure 3.1. There are two inputs to the encryption function: I) the plaintext to be encrypted II) the key. Plaintext is 64 bits in length and the key is 128 bits in length in this particular case. The 64 bit data block is divided into four 16-bit sub blocks: X1, X2, X3 and X4. Similarly 128 bit key block is divided into six 16-bit sub blocks: Z1, Z2, Z3, Z4, Z5 and Z6. These four sub blocks as well as six key blocks become input to the first round of algorithm. The IDEA algorithm consists of eight rounds followed by a final transformation. In each round, the sequence of events is shown in figure 3.1.

The output of the first round is the four subblocks. Swap the two inner blocks (except for the last round) and that's the input to the next round. After the eighth round, there is final output transformation as shown in figure 3.2:

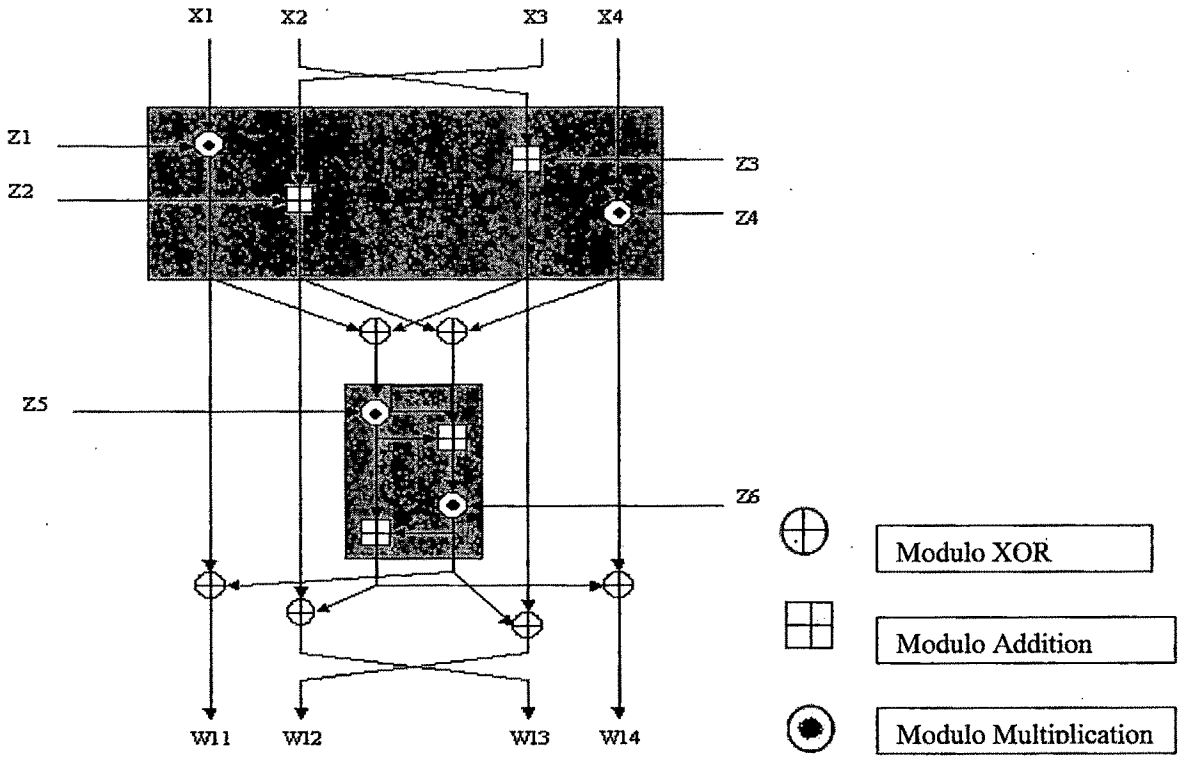

Figure 3.1 Single Round of IDEA

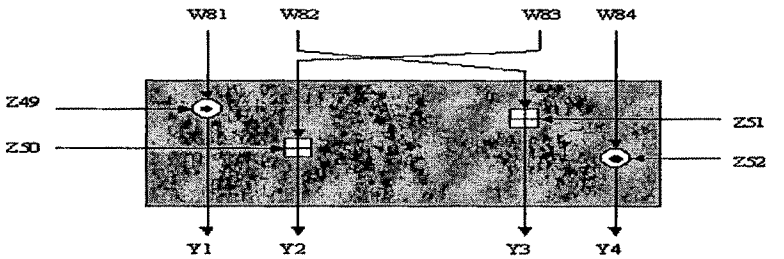

Figure 3. 2 Output Transformation stage of IDEA

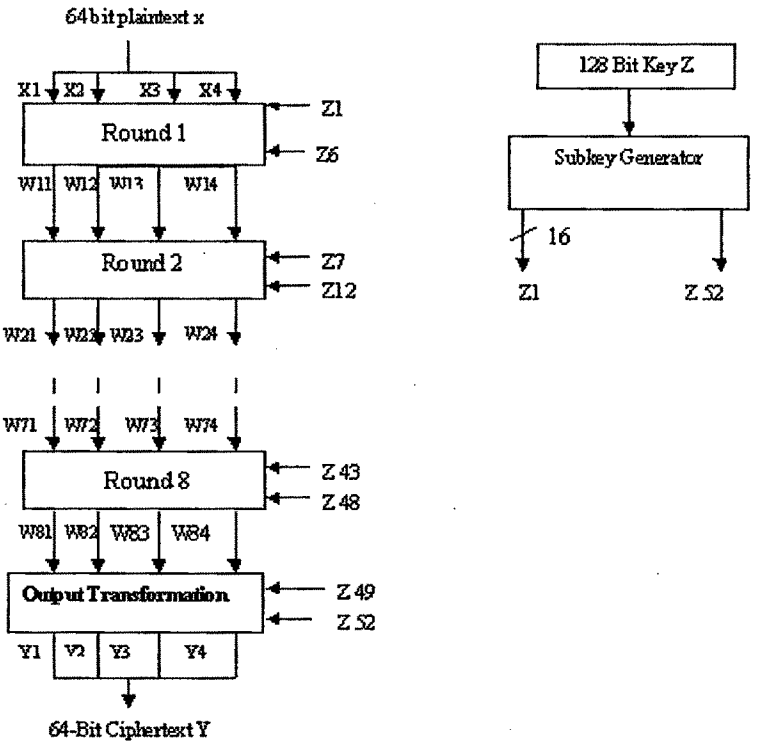

Figure 3.3: Overall IDEA Structure

Finally, the four sub blocks are reattached to produce cipher text. Creating the subkeys is easier. The algorithm uses 52 of them (six for each of the eight rounds and four more for the output transformation.) First, the 128 –bit key is divided into eight 16-bit subkeys. These are first eight subkeys for the algorithm. Then the key is rotated 25 bits to the left and again divided into eight subkeys. The first four are used in round 2; the last four are used in round 3. The key is rotated another 25 bits to the left for the next eight subkeys, and so on until the end of the algorithm. The IDEA algorithm was implemented as shown in *Figure 3.3*, using ALTERA's MAX PLUS/QUARTUS software for VLSI synthesis. The major blocks implemented are: (i) 16 bit XOR: Bit – by--Bit exclusive-OR, ii) 16-bit adder: Addition of integers modulo 216 (modulo 65536), with inputs and outputs treated as unsigned 16-bit integers.

(iii) Modulo 16 multiplier: Multiplication of integers modulo 216+1 (modulo 65537), with inputs and outputs treated as unsigned 16-bit integers, except that a block of all zeros is treated as representing 216. The IDEA algorithm is implemented using ALTERA's QUARTUS tool for VLSI synthesis. The results for implementation are shown in Table 3.2.

Table 3.2 IDEA VLSI area requirement

Algorithm/ Project	Device	Total Logic Elements	Total Pins	Total Memory Bits	Maximum clock Frequency (In MHz)
Single round IDEA/ide1rnd	EP20k15 00EBC6 52-1	6,660 / 51,840 (13 %)	225 / 488 (46 %)	0 / 442,368 (0 %)	4.34
Eight round IDEA /ide8rnd	EP20k15 00EBC6 52-1	43,249 / 51,840 (83 %)	257 / 488 (52 %)	0 / 442,368 (0 %)	3.85

3.2.4. Interfacing Considerations

This block, can be interfaced with NPU in three different ways:

1. Security Co-processor coupled with NPU using Look aside Interface.
2. Designing inline security processor to achieve high data rates.
3. Adding encryption circuitry into the same silicon as the network processor,

As a first phase in designing of security engine, we would like to implement this as a coprocessor working on IDEA algorithm and interfaced with NPU using the Look Aside Interface. The Hardware Working group of Network Processor Forum will be continuously validating this Look-Aside Interface during the development of the Open Interface standard [32]. This encryption co-processor operates using a request/response model and the LA-1 specification.

Existing designs have added security to the network through either a co-processor or an inline security processor. As data rates go up, the co-processor solution becomes less and less practical [25]. Inline security processors can actually scale to the higher data rates but must perform many of the same functions as the network processor. Then based on the system performance and evaluation, we may go for adding encryption circuitry into the same silicon as the network processor.

3.3 PACKET CLASSIFICATION FUNCTION

3.3.1 Introduction

Network processor can be used as a network terminal and router. Network terminals are end points in a computer network, so network terminal is either the source or destination in communication. No

data passes through the network terminal, so network terminal does not need a routing capability. Network terminals have to handle all layers of protocols in the ISO/OSI reference model. It receives more data than it transmits.

Routers forward data to the correct destination and it operates at many layers in the ISO/OSI reference model. Router functionality can be divided into two planes, the data plane and the control plane. Data plane forward the packets to the correct destination at high speed. Control plane handles complex activities, and is slow in speed.

This topic discusses the implementation of three different approaches for implementing packet classifier, viz. Software realization, Hardware realization for router, and Hardware/Software Co-design for network terminal. Software realization is carried out using a C program. The hardware and hardware/software co-design architectures are implemented using VHDL and are synthesized in Quartus 4.1 to get the VLSI area requirement. Further hardware/software co-design is verified using ModelSim 6.0c for a TCP/UDP over IP over Ethernet frame. The frame considered for the implementation is Ethernet frame containing IP or ARP packets, containing TCP or UDP segments. All other packets are discarded. Packet is intended for MAC Address 0x001217834586, IP Address 192.168.0.221 and TCP/UDP Port 137. Figure 3.5 (a), captured by "Ethereal" software shows a sample frame of the same.

3.3.2 Software Realization of Packet Classifier

A classifier needs to compute logical AND of the several conditions specified in the rules. The most conventional and straightforward implementation of a classifier uses a series of tests to examine one header field at a time. As soon as it finds a field that does not match a specified value, the classifier stops and declares that the packet is not a match. Only after all tests are completed the classifier asserts that a packet matches.

A C program has been implemented to realize software classification. It also demonstrates the sequential process flow, i.e. in conventional classification methods the entire received packet is first stored into the memory and then is decoded header by header. We use information regarding the header that is known a priori. We have considered following classification rules (Ethernet/IP/TCP).

Ethernet Header:

1) Destination address field of packet should match with our node address.
2) Length/Type field should contains 0x0800 (IP) or 0x0806 (ARP)

IP Header:

1) Version should contain 4,
2) Header length should be greater than 5
3) TTL field should not be zero
4) Protocol field should contain 0x11 (UDP) or 0x06 (TCP)
5) Header Checksum should match
6) Destination IP address field should match with our IP address field.

TCP Header:

1) Destination Port should match
2) Checksum should match [15]

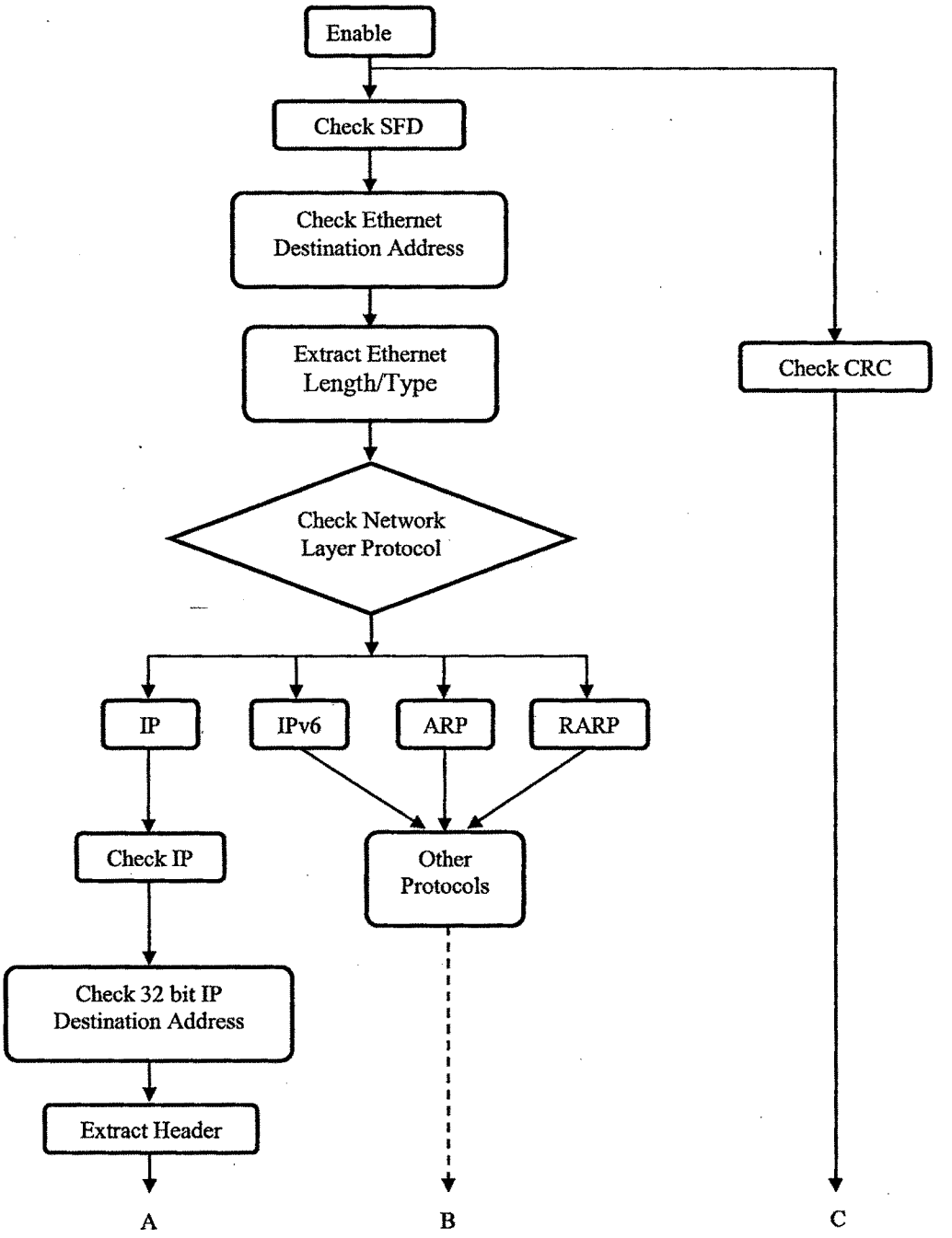

Figure 3.4 Classification Flowchart

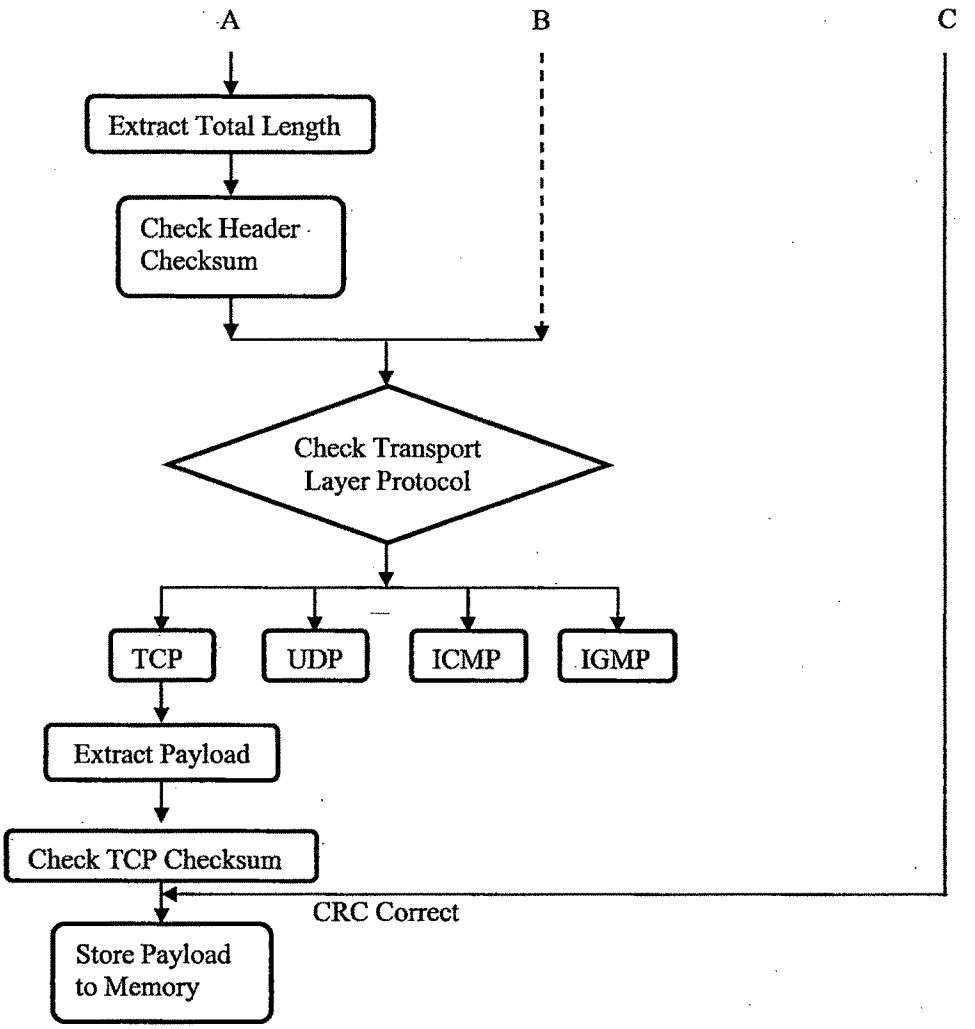

Figure 3.4 Classification Flowchart (Contd.)

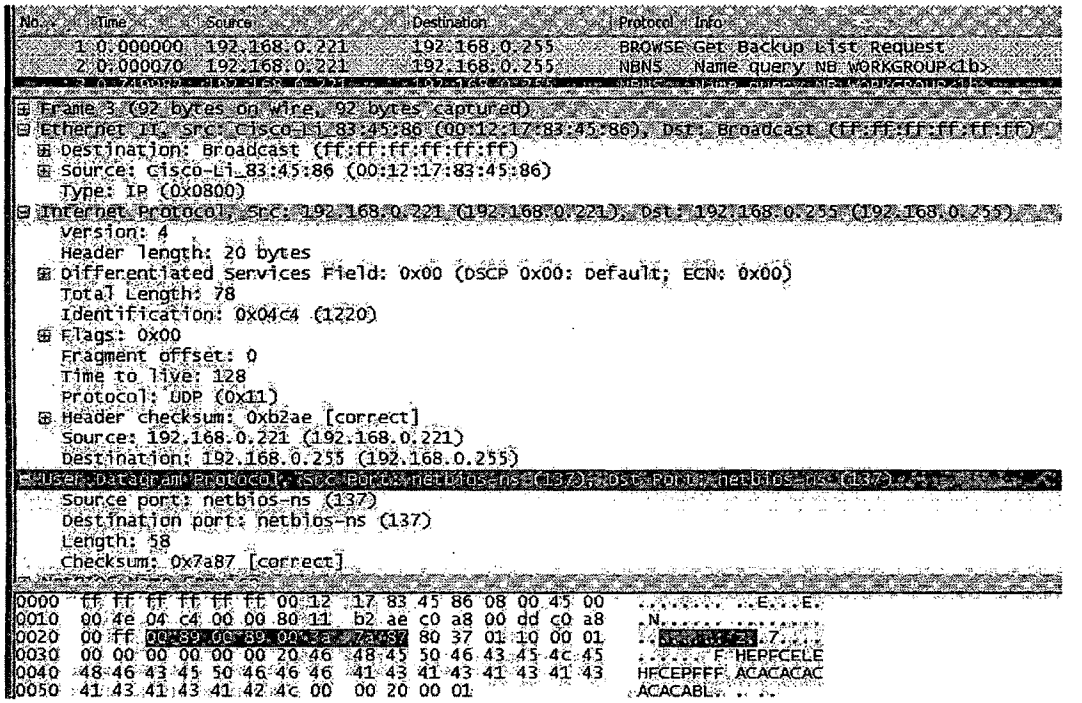

Figure 3.5 (a) Capture by "Ethereal" software. (TCP or UDP packet only)

```
Source Ethernet Address:0 12 17 83 45 86
Network Layer Protocol:IP
IP Version:4
IP Header Length:20 bytes
IP Type of Service:Precedence    Delay    Throughput    Reliability    Monetary
                         0          0          0              0             0
IP Payload Length:78
TTL:128
Received IP Checksum:b2af
Destination IP Address Matched!
Soure IP Address:c0a800dd
Calculated IP Checksum:b2af
IP Checksum Matched!
Transport Layer Protocol:UDP
Source Port Address:137
Destination Port Address:137
UDP Payload:58
UDP Received Checksum:7a87
Received Packet is OK.
```

Figure 3.5 (b) Packet classification by C program

Results of C-Program are as shown in figure 3.5 (b).

3.3.3. Hardware Realization of Packet Classifier

In this realization parallel testing is performed by hardware to avoid testing header fields sequentially; the classifier extracts pertinent fields, concatenates the fields into multi-octet value, and compares the resulting value to a constant. The value of the constant is derived directly form the classification rules. Static classification implemented in hardware performs field extractions and comparisons in parallel, instead of examining each field sequentially. So hardware classifier is faster.

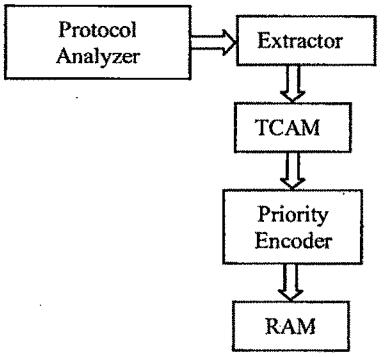

Figure 3.6 Hardware packet classifier

The hardware-based packet classifier is shown in the figure 3.6 which will consist of the protocol analyzer and TCAM based classifier. Protocol Analyzer will receive fixed width (8-bit) stream of packet data from the PHY and output a stream of data into 8, 16, 24 or 32-bit field data along with the 8 bit field type signal. For fields that do not contain 32 bits worth of data, the most significant bits will be padded with zero. The basic function of the extractor is to extract the specified field of the entire frame and put into the header register for packet classification. The output of a protocol analyzer is given to the TCAM based classifier, which consist of priority encoder and action memory. Action memory is a RAM, where address is used to find the action associated with that matching rule. The necessary field values for packet classification from the hardware register are given as search key of the TCAM. The rules of the packet classification are stored in low priority order in TCAM, which will perform the lookup and matching operation and give the output result as a match flag of their memory array. This result will be given to the priority encoder and will resolve the priority or the matching rules and gives the address of the highest priority match to action memory. Here we implement the packet classifier using TCAM which produces multi-match classification in VHDL, by using Quartus II tool of Altera to get the chip area requirement in table 3.3 [15][29][83] .

Table 3.3 Hardware Packet Classifier area requirement

Algorithm/Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
Hardware Packet classifier/ final_block	EP20k1500 EBC652-1	869 / 51,840 (1 %)	169 / 488 (34 %)	2,224 / 442,368 (< 1 %)

3.3.4. Hardware/Software Co-Design of Packet Classifier

The traditional packet classifiers are generally based on general-purpose microprocessor. Since the execution takes place sequentially in such processors they are not able to keep up with the network speed. The architecture implemented here is a dedicated architecture for packet processing which operates in a data-flow fashion directly on the data that is received on the network terminal. Thereby no load and store operations are necessary. So the packets are already processed to a large extent when the payload is written into memory. This saves data memory bandwidth, program memory size, processing time and power consumption. Most of the packets those should be discarded never have to be stored in memory at all.

We have implemented "Linkoping Architecture" designed by Tomas Henriksson, Ulf Nordqvist and Dake Liu at Linkoping University by writing the code ourselves [38][35][41].

The architecture mainly focuses on the terminal protocol processing. Especially it deals with classification of single packet or frame. For task that involves several packets, the implemented architecture only provides supporting functions, such as extraction, classification and information pre-processing. The processor architecture is designed to be re-configurable; hence classification of newer protocols can be carried out, when required. It consists of "Control Unit", "Counter Unit" (named as CCU) and some "Dedicated Modules" (DMs). The Counter Unit keeps track of the input bits arrived (i.e. frame length) and invoke the necessary signal when required, while the Control Unit is a simple microprocessor (FSM based architecture) that generates necessary control signals to drive the Dedicated Modules (DMs). It also interfaces with external application (may be a microcontroller or some other dedicated hardware) and with program and payload memory. CCU handles only macro level processing while the actual processing is carried out in Dedicated Modules(DM), where each module carries out some specific task like IP Length Counter DM, Internet Checksum DM and Payload Memory DM. There may be single or multiple DM of the same type depending on the processing speed or complexity to be handled. The data is fed at different instants to the corresponding module enabled by Control Unit. Hence several modules are working. in parallel and correspondingly the results are produced at different times. These results are used by Control unit to decide whether to discard or process the receiving frame, on the fly.

3.3.4.1 Overview

The architecture discussed here is aimed to be operating on a raw data stream, i.e. data stream arriving on physical layer. The input data is implicitly always loaded into the input buffer, so no load operation is required. Input buffer is basically a 32-bit serial shift register of FIFO type where incoming bit is shifted in at line speed. This calls for hard real-time requirements for rest of the blocks too, since synchronization mismatch will cause loss of data. This is where the architecture differs from the traditional architecture where entire packet is first buffered into the memory and then read while processing; hence not requiring real time constraints. However in such a system, the processor needs

several instructions per data word, e.g. for explicit loads and stores, so the processor must run at a frequency much higher than the frequency of the data words.

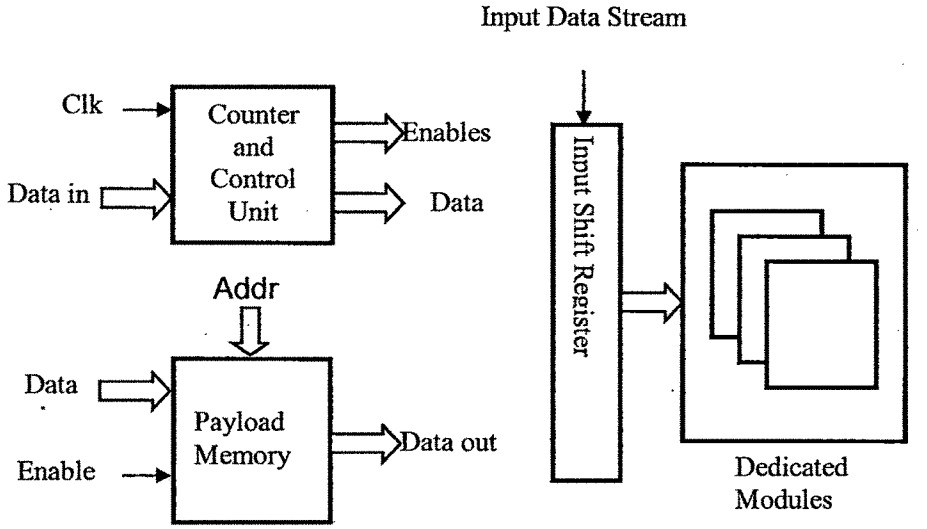

Figure 3.7 Architecture Overview

All hardware modules are acting independently, hence they need to be synchronized. Here since data is processed on the fly the program is required to be highly synchronized with the input data clock by clock. We cannot survive with unpredicted behaviour. Hence all instructions are having fully predictable execution time and no pipeline is used, since any conditional jump may lead to penalty in terms of unpredictable execution time. On the other hand, as there is no pipeline, in order to utilize high frequency clock, time spent for instruction execution must be minimum. This is solved by using three look-up tables inside the processor core. Look-up table have short access time and decoding is kept minimal to reduce time requirement. Instruction set is also optimized for the application and hence code size is very small.

An overview of the entire architecture is as shown in Fig. 3.7. Since there is no internal register file, no intermediate results are stored. Computations that need intermediate results are off-loaded to dedicated modules. These modules communicate with the control unit via synchronous control and status signals. There is one master instruction flow in the core, which can start the execution of the dedicated modules dynamically dependent on the input data. The modules thereafter execute independently of the core instruction flow until a merge occurs when the module have completed execution. Dedicated Modules have generally four control signals two inputs viz. start and stop and two outputs viz. end and correct see Figure 3.8. However, all dedicated modules may not have all the control signals, e.g. some modules do not have any external stop signal to indicate when to stop the process, similarly, some modules have nothing to say about correctness of received data. All the control signals are connected to the PP, which generates or processes the respective control signal.

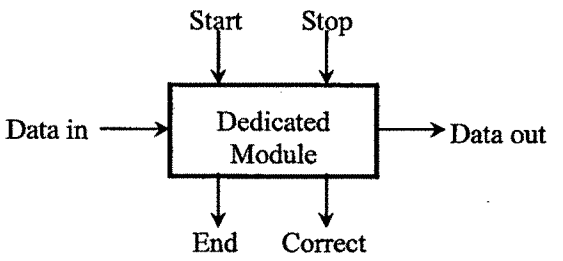

Figure 3.8 Dedicated Module Overview

3.3.4.2 PP Architecture

Protocol Processor is implemented with an aim to process the incoming packet before they are stored in a memory. This is beneficial for several reasons. If it is discovered while processing the packet header, that the packet is not interesting for the terminal, then the packet can be discarded before it is fully stored in memory and further processing can be cancelled. The processor then can go into sleep mode and wake up when the next packet arrives on the input port. This saves energy as well as memory bandwidth. Since the processing is already taken care of when the packet is stored in the memory it can be directly delivered to the application just by passing a pointer. Initially the architecture is implemented for a small set of network protocols. However the architecture is made such that later on, by making only minor changes other protocols can be processed. The most important design goal was to be able to execute if-else and switch-case statements in one clock cycle, no matter which branch was taken. These are used e.g. in protocol demultiplexing and Address Resolution Protocol (ARP) handling, as shown in figure 3.9.

```
//Process Network Layer Protocol
switch(NtwrkPro)
{
        case IPv4:      //IPv4 packet 0x0800
                CheckIP();
                break;
        case IPv6:      //IPv6 packet 0x86DD
                CheckIPv6();
                break;
        case ARP:       //ARP packet 0x0806
        case RARP:      //RARP packet
                        0x8035
                CheckARP();
                break;
        default: //Handle unknown packet
                Unknown();
                break;
}
```

```
switch(ARPOperation)
{
        case ARPRequest:
                --------
                break;
        case ARPReply:
                --------
                break;
        case RARPRequest:
                --------
                break;
        case RARPReply:
                --------
                break;
        default:Flag.Discard=1; //Discard
                                //Packet
}
```

Figure 3.9 C Code Example using switch-case statements

Internal architecture of the protocol processor is as shown in Fig. 3.10. There are other blocks also used for communication with dedicated modules and with external input/outputs.

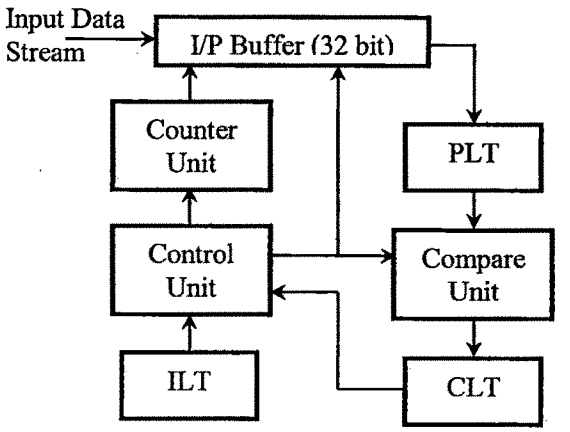

Figure 3.10 Protocol Processor Architecture

3.3.4.2.1 Input Buffer

It is a 32-bit serial shift register in which one bit is shifted in every clock cycle. It is basically a FIFO, in nature and hence it is necessary that data within it is processed or used at proper instants before it is pushed out. As shown in figure 3.29 *InputStream* is input port for the incoming serial data. Control signal *EnOut* is used to enable the 32-bit output data port. *Ready* is an output signal that is used to indicate that a valid signal is available on output port, *OutData*. When data is read *EnOut* signal is pulled down, which causes *Ready* signal to go low. Table 3.4 shows the VLSI area requirement of input buffer.

Table 3.4 Input buffer area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
InputBuf	EP20k1500EB C652-1	64 / 51,840 (< 1 %)	37 / 488 (7 %)	0 / 442,368 (0 %)

3.3.4.2.2 Instruction Look-Up Table (ILT)

The ILT stores the main program. Each instruction is 24-bit wide and ILT can hold up to 256 instructions. Hence it requires total 6144 bits of memory. The contents of ILT are accessed by program counter (PC), which is connected to the address bus of ILT. All three memories (i.e. look-up tables) can be programmed externally by initializing PP in "program" mode. As shown in figure 3.30 *Addr* (8-bit) signal specifies the memory location to be fetched. It is controlled by PC of Control unit in run mode and by external device in program mode. *WrEn* signal when high enables the memory write function (i.e. in program mode), while *Ready* goes high whenever *WrEn* is low and a valid data

is available on *OutData* (24-Bit). This data, which is an instruction, is then used by Control unit for further processing.

Table 3.5 Instruction Look-Up Table area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
ILT	EP20k1500EB C652-1	1 / 51,840 (< 1 %)	59 / 488 (12 %)	6,144 / 442,368 (1 %)

3.3.4.2.3 Parameter Look-Up Table (PLT)

PLT contains the parameters, which are used for comparison with the incoming data, e.g. MAC Destination address, Ethernet Type field, IP destination address, UDP Port number, etc. It consists of 16 pages, each having four 32-bit parameters. This requires a total of 2048 bits of memory. The input to PLT is 4-bit address line pointing to the specific page, while output is four 32-bit parameters of that page, i.e. 128-bit data output, which acts as input to compare unit. As shown in figure 3.31, control signals are functionally same as for ILT. *OutData* (128-bit) is connected to Compare unit since data stored here is a compare parameter.

Table 3.6 Parameter Look-Up Table area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
PLT	EP20k1500EB C652-1	13 / 51,840 (< 1 %)	167 / 488 (34 %)	2,048 / 442,368 (< 1 %)

3.3.4.2.4 Control Look-Up Table (CLT)

The CLT contains jump addresses required to be loaded in PC after the successful comparison match occurs. The CLT has 8 pages corresponding to the first 8 pages of PLT, each having four 8-bit jump locations. This requires a total of 256 bits of memory. The reason for having more pages in the PLT than in the CLT is that the CLT is only used for CMP and CPS instructions, with the jump bit set to 1, while PLT is also used for CMP and CPS instructions with the jump bit set to 0 and for JMP instructions with type 10. Inputs to the CLT are the 3 least significant bits of the PLT address (pointer) and 4-bit result from the compare unit. The pointer selects a page and the 4-bit result from the comparators selects a value within that page. This value simply modifies the PC value to decide where the control should go. In case comparison has failed, comparator output will be zero and PC

will be loaded with the next instruction location only. PLT and CLT as combined, help to manage multiple comparisons in a single clock cycle. As shown in figure 3.32, Control signals are functionally same as for ILT. *Addr* (7 bit) is formed of 3 least significant bits of the PLT address (pointer) and 4-bit result from the compare unit. The pointer selects a page and the 4-bit result from the comparators selects a value within that page. *OutData* (8-bit) is connected to Control unit, which is used to modify PC, if specified.

Table 3.7 Control Look-Up Table area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
CLT	EP20k1500EB C652-1	3 / 51,840 (< 1 %)	26 / 488 (5 %)	256 / 442,368 (< 1 %)

3.3.4.2.5 Compare Unit

Compare unit consists an array of four comparators each being capable of comparing 32-bit data as shown in figure 3.11. It makes it possible to execute the complete switch-case network statements in one clock cycle. At present only 4 cases and 1 default case can be handled, since only 4 comparators have been used. However this can be increased by increasing parallel comparators. During compare operation, the four parameters of the selected page from PLT are fed to the array as one data and the content of input buffer as another data. The result of the comparators is a 4-bit array, which is used as an input to the CLT.

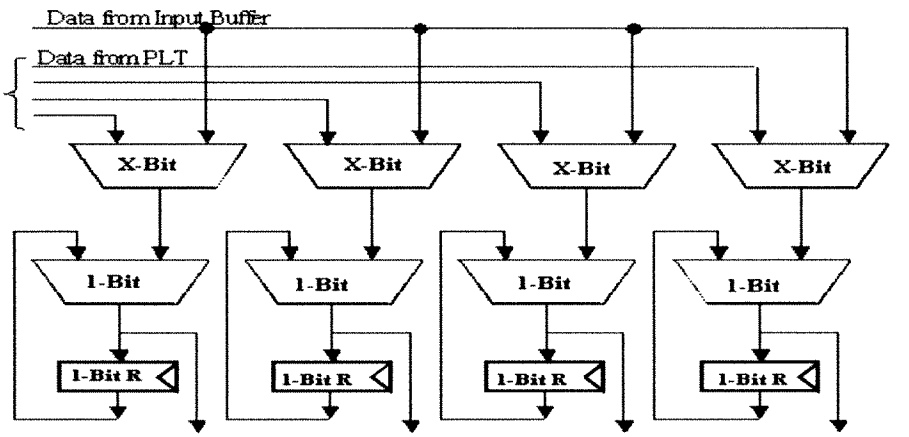

Fig.3.11 Compare Unit

Three control signals are used to complete the entire compare operation viz. *Width*, *Offset* and *New*. The *Width* bits (2-bit) carried by an instruction are used to mask the data to the desired width X (4, 8, 16, or 32 bits). Hence a flexible comparison can be carried out. The *Offset* bit decides where to mark the incoming bit stream as starting point for 32-bit data in the input buffer, which makes it possible to

easily extract the desired data for comparison. The comparison is controlled by the *New* bit, which specifies if the comparison should start from scratch or if it is a continuation of a previous comparison. The intermediate result of comparison is always implicitly stored within the compare unit for this purpose. For example for MAC Destination addresses a 48-bit comparison can easily be accommodated for in this way. As shown in figure 3.33, *InData* (32-bit), received from Input Buffer, is compared with four 32-bit data (4x32=128 bit), *PLTData*, received from PLT. *En* enables the Compare unit to perform comparison. *Width* specifies whether to carry out 4, 8, 16 or 32 bit comparison. *NewComp* when high indicates that comparison should start from scratch else it is a continuation of a previous comparison. If comparison fails then *CompFail* signal will go high and data on *OutData* will be "0000". However in case of successful comparison *OutData* specifies with which parameter out of the four parameters, successful comparison is occurred. *Ready* signal goes high when comparison is completed.

Table 3.8 Compare unit area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
Compare	EP20k1500EB C652-1	149 / 51,840 (< 1 %)	170 / 488 (34 %)	0 / 442,368 (0 %)

3.3.4.2.6 Counter Unit

Counter Unit is a 12-bit counter that keeps track of number of bits received in Input Buffer. It generates a signal that is used to indicate when Input Buffer should output a valid 32-bit data. When *Offset* bit in the instruction is low, Counter Unit counts for incoming 32 bits thereon, while in other case, it counts the incoming 16 bits and makes signal high to enable Input Buffer to output a valid 32-bit data, see Fig. 3.12.

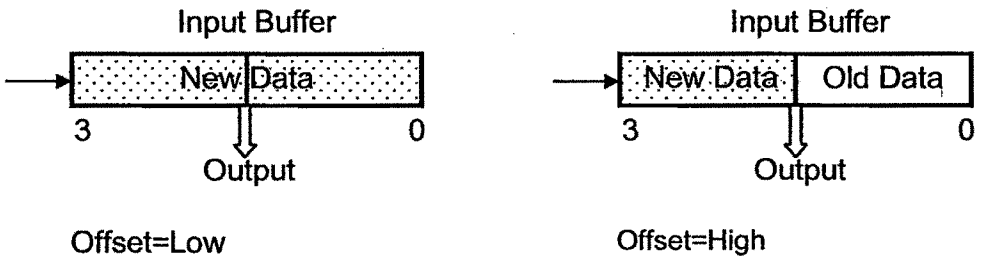

Figure 3.12 Effect of *Offset* on Output from Input Buffer

It also generates necessary clock input for the Control unit. It can be configured to generate lower clock rate, i.e. in factor of incoming bit rate, for the Control unit since Control unit does not require high clock rate for processing. *CntrlClk* is the clock generated for Control unit. *Offset* signal is received from Control unit, which indicates where to mark the incoming bit stream as starting point

for 32-bit data in the Input Buffer. *IPBufEnOut* signal enables Input Buffer to output a valid data on its output port. Input Buffer then raises *IPBufReady* to indicate that data is now available, correspondingly Counter unit raises *DataReady* signal, which is connected to Control unit.

Table 3.9 Counter unit area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
Counter	EP1S10F484C 5	13/10570 (< 1 %)	9/336 (2 %)	0

3.3.4.2.7 Control Unit

The Control unit provides necessary signal interactions to make the data flow properly through other blocks and thereby performs the expected functions. Control unit architecture is Finite State Machine (FSM) based that causes all appropriate signal values to be updated based on current state and input signals and produces a next state for the state machine. It maintains the main flow of program and synchronizes all the dedicated modules by enabling and utilizing their results at right instants.Since all instructions are implemented using FSM logic, they are absolutely predictable. No pipeline is used since a jump can lead to pipeline penalty and hence unpredictable behavior. It is not possible to simulate Control unit as a separate entity, since it requires data from many other units including the ILT from where the instructions is to be fetched, hence no separate simulation report is shown.

3.3.4.2.8 Inputs and Outputs

It consists of 19 inputs and 10 general-purpose outputs. Outputs are mainly used to enable the DM, while inputs are used for checking status and results of the DM. The inputs are used for two purposes, jumps and waits. The program execution can be conditionally halted until a certain input pattern occurs. Likewise a jump can be conditionally executed dependent on the inputs. For the conditional jumps only 9 of the 19 inputs can be used.

3.3.4.3 Dedicated Modules

The Protocol Processor core that has been described so far only handles comparison and decision-making part of packet reception processing. The other tasks such as payload storage, IP length counting and checksum calculation are handled by Dedicated Modules. These modules work as standalone unit and hence can be added or removed as per requirement (i.e. for different protocol, different modules can be used).

The reason for dividing the processing is that tasks like comparison (or field matching) and decision-making uses only the packet headers, while checksum calculation and payload storage uses the whole

packet. Here, since core and DMs uses the same data at the same time, the architecture can be called as MISD (Multiple Instruction Single Data) type architecture.

The program flow in different modules is also different, e.g. in processor core field matching consist of almost only if-then-else or switch-case statements, checksum calculation consists of bitwise operations and payload storage implies storing the payload in the correct memory location based on the result of comparison operations, so that later on application can access the payload directly.

Currently only three modules are discussed and implemented as required by the TCP/UDP over IP over Ethernet, however for other protocols other modules are required and can be added.

3.3.4.3.1 Payload Memory

To store the payload data various methods are possible. One alternative is to a shared off-chip memory on the motherboard, but this will create unwanted interrupts for host processing and hence leads to performance degradation. A better option is to use on-chip memory that can be used to store payload of only valid (or accepted) packets received in the terminal. From this memory, payload data can be accessed by the main host memory, as shown in figure 3.13. The payload memory should be of around 1MB, so that it can store at least 100ms of traffic in 10GB/s networks. In the on-chip memory, an alternative is of course to store all incoming data until we receive a discard signal from Control Unit. However, the problem is that we do not want to send the payload data to the host memory before we know if it should be discarded or not. The purpose of Protocol Processor is to offload and accelerate the application-processing running on the host.

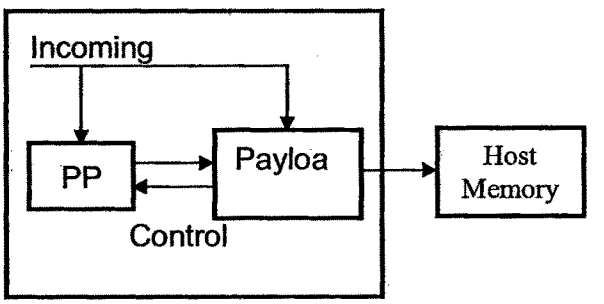

Figure 3.13 Payload Memory Interfacing

To solve the problem, FIFO is inserted before payload memory. This FIFO will hold the packet until the PP decides if it should be discarded or accepted. The length of FIFO can be optimized to find the most power efficient architecture. Figure 3.14 shows the idea.

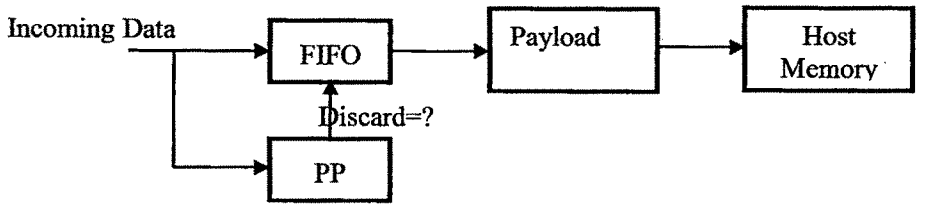

Figure 3.14 Using FIFO Enhance Performance

As shown in figure 3.35,*Start* pulse enables FIFO to store data, while *Stop* pulse stops FIFO write operation. If *Discard* signal is asserted high before *Stop* pulse is applied, FIFO is flushed. If *Discard* is low and *Stop* pulse is applied, then FIFO content is considered as valid data and is transferred to payload memory. Content of payload memory can be accessed by enabling *Read* signal. *Ready* signal become high when a valid data is available on *OutData*.

Table 3.10 Payload Memory area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
PayloadMemr	EP20k1500EB C652-1	60 / 51,840 (< 1 %)	80 / 488 (16 %)	40,960 / 442,368 (9 %)

3.3.4.3.2 IP Length Counter

IPv4 packet is of variable length. Hence it is necessary to count the received bytes based on header information. IP Length Counter DM is a counter that keeps track on number of bytes received. Besides the processor core this module also interfaces with Internet Checksum, since IP data length information is required for checksum calculation. As shown in figure 3.36, *Start* pulse enables IP Length Counter, however no external stop signal is required in this case. *Ready* signal is asserted high when header is finished.

Table 3.11 IP Length Counter area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
IPLengthCounter	EP20k1500EB C652-1	42 / 51,840 (< 1 %)	36 / 488 (7 %)	0 / 442,368 (0 %)

3.3.4.3.3 Internet Checksum

The checksum calculation is traditionally carried out in host processor within the operating system kernel, before the payload data is handed over to the application. However here this task is carried out by a separate module, so as to increase parallelism and hence performance. Two separate DMs are used, one is used for IP checksum calculation and other is used for TCP/UDP checksum calculation. This module has relatively straightforward task for most packets. For non-fragmented packets, the incoming 32-bit data is split into 16 bit words, adds the first and second part and then adds that sum to the accumulated sum, as shown in figure 3.15.

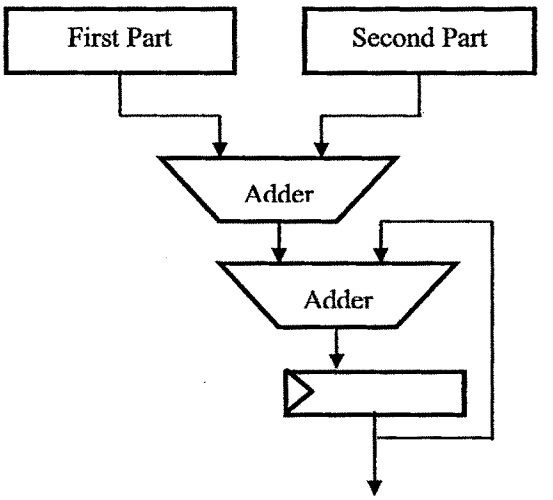

Figure 3.15 Internet Checksum Calculation

One important task in TCP/UDP checksum is to handle the pseudo header from the IP header that must be included in the calculation. The pseudo header that is included in calculation for IPv4 and IPv6 is as shown in figure. 3.16 and figure 3.17 respectively.

The computation gets a lot more complicated when fragmented packets are considered. Fragmented packets are not dealt with at this stage.

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31		
Source IP Address		
Destination IP Address		
0	IP Protocol	Total Length

Figure 3.16 Pseudo Header (IPv4) used in TCP/UDP Calculation

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	
Source IP Address	
Destination IP Address	
Upper Layer Packet Length	
0	Next Header

Figure 3.17 Pseudo Header (IPv6) used in TCP/UDP Calculation

Table 3.12 shows the area requirement of Internet checksum module. As shown in figure 3.37, *Start* pulse enables Internet Checksum to perform checksum operation. It continues this operation until *Stop* signal is applied form IP Length Counter. *Ready* signal is asserted high when checksum operation is finished. Further, *Chksum* signal is asserted high if the checksum matches.

Table 3.12 Internet Checksum area requirement

Project	Device	Total Logic Elements	Total Pins	Total Memory Bits
InternetChecksum	EP20k1500EBC652-1	215 / 51,840 (< 1 %)	38 / 488 (7 %)	0 / 442,368 (0 %)

3.3.4.3.4 CRC Checksum

The most computational demanding task while processing TCP/UDP over IP over Ethernet/ATM is undoubtedly the link layer checksum calculation. In simple RISC machine a 1500 byte long frame require almost 44000 (non optimized) instructions to process only the CRC checksum [72]. Therefore it is necessary to have a separate dedicated hardware to carry out this task. Several architectures are available for CRC calculation; however parallel implementation is most suitable to tackle with gigabit rate data. CRC checksum module is not implemented at this stage and while simulation it is assumed that CRC is correct.

3.3.4.4 Instruction Set

The PP architecture has only 6 instructions. These instructions are optimized for processing of Ethernet, ARP, IP and UDP. For other or more general protocol stacks small modifications in the instruction set may be expected.

3.3.4.4.1 Instruction Format

The general format of the instruction is as shown in Figure 3.18. It is 24-bit wide with 4 bits used to specify the instruction code, which allows for 16 possible instructions. Since only six instructions are

used, more instructions and future changes can be accommodated. The bit next to the *Code*, i.e. *Offset*, decides where to mark the incoming bit stream as starting point for 32-bit data in the input buffer. A '1' in this bit means extract data when the counter value in counter unit is 16, while '0' means extract data when counter value is 32, i.e. 32 latest bits are received.

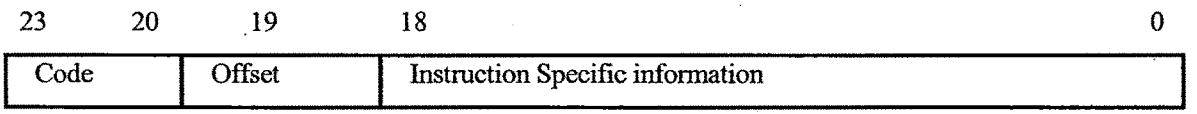

Figure 3.18 Instruction Format

3.3.4.4.2 Synchronization Instruction (SYN)

It is used for synchronization purpose with the incoming bit stream. If some fields are not included in the processing then they should ignored. This is done by using SYN instruction. Bit 19 acts as *Offset* field that is discussed in previous section. Positions 18 through 0 are unused.

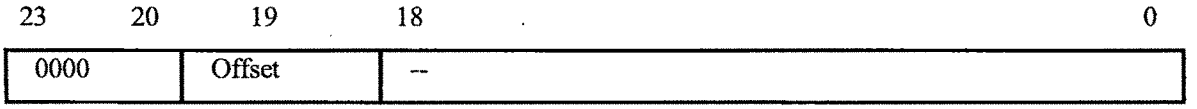

Figure 3.19 SYN Instruction Format

3.3.4.4.3 Compare Instruction (CMP)

Compare instruction uses the most advanced feature of PP. This instruction is used to perform compare operation. It makes it possible to execute the complete switch-case network statements in one clock cycle. At present only 4 cases and 1 default case can be handled, since only 4 comparators have been used. However this can be increased by increasing parallel comparators. It makes use of values stored in PLT and CLT.

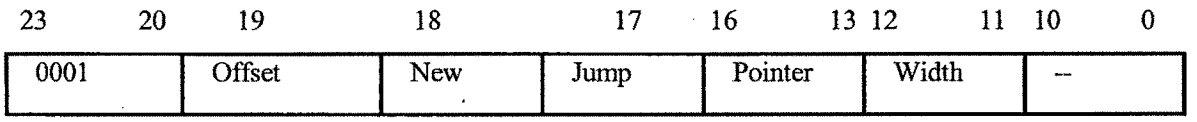

Figure 3.20 CMP Instruction Format

Format of CMP instruction is as shown figure 3.20. The field, *New*, specifies if the comparison should start from scratch or if it is a continuation of a previous comparison. A '1' at this field implies new comparison, while a '0' indicates continued comparison. The intermediate result of comparison is always implicitly stored within the compare unit for this purpose. E.g. for MAC Destination addresses a 48-bit comparison can easily be accommodated for in this way. The *Pointer* field indicates the page address for PLT. Since it is 4-bit wide maximum 16 pages can be accessed from the PLT. The

positions 15 through 13 are used as page address for CCB also, causing maximum of 8 pages access. A '1' in *Jump* field indicates that a jump should be performed at a match. The jump address used will be taken from CCB. A '0' in this field implies that simply store the comparison result in compare unit, which may be used for next continued comparison. The *Width* bits carried by an instruction are used to mask the data to the desired width (4, 8, 16, or 32 bits), i.e. it decides the width of comparison. Positions 10 through 0 are unused.

3.3.4.4.4 Set Instruction (SET)

This instruction is used to set the internal and external outputs of PP core. The internal outputs of the core are used as control signals for the dedicated modules (start and stop signals), while the external outputs are general-purpose and can be used for other applications such status indication. The positions 9 through 0, *Output Bitmap*, specify which output should be high or low. Positions 18 through 10 are unused.

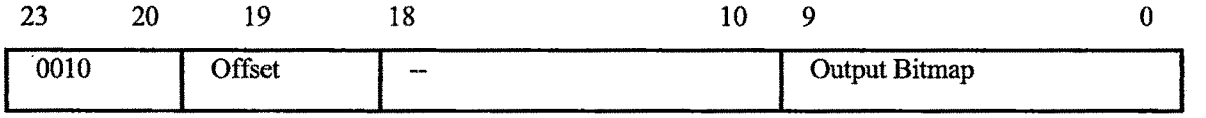

Figure 3.21 SET Instruction Format

3.3.4.4.5. Compare and Set Instruction (CPS)

For efficient execution and synchronization purpose, it is sometime necessary to perform comparison and set operation simultaneously. This is done by CPS instruction. Position 10 is unused.

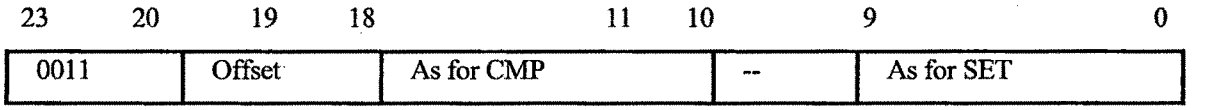

Figure 3.22 CPS Instruction Format

3.3.4.4.6 Jump Instruction (JMP)

General format of Jump instruction is as shown in Fig. 3.23. Total three types of jump are possible, as shown in table 3.13. Field *Type* specifies the type of jump to be performed. The positions 7 through 0, *Absolute Address*, specify the exact location where program counter should jump. Since it is an 8-bit wide field, entire ILT is covered, i.e. 256 words. Positions 16 through 8 are type specific information.

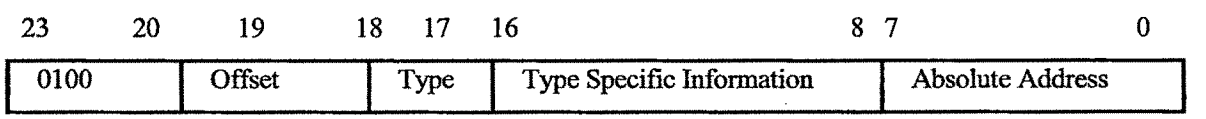

Figure 3.23 General Format of JMP Instruction

Table 3.13 Possible Types of Jump

Type Field	Jump Type
00	Unconditional
01	Conditional, dependent on the inputs
10	Conditional, dependent on the comparison results

Figure 3.24 shows detailed format for all three types of jump. In case of unconditional jump (Type 00), positions 16 through 8 are unused. For conditional jump dependent on inputs (Type 01), positions 16 through 8, *Input Bitmap*, specify the required input bit pattern to perform jump. *Input Bitmap* corresponds to 0-8 inputs i.e. only 9 inputs are used. For conditional jump dependent on comparison results (Type 10), fields *New*, *Pointer* and *Width* have same functionality as for CMP instruction. Position 8 is unused. On successful comparison jump is carried out to the location specified by *Absolute Address*.

23 20 19 18 17 16 8 7 0

0100	Offset	00	--				Absolute Address
0100	Offset	01	Input Bitmap				Absolute Address
0100	Offset	10	Pointer	Width	New	-	Absolute Address

Figure 3.24 JMP Instruction Format

3.3.4.4.7 Wait Instruction (WAT)

The wait instruction is used for synchronizing the program flow with external events, e.g. the arrival of a specific header field or the completion of a dedicated module task. The format of WAT is shown in Fig. 3.25. It simply defers the updating of the program counter until the inputs match positions 18 through 0, *Input Bitmap*, in the instruction word. For every '1' in the *Input Bitmap*, the corresponding inputs must also be '1' until the program counter is advanced to the next instruction.

23 20 19 18 0

0101	Offset	Input Bitmap

Figure 3.25 WAT Instruction Format

3.3.5 Demonstration Example of execution of Processor

The best way to explain the execution of processor is by an example program. The program is intended for Ethernet II frame containing IP/UDP or ARP packets. All other packets are discarded. Packet is intended for MAC Address 0x001217834586, IP Address 192.168.0.221 and UDP Port 137.

3.3.5.1 Program Code

As mentioned in previous chapter, ILT block contains the program that controls the main flow of entire execution. In following sections ILT, PLT and CLT contents are shown and section 3.3.5.2 program execution is explained line by line.

3.3.5.1.1 ILT Content

The ILT contents are as shown in Fig. 3.26. This is the main code that decides the flow of entire architecture. The 24-bit instructions (in hex format) are shown along with respective assembly representation.

3.3.5.1.2 PLT Content

The PLT contents are as shown in Fig. 3.27. It consists of 16 pages, each containing four 32-bit parameters. Page 0 contains the Ethernet codes for IP (0x0800) and ARP (0x0806), while page 1 contains the protocol value for UDP (0x11). Page 8 contains the first part of the hardware address (MAC Address) and page 9 the second e.g.. MAC Address 0x001217834586 is stored as 0x00121783 as first part and 0X00004586 as second part. Page 10 contains the acceptable IP destination addresses and finally page 11 contains the acceptable UDP destination ports. The other pages are not used by the example program.

```
0x580001        -- 0: WAT 1,(0)
0x1D1800        -- 1: CMP 1,1,0,8,3
0x4D3005        -- 2: JMP 1,10,9,2,0,5
0x200040        -- 3: SET 0,(6)
0x400000        -- 4: JMP 0,00,0
0x000000        -- 5: SYN 0
0x080000        -- 6: SYN 1
0x3E101A        -- 7: CPS 1,1,1,0,10,(1,3,4)
0x400003        -- 8: JMP 0,00,3
---------- Code for IP ----------
0x200024        -- 9: SET 0,(2,5)
0x000000        -- 10: SYN 0
0x000000        -- 11: SYN 0
0x3E2806        -- 12: CPS 1,1,1,1,01,(6)
0x400003        -- 13: JMP 0,00,3
0x000000        -- 14: SYN 0
0x080000        -- 15: SYN 1
0x4DA812        -- 16: JMP 1,10,11,2,0,18
0x400003        -- 17: JMP 0,00,3
0x580004        -- 18: WAT 1,2
0x4D7414        -- 19: JMP 1,10,11,10,1,20
0x400003        -- 20: JMP 0,00,3
0x200020        -- 21: SET 0,(1)
0x50001A        -- 22: WAT 0,(4,3,1)
0x22E018        -- 23: JMP 0,01,(7,6,5),24
0x400003        -- 24: JMP 0,00,3
0x200080        -- 25: SET 0,(7)
0x400000        -- 26: JMP 0,00,0
---------- Code for ARP ----------
0x280002        -- 27: SET 1.(1)
0x500002        -- 28: WAT 0,(1)
0x22101F        -- 29: JMP 0,01,(4),31
0x400003        -- 30: JMP 0,00,3
0x200080        -- 31: SET 0,(7)
0x400000        -- 32: JMP 0,00,0
```

Figure 3.26 Contents of ILT

--Page 0	--Page 6	--Page 12
0x00000800	0x00000000	0x00000000
0x00000806	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000
--Page 1	--Page 7	--Page 13
0x00000011	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000
0x000J0000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000
--Page 2	--Page 8	--Page 14
0x00000000	0xFFFFFFF	0x00000000
0x00000000	0x00121783	0x00000000
0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000
--Page 3	--Page 9	--Page 15
0x00000000	0x0000FFFF	0x00000000
0x00000000	0X00004586	0x00000000
0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000
--Page 4	--Page 10	
0x00000000	0XC0A80001	
0x00000000	0XC0A800DD	
0x00000000	0XFFFFFFF	
0x00000000	0x00000000	
--Page 5	--Page 11	
0x00000000	0X00000089	
0x00000000	0X000007E9	
0x00000000	0x00000000	
0x00000000	0x00000000	

Figure 3.27 Contents of PLT

3.3.5.1.3 CLT Content

The CLT contents are shown in Fig. 3.28. It consists of 8 pages, each containing four 8-bit parameters, representing the absolute jump location. CLT contents are in correspondence to the PLT contents, e.g. Page 0 contains the corresponding absolute jump addresses for the Ethernet codes and page 1 contains

the corresponding absolute jump addresses for the protocol values.

--Page 0	--Page 2	--Page 4	--Page 6
09	00	00	00
1B	00	00	00
00	00	00	00
00	00	00	00
--Page 1	--Page 3	--Page 5	--Page 7
0E	00	00	00
00	00	00	00
00	00	00	00
00	00	00	00

Figure 3.28 Contents of CLT

3.3.5.2 Program Execution

From the beginning, instruction 0 waits for input 0, which indicates packet start and also implicitly triggers Ethernet CRC calculation dedicated module, as shown in figure 3.38. Instruction 1 then compares the first 32 bits of the Ethernet destination address with the acceptable parameters from PLT page 8.

Instruction 1: 0x1D1800, CMP 1,1,0,8,3 (As shown in figure 3.39)

CMP Inst. Format

0001	Offset	New	Jump	Pointer	Width	--

Decoding: Comp. instr., Offset = 1, New = 1, Jump = 0, Pointer = 8, Width = 3

Figure 3.39 shows execution of comparison instruction with the above parameters. Since New is set (see ***newcomp*** of section Compare) a new comparison is performed. ***indata*** represents the data as received on physical layer. ***addr*** in PLT section is the value represented by Pointer in the instruction. Hence the data from the page 8 is fetched. Data of page 8 in PLT are as shown below. ***pltdata*** in Compare section shows the data as fetched from PLT. ***width*** is 3, hence 32 bit comparison is carried out. The ***indata*** matches with the second data of the page (0x00121783), hence the ***outdata*** value is 2. ***ready*** signal goes high as soon as comparison is over. Since comparison is successful ***compfail*** is low. The result of the comparison is only stored locally in the compare unit since the jump bit is set to 0.

PLT Page 8 data:

0xFFFFFFF, 0x00121783, 0x00000000, 0x00000000

Instruction 2: 0x4D3005, JMP 1,10,9,2,0,5 (not shown in fig.)

Continues the comparison, since the new bit is set to 0. Here only 16 bits are used and compared to PLT page 9, since the width code is 10. If any match occurs, i.e. the Ethernet frame is destined for the host, a jump is done to instruction 5.

PLT Page 9 data:

0x0000FFFF, 0X00004586, 0x00000000 , 0x00000000

JMP Inst. Format (Conditional Jump)

0100	Offset	10	Pointer	Width	New	-	Absolute Address

Instructions 5 and 6 are SYN to align the data flow processing (in this example we do not care about the Ethernet source address). Instruction 7 compares the Ethernet type field with PLT page 0 and uses the jump addresses from CLT page 0. So if the type field is 0x0800 a jump is done to instruction 9, otherwise, if it is 0x0806 a jump is done to instruction 27. If there is no match the execution continues with instruction 8. At the same time outputs 1, 3, and 4 are set. These are used to trigger the start of dedicated modules for payload storage, IP header checksum calculation, and UDP checksum calculation.

Continuing the execution at instruction 9 (assuming that the arriving packet is IP) outputs 2 and 5 are set. Output 2 triggers the length counter module for IP and output 5 stops the payload storage. For an IP/UDP packet only the UDP payload should be stored. For an ARP packet on the other hand, the whole Ethernet payload is stored, since the data is needed by the microcontroller in order to compile the ARP reply. Instruction 10 and 11 are again for data aligning. Instruction 12 checks the protocol field in the IP header and if it is 0x11 (UDP) a jump is done to instruction 14. In instruction 16 IP destination address is compared with PLT line 10. For a correct packet, then the UDP port is checked by instruction 19 and instruction 21 triggers the payload storage to start again. Instruction 18 waits for the input 1 to be high indicating that IP header is finished. After the header has been processed, the processor waits for inputs 2, 3, and 0 in instruction 22. These three inputs indicate that the IP header checksum module, the UDP checksum module and the Ethernet CRC calculation module have completed their computations. In instruction 23 a conditional jump is done on inputs 5, 6, and 4. These are all 1 if the just mentioned modules have received correct checksums. Then finally, the reception of a valid IP packet is acknowledged through output 7 in instruction 25 and instruction 26 jumps back to

instruction 0 in order to wait for the next packet.

If the Ethernet code would be ARP, instructions 26 to 31 would have executed in a similar manner. Whenever the received packet does not match the requirements the packet is discarded and the processor waits for the next packet. This is done by a jump to instruction 3, which sets output 6; discard payload, and then instruction 4 jumps back to instruction 0.

3.3.6 Comparison

Table 3.14 : Comparison Table

Parameter	Software based classifier	Hardware based Classifier for router		Hardware/ Software co-design Classifier for network terminal
Execution Type	Sequential	Parallel		Parallel
Architecture	Pentium processor	FPGA based		FPGA based (PP+DM)
Size	1302 lines in C Code	Logic Elements	869	957
		Memory Bits	2224	49,408 (including Payload Memory)
Flexibility on Protocol	Yes	Partial, by adding/ updating the protocol RAM		Yes, by reconfiguring memory and adding/updating Dedicated Modules

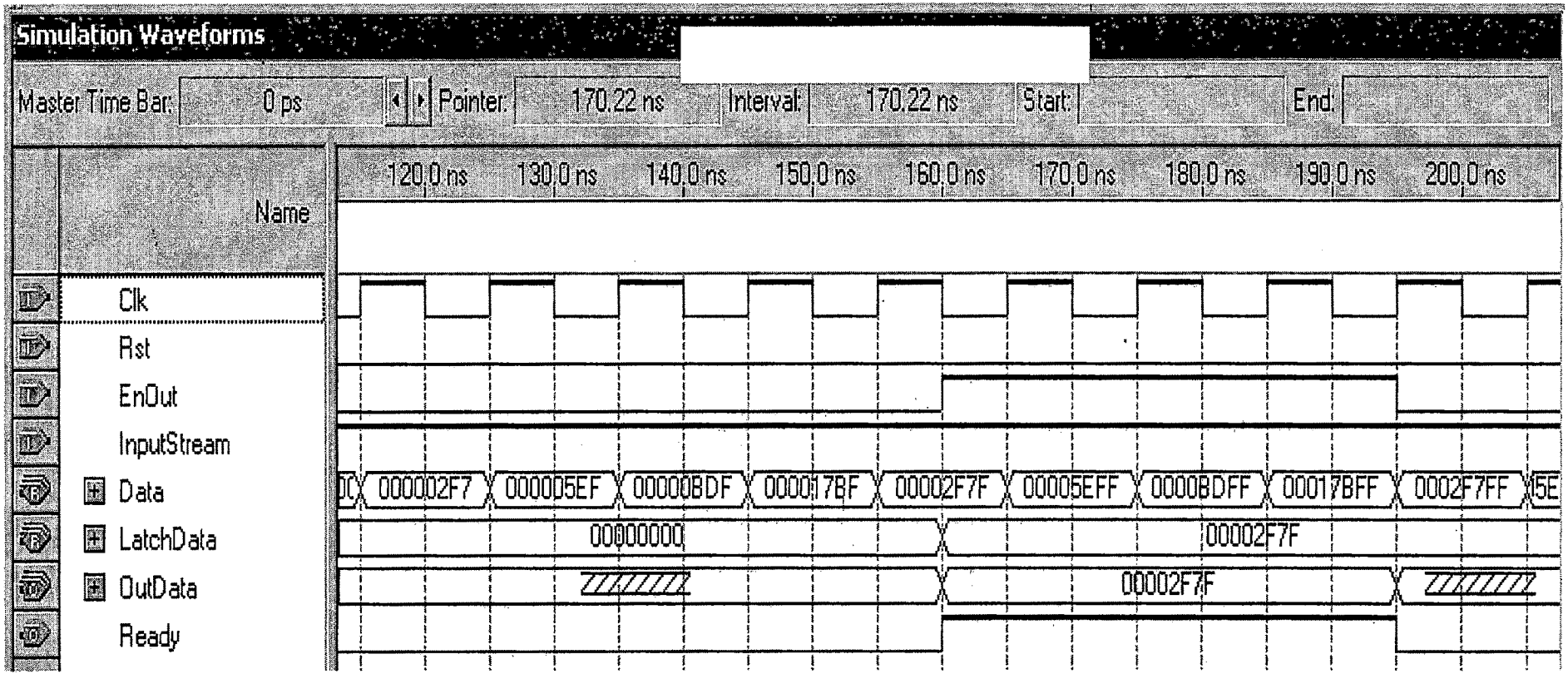

Figure 3.29 Simulation result of Input buffer

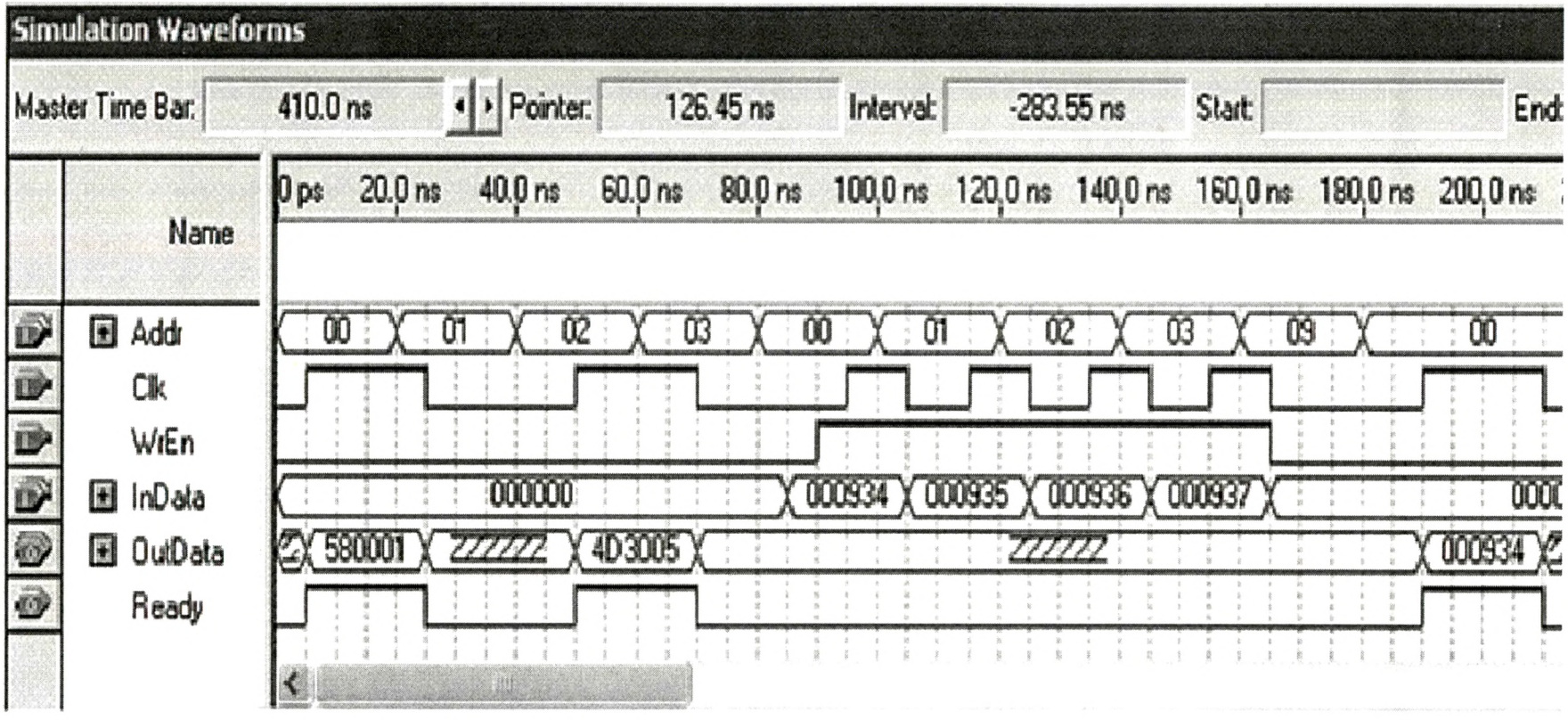

Figure 3.30 Simulation result of I LT

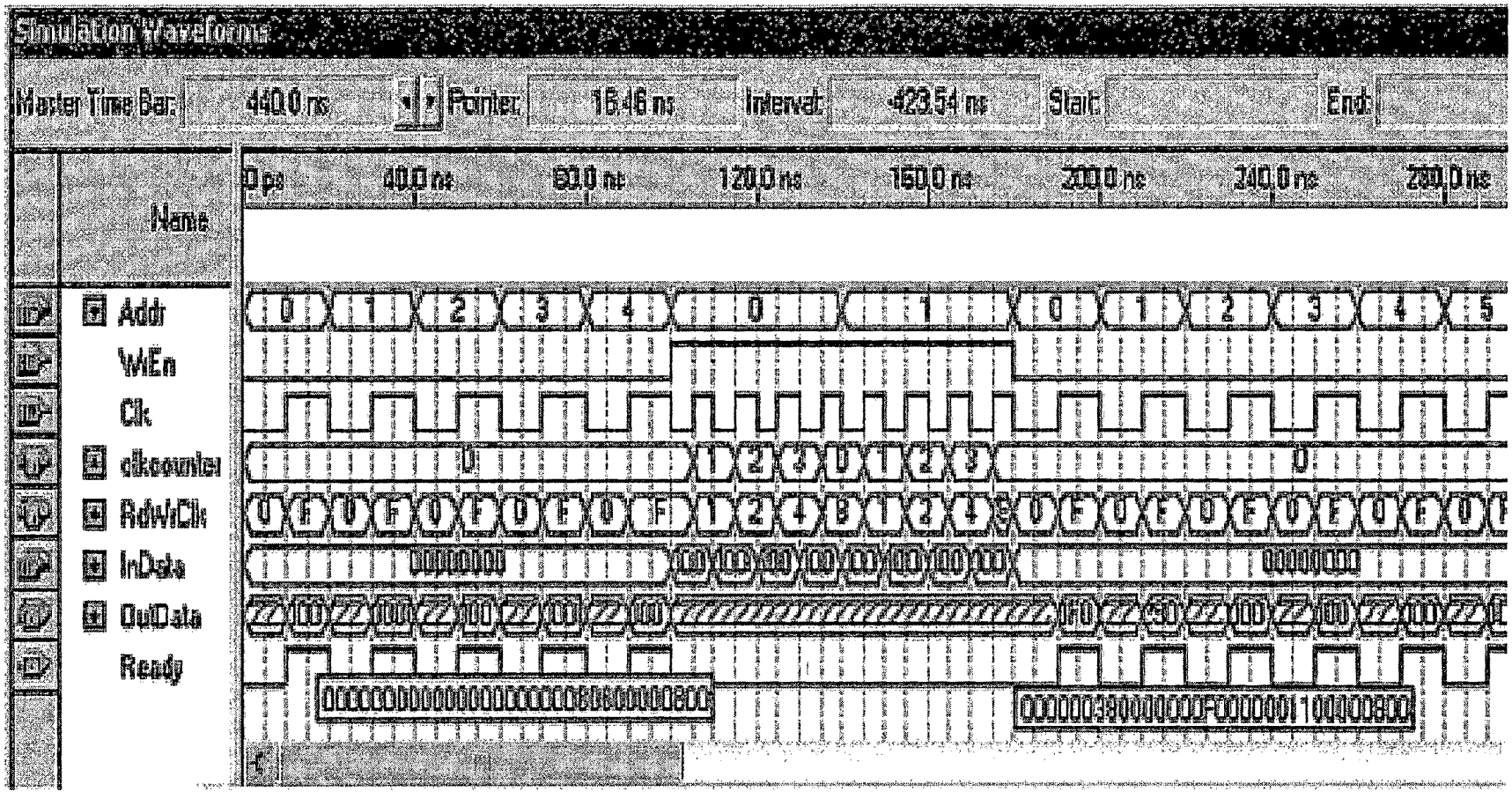

Fig. 3.31 Simulation Result of PLT

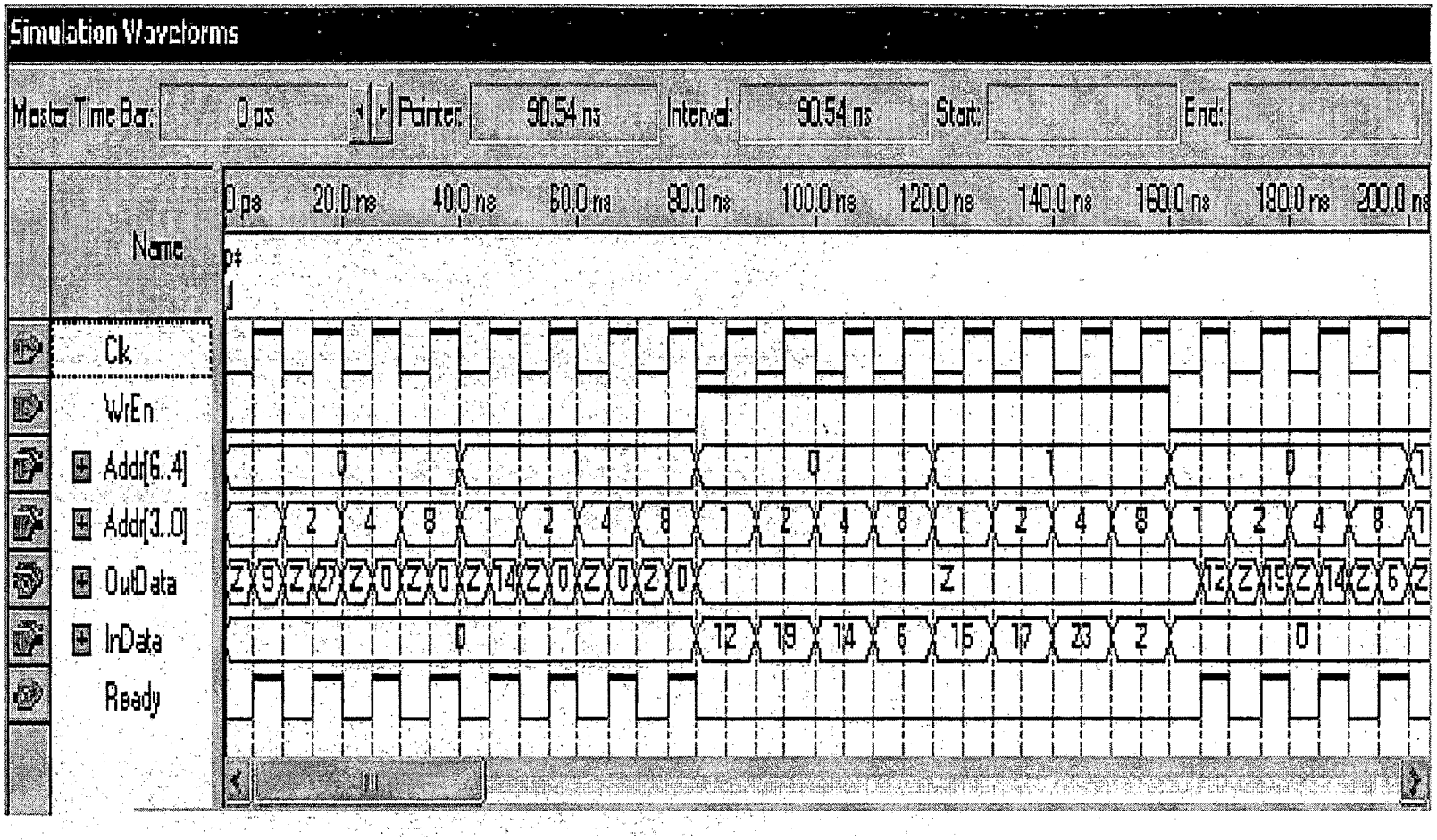

Figure 3.32 Simulation result of CLT

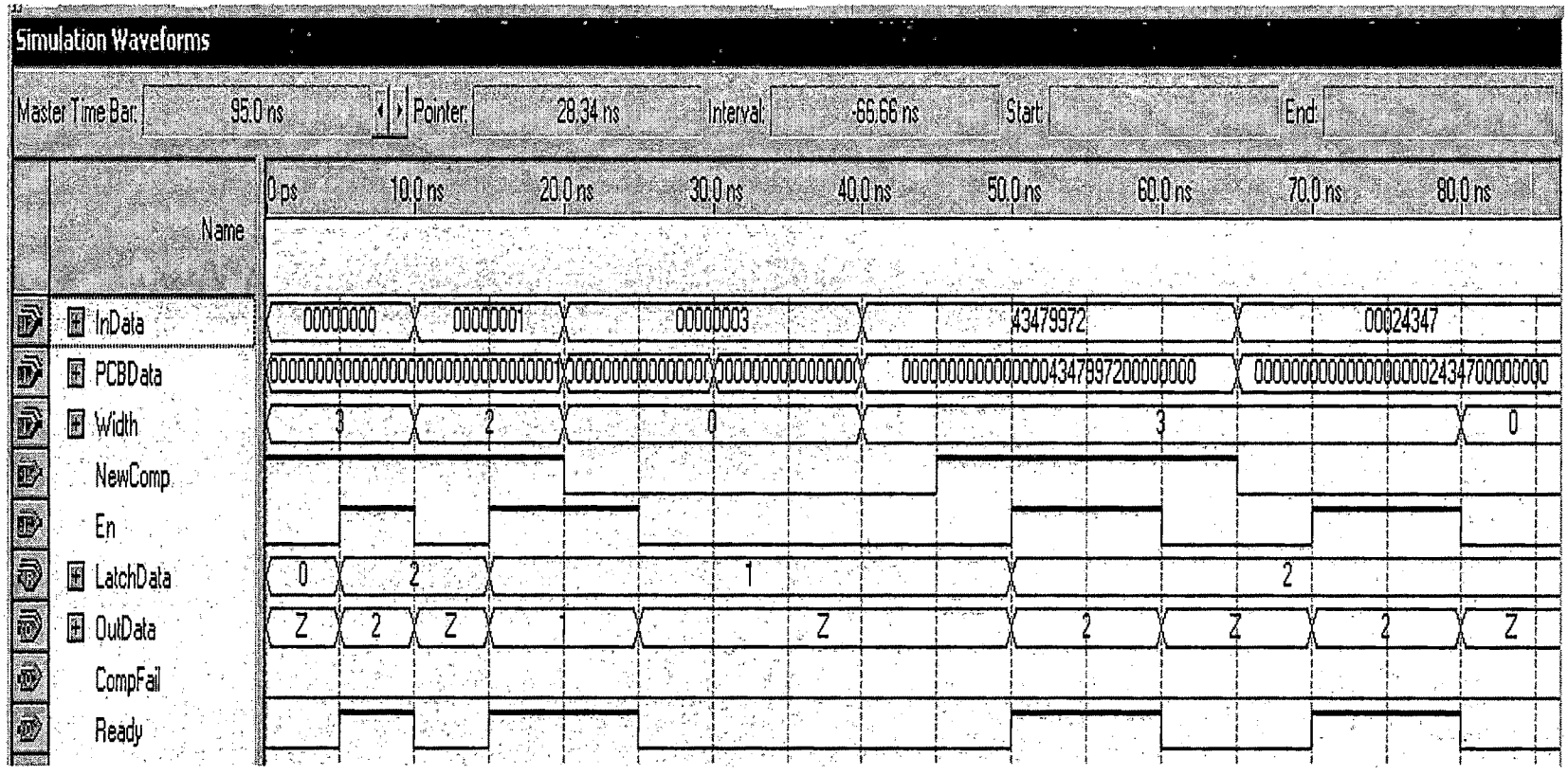

Figure 3.33 Simulation result of compare unit

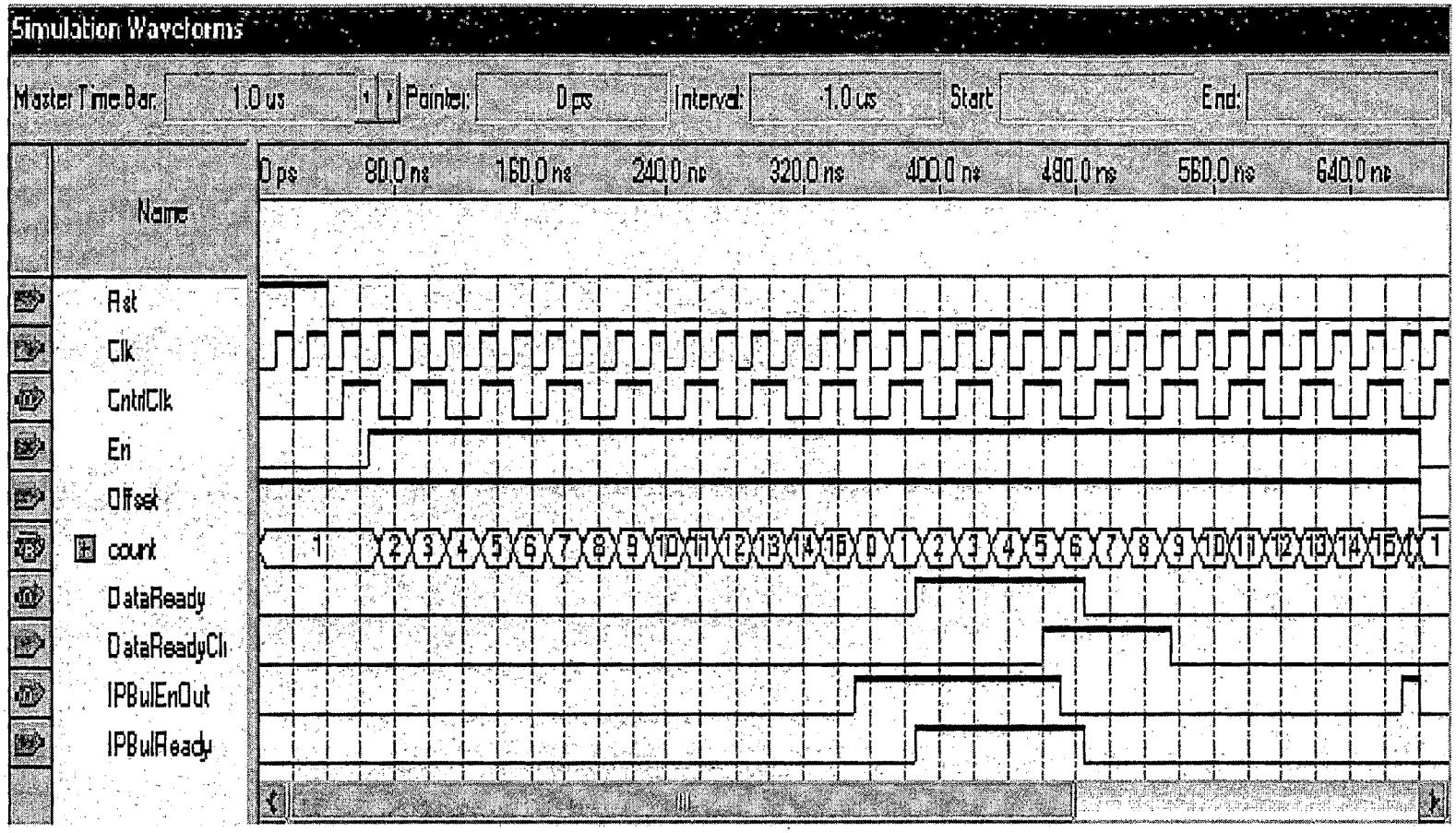

Figure 3.34 Simulation Result of Counter Unit

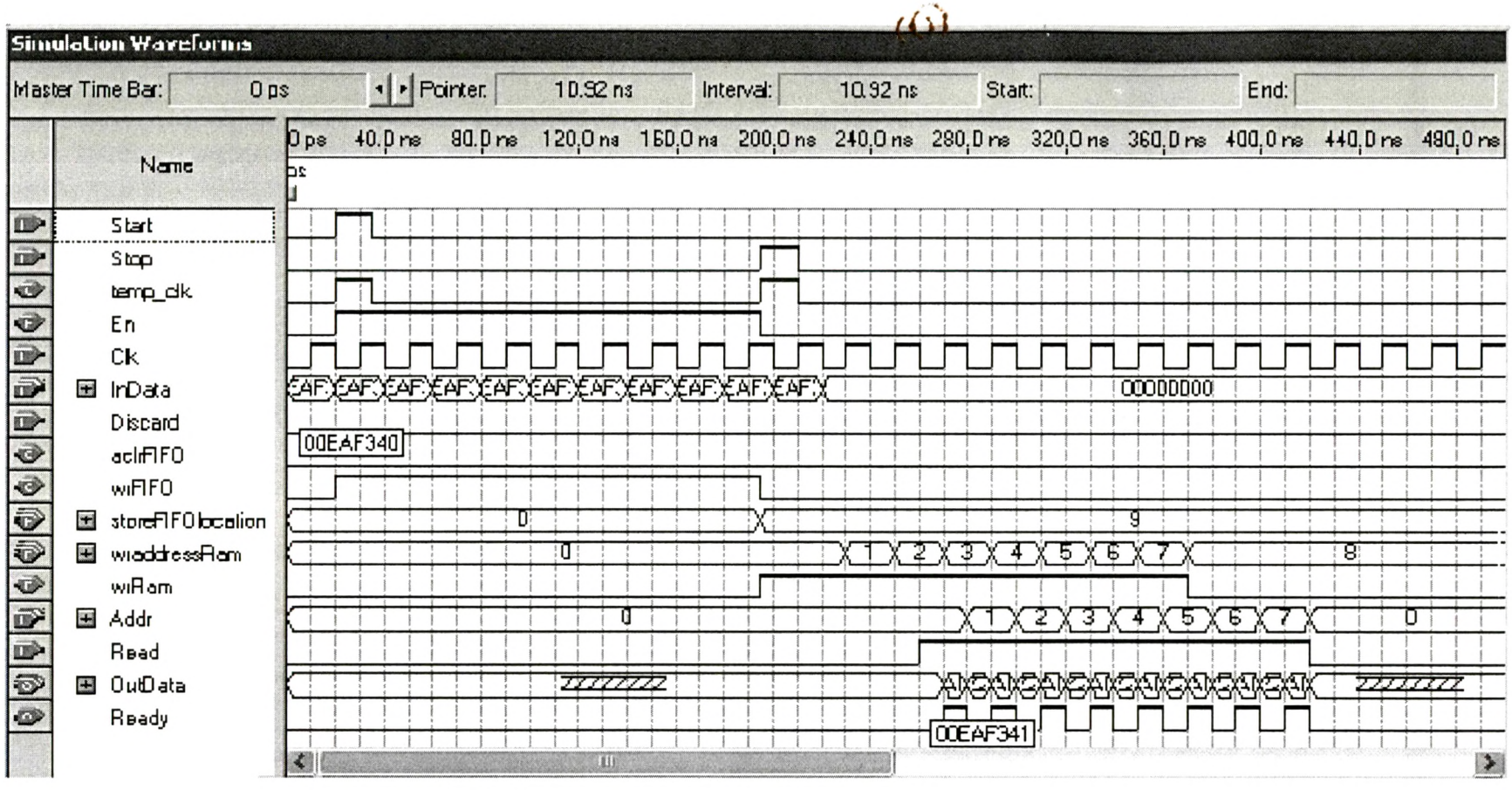

Figure 3.35 Simulation Result of Payload Memory

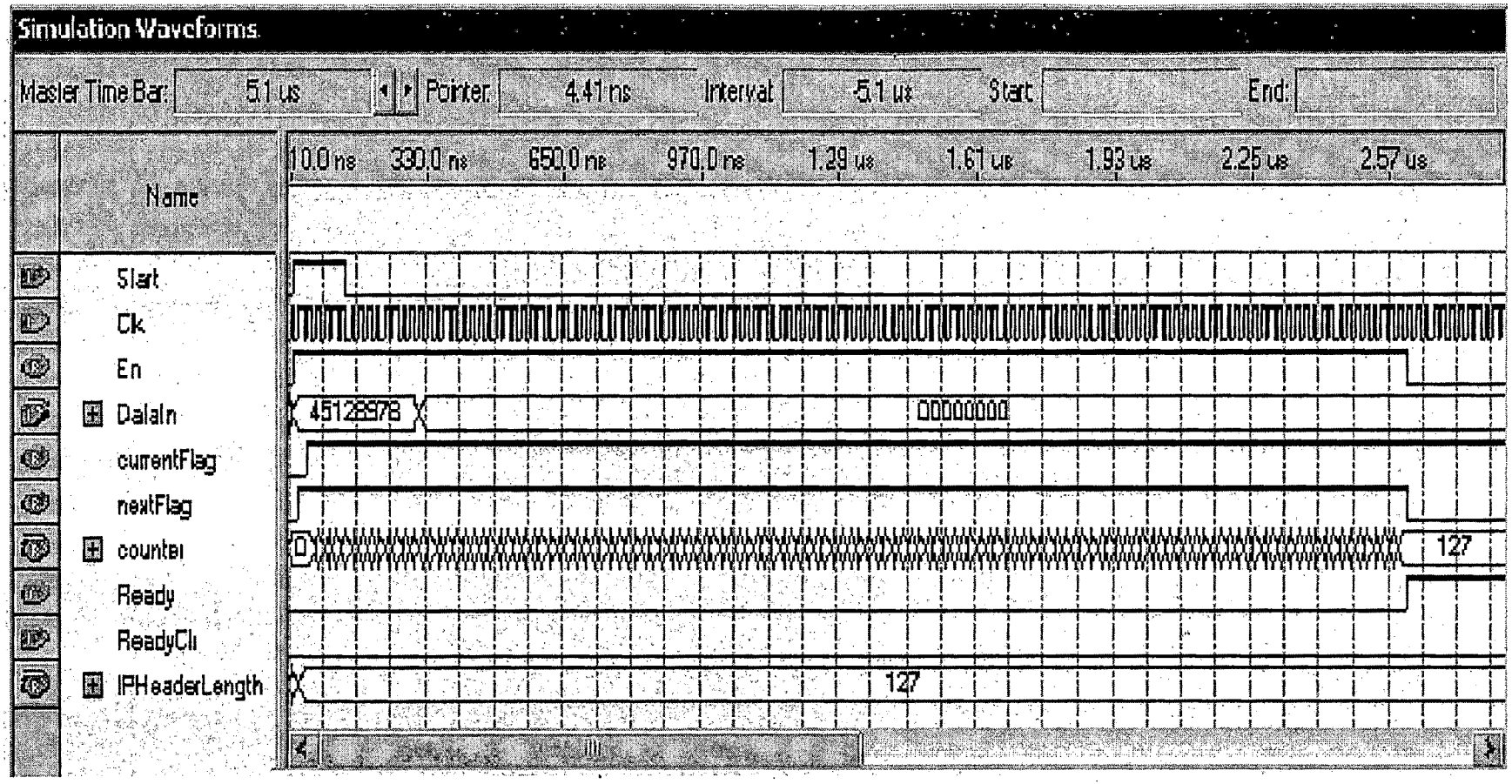

Figure 3.36 Simulation Result of IP length counter

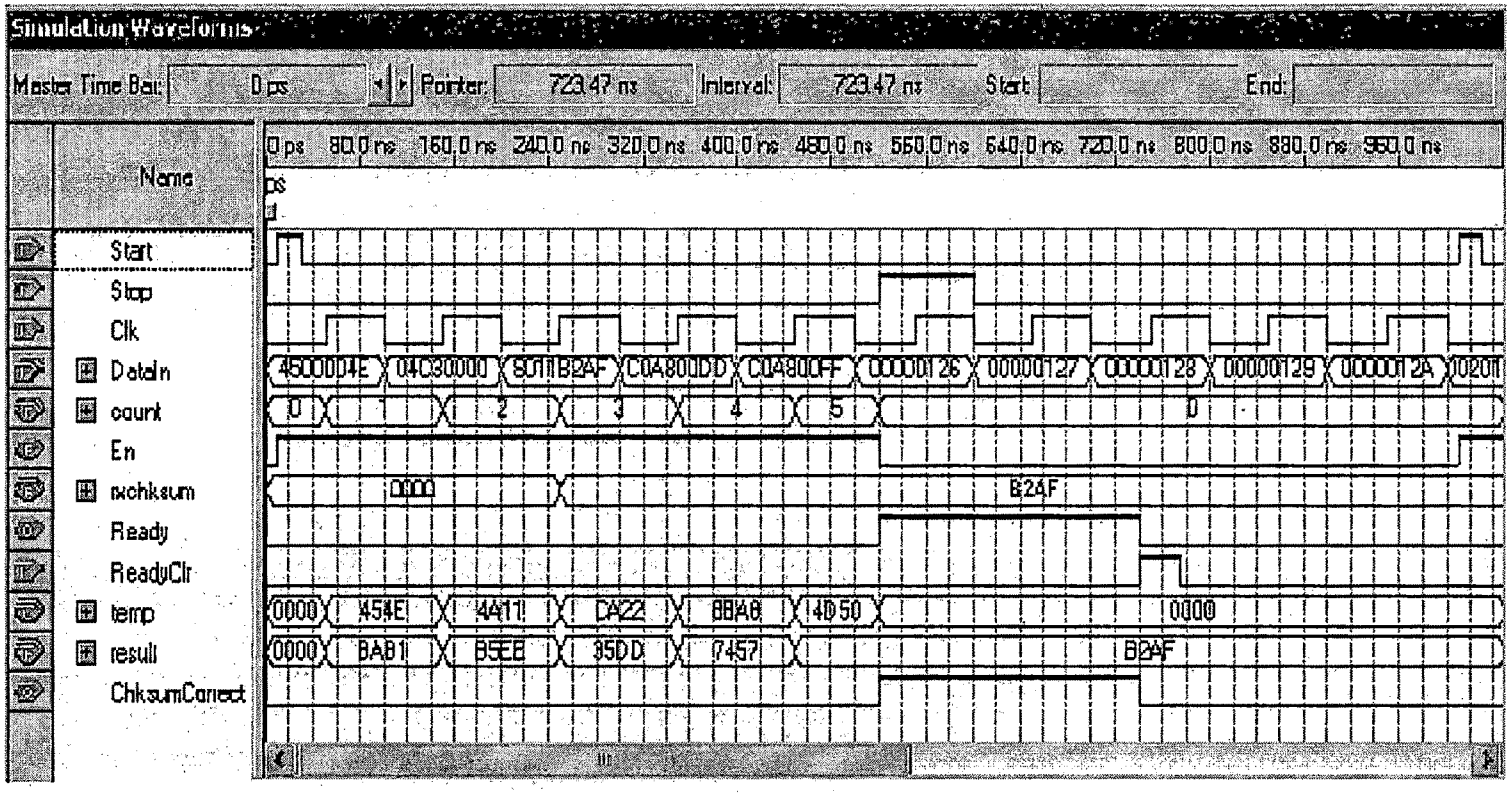

Figure 3.37 Simulation Result of Internet Checksum

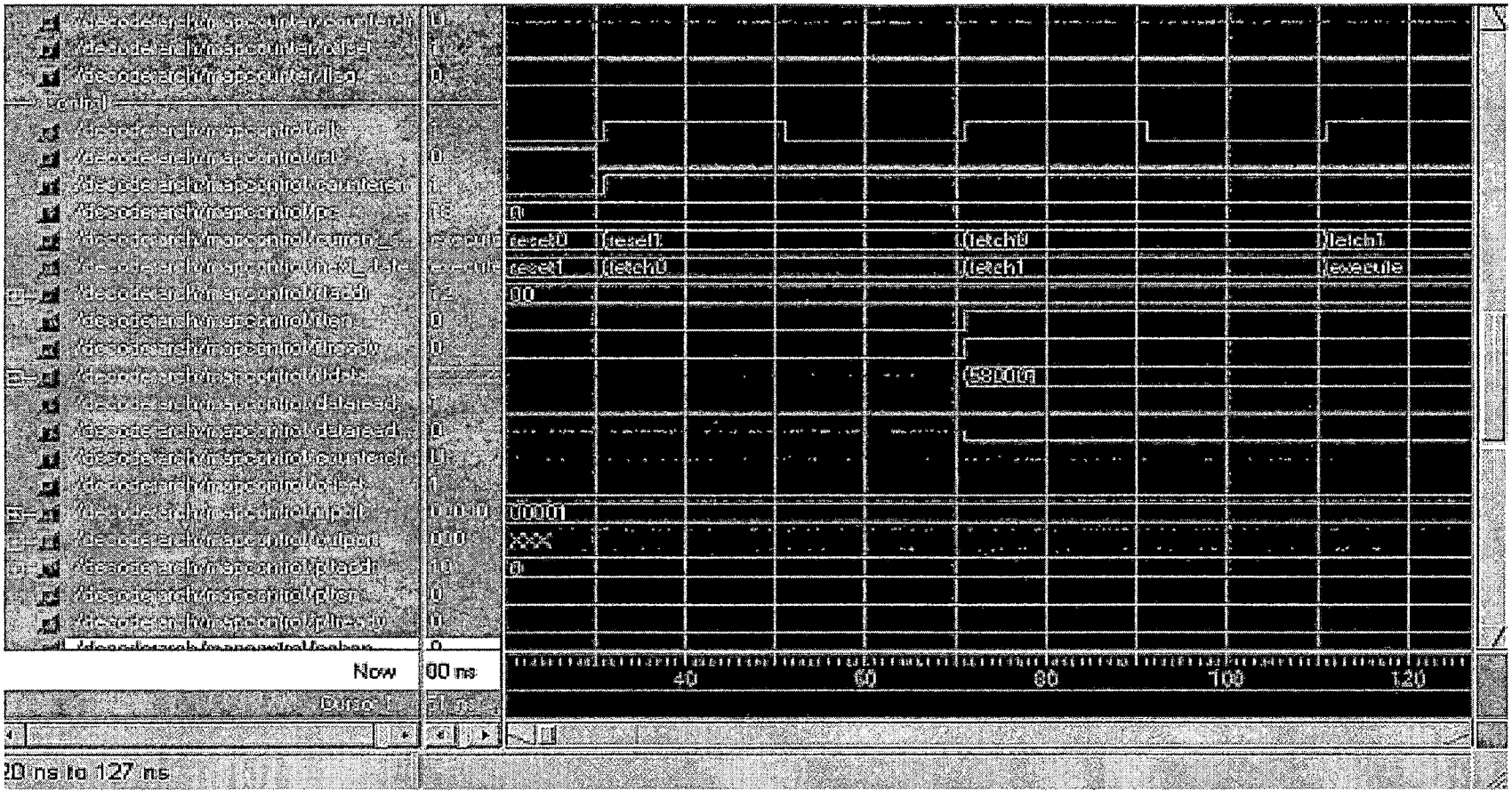

Figure 3 38 Execution of Instruction -1

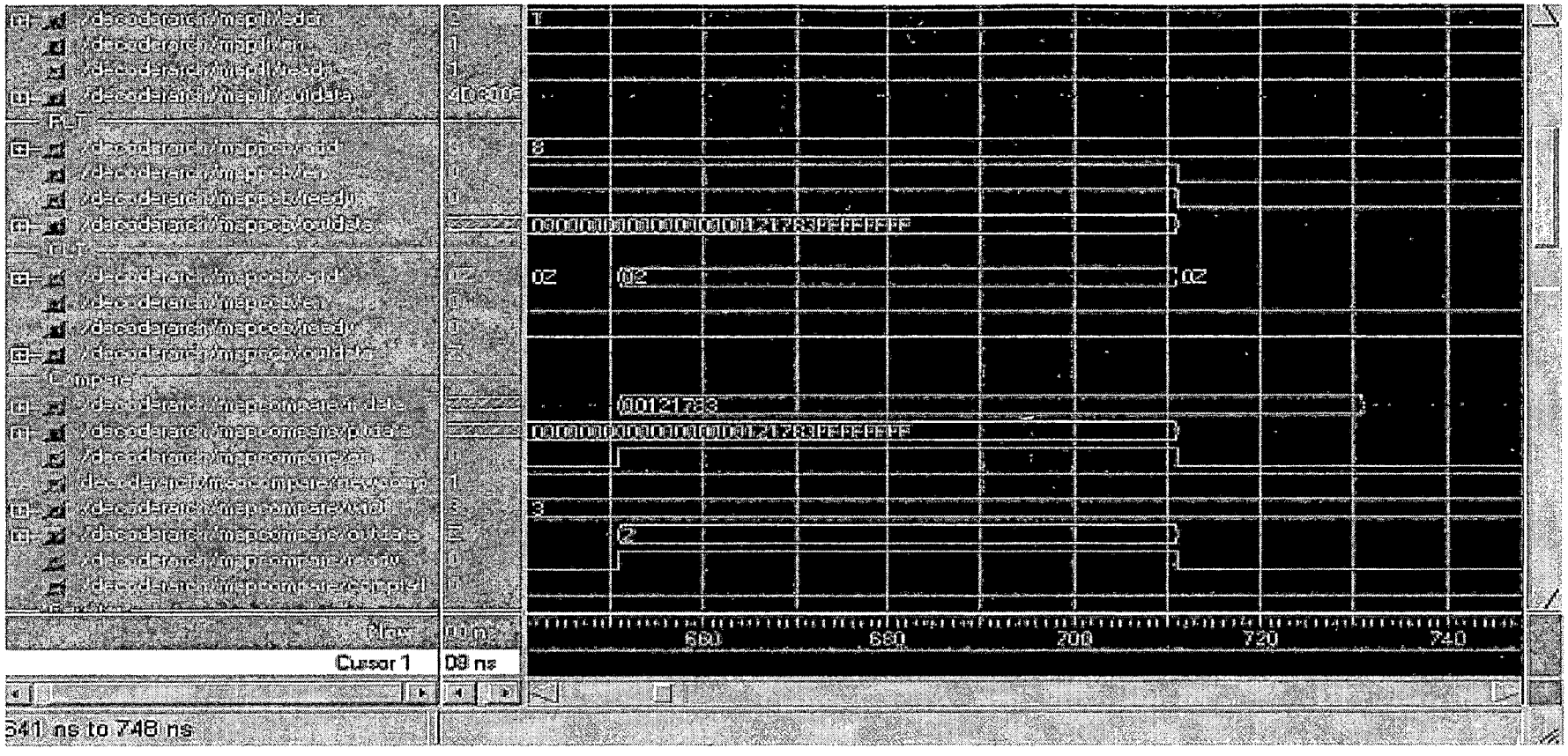

Figure 3.39 Execution of Instruction -2

3.4 SUMMARY

In this chapter, we have discussed and implemented two packet processing functions of Network Processor: Packet Encryption and Packet Classification, to understand above packet processing functions and get the VLSI area requirement. We have implemented and tested 8 round of IDEA algorithm using Quartus tool in VLSI in EP20k1500EBC652-1 device. IDEA requires 43249 logic elements with maximum clock frequency 3.85 Mhz and 246.4 Mbps data rate. We have implemented packet classification for router using hardware approach and packet classification for network terminal using hardware software co-design. The implemented architecture for network terminal using hardware software co-design has higher performance and is modular, so it can be modified/configured to incorporate any number of protocols. Currently no optimization is carried out, so performance can still be improved by applying optimization techniques.