2003

Processor Microarchitecture for Implementation of Ephemeral State Processing within Network Routers

Muthulakshmi Muthukumarasamy
University of Kentucky, mmuthulakshmi@gmail.com

Muthulakshmi Muthukumarasamy, Student

Dr. J. Robert Heath, Major Professor

Information not available, Director of Graduate Studies

ABSTRACT OF THESIS

PROCESSOR MICROARCHITECTURE FOR IMPLEMENTATION OF EPHERMERAL STATE PROCESSING WITHIN NETWORK ROUTERS

The evolving concept of *Ephemeral State Processing* (ESP) is overviewed. ESP allows development of new scalable end-to-end network user services. An evolving macro-level language is being developed to support ESP at the network node level. Three approaches for implementing ESP services at network routers can be considered. One approach is to use the existing processing capability within commercially available network routers. Another approach is to add a small scale existing ASIC based general-purpose processor to an existing network router. This thesis research concentrates on a third approach of developing a special-purpose programmable *Ephemeral State Processor* (ESPR) Instruction Set Architecture (ISA) and implementing microarchitecture for deployment within each ESP-capable node to implement ESP service within that node. A unique architectural characteristic of the ESPR is its scalable and temporal *Ephemeral State Store* (ESS) associative memory, required by the ESP service for storage/retrieval of bounded (short) lifetime *ephemeral* (*tag, value*) pairs of application data. The ESPR will be implemented to Programmable Logic Device (PLD) technology within a network node. This offers advantages of reconfigurability, in-field upgrade capability and supports the evolving growth of ESP services. Correct functional and performance operation of the presented ESPR microarchitecture is validated via Hardware Description Language (HDL) post-implementation (virtual prototype) simulation testing. Suggestions of future research related to improving the performance of the ESPR microarchitecture and experimental deployment of ESP are discussed.

KEYWORDS: Ephemeral State Processing, Ephemeral State Store, Ephemeral State Processor, PLD Technology, HDL Virtual Prototyping.

[redacted]

8-22-03

PROCESSOR MICROARCHITECTURE FOR IMPLEMENTATION OF EPHERMERAL STATE PROCESSING WITHIN NETWORK ROUTERS

By

Muthulakshmi Muthukumarasamy

[redacted]

Director of Thesis

[redacted]

Director of Graduate Studies

August 22, 2003

THESIS

Muthulakshmi Muthukumarasamy

The Graduate School

University of Kentucky

2003

PROCESSOR MICROARCHITECTURE FOR IMPLEMENTATION OF EPHERMERAL STATE PROCESSING WITHIN NETWORK ROUTERS

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Electrical
Engineering in the College of Engineering
at the University of Kentucky

By

Muthulakshmi Muthukumarasamy

Lexington, Kentucky

Director: Dr. J. Robert Heath, Associate Professor of Electrical and Computer Engineering

Lexington, Kentucky

2003

ACKNOWLEDGEMENTS

My sincere thanks and gratitude are due to my academic advisor and thesis director, Dr. J. Robert Heath for his guidance and support throughout the thesis. I am very thankful for his constant encouragement, suggestions and evaluations, and for the help he provided in editing various versions of this thesis. I would also like to express my sincerest thanks to Dr. Ken Calvert and Dr. James Griffioen, for providing me an opportunity to work on this thesis research and for the support and inspiration they provided. I would like to extend my thanks to Dr. Hank Dietz and Dr. William Dieter for serving in my thesis committee and providing me with invaluable comments and suggestions for improving the thesis and for possible future research.

I extend my deepest gratitude and thanks to my parents for their support and belief in me. My heartfelt thanks are due to my friend Balaji, who provided on-going support throughout this thesis and encouraged me in difficult times to complete this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter One

Introduction

This Chapter discusses the background needed for a better understanding of the research work, goals and objectives of the thesis.

1.1. Background and Positioning of Research

In order for the Internet to support new end-to-end communication services required by emerging network applications, additional network-level mechanisms are needed. There are three approaches which provide the needed network-level mechanisms in their own way. The first, more traditional approach, is to target a specific end-to-end problem and develop a focused, stand-alone network-based solution [1,2]. The second approach is to deploy a flexible infrastructure (e.g., active networks [3,4,5]) that can be reprogrammed to provide any needed functionality. The third approach is to extend the network functionality through simple *building-blocks,* which can be composed and combined by end-systems in different ways to create new services. The viability of the third approach depends on the following factors: it must be sufficiently general and useful to support a wide range of end-to-end network applications and must be able to justify the cost (financial, operational and performance) of deployment in network infrastructure.

Ephemeral State Processing (ESP) [6,7,8] is one such network-layer *building-block* approach which offers a possible solution for the development of new Internet end-to-end services and capabilities. The basic idea of ESP is to retrieve, store and process ESP packets from router nodes by means of creating and computing using *temporary state* in the network. Each ESP Packet carries a macro instruction – a 'program' (described in the following chapters) and collective programs provide/implement specific end-to-end network applications/services. Other publications [6,7] describe end-to-end services based on ESP, including 1.) Services for large-scale group applications, in which a relatively modest amount of in-network processing can pay big dividends in terms of scalability [10]; and 2.) Topology-exploring services, in which network elements having specific characteristics are found and flagged as locations for special processing [11].

ESP can be considered as a form of active networking, which offers: (1) lightweight Packet Processing service and (2) Computations involving multiple packets and multiple nodes, and the service is primarily focused on being implemented in fully programmable routers.

Ephemeral State Processing (ESP) [6,7,8] is an evolving research area of active networking, and this service offers very limited programmability that can be easily implemented in hardware. Multiple implementations of the ESP service are currently being investigated. One alternative is the use and adaptation of commercially available routers [26] such as the network processor described in [18]. Implementations based on commodity components have been explored (e.g., Linux-based routers), and have implemented the service as a module and user-level daemon on these lower level traffic routers often found near the periphery of the network. The goal is to implement ESP in core routers which should be able to offer the service at line rates by implementing it on the interface card. To that end, this current research approach targets PLD platforms that can be field upgraded to meet changing ESP functional and performance requirements. Under this approach, another processor, such as the one described in [18], would implement the routing function within the node.

This thesis research work aims at implementing ESP on a Special Purpose (SP) programmable processor within each network node – an Ephemeral State Processor (ESPR). ESP is implemented by a set of *macro-instructions*, which can be invoked on an Ephemeral State Processor (ESPR) at ESP-capable node routers as they receive, process and possibly forward specially-marked ESP packets in IP datagrams. Separation of ESP packets from other packets (such as the Internet Protocol (IP) packets) is carried out by logic inside the router and the ESPR only sees ESP packets. Figure 1.1 provides a high-level view of how an ESPR can be deployed in a router to perform the ESP service.

At most one macro-instruction is invoked by each ESP datagram (packet) as it enters an ESP capable node. An ESP packet's macro-instruction is executed by a successor ESP-capable node every time the packet is forwarded by an ESP-capable router. An ESP macro-instruction implemented by the ESPR of an ESP-capable node operates on values carried in the packet or stored at a node router in an associative memory of the ESPR called the *Ephemeral State Store (ESS)*.

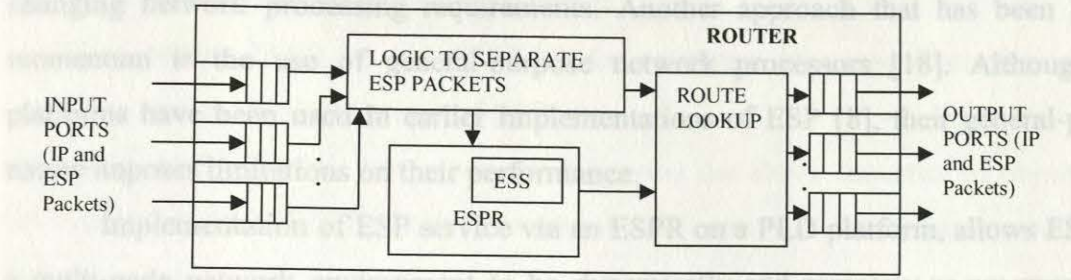

ROUTER
LOGIC TO SEPARATE ESP PACKETS
INPUT PORTS (IP and ESP Packets)
ESS
ESPR
ROUTE LOOKUP
OUTPUT PORTS (IP and ESP Packets)

Figure 1.1. ESP Processing in Router

The ESS allows data *values* to be associated with keys or *tags* for subsequent retrieval and/or update. The unique characteristic of the ESS is that it supports only *ephemeral* storage of (tag, value) pairs. Each (tag, value) binding is accessible for only a fixed interval of time after it is created. The *lifetime* of a (tag, value) binding in the ESS is defined by the parameter 'τ', which is globally specified and required to be approximately the same at each node. The value in the binding may be updated by any number of instructions (packets) during the lifetime τ. The ESS must provide fast associative creation, access, and reclamation of bindings, in order to process packets at "wire speeds". For given rates of instruction processing (instructions/sec), binding creation (new bindings/instruction) and a given lifetime (seconds), the size of ESS necessary to sustain those rates is fixed.

To our knowledge no research group has developed a SP programmable and reconfigurable network node processor architecture to implement ESP, such as the one to be described in this thesis. The ESPR microarchitecture can be implemented as an Application Specific Integrated Circuit (ASIC) chip or to a Programmable Logic Device (PLD) platform and fast/dense/cheap commodity memory chip technology. PLD technology is of interest because of its rapidly increasing density and performance at decreasing cost. Moreover, the use of PLD technology allows the ESP hardware to evolve over time as the concept of ESP evolves. Special purpose fixed-architecture communications node processors have been developed and implemented in the past, particularly in the context of ASICs, but they lack the programmability and flexibility as that of a PLD platform. These ASIC-based technologies offer no reasonable opportunity for in-field upgrades of the architecture or its instruction set architecture in response to changing network processing requirements. Another approach that has been gaining momentum is the use of general-purpose network processors [18]. Although such platforms have been used in earlier implementations of ESP [8], their general-purpose nature imposes limitations on their performance.

Implementation of ESP service via an ESPR on a PLD platform, allows ESPRs in a multi-node network environment to be dynamically and remotely re-programmed to incorporate architectural improvements or changes to the macro and micro instruction set as ESP evolves. Utilization of PLD technology for implementation of the ESPR within ESP capable nodes of a network would promote in-field upgrade capability of an ESPR instantiation whenever line speeds may increase or when the density of ESP packets in the total IP datagram traffic increases to a level requiring a higher performance ESPR. A higher performance ESPR architecture may be obtained by deeper pipelining, by instantiating multiple copies of the ESPR to a node PLD platform in a multiprocessor configuration, or by other architectural performance enhancements.

1.2. Goals and Objectives

The main goal of this thesis research work is to develop a processor microarchitecture – Ephemeral State Processor (ESPR) to implement ESP. Two versions of the ESPR architecture – ESPR Version 1 (ESPR.V1) and ESPR Version 2 (ESPR.V2) are developed for performance improvement reasons. Development of both versions can be accomplished by means of the following objectives:

(1) understand the concepts of ESP

(2) understand the ESP macro instructions and develop an implementing micro instruction set

(3) develop functional/operational/performance requirements for ESPR architecture versions

(4) develop a unique organization/architecture for the associative ESS

(5) develop a special-purpose programmable high-performance pipelined architecture for ESPR (ESPR.V1 and ESPR.V2)

(6) perform the design capture of ESPR.V1 and ESPR.V2 on to a Xilinx Virtex FPGA [17] using behavioral VHDL

(7) testing ESPR.V1 and ESPR.V2 for validation of correct execution of micro and macro instructions of ESP

This thesis research work was conducted following the above sequence of objectives to develop and validate ESPR. Brief contents of the chapters of this thesis are outlined as follows.

Chapter 2 – Here the concept of ESP and a description of example end-to-end network service applications is presented in detail.

Chapter 3 – Highest level Functional Organization/Architecture of an ESPR is described. The instruction set format, types and additional architectural details are described in this chapter.

Chapter 4 – Detailed description of an associative ESS design using Content Addressable Memory (CAM) is presented here.

Chapter 5 – Design of the first version of ESPR – ESPR.V1, comparison of ESPR with general-purpose processors and an analytical performance model for ESPR is described in this chapter.

Chapter 6 – This chapter deals with the detailed post-synthesis and post-implementation simulation validation testing of the ESPR.V1 architecture.

Chapter 7 – This chapter outlines the need for the second version of ESPR – ESPR.V2 and its design description. The description of a pipelined version of the previously designed ESS, for ESPR.V2, is also presented here.

Chapter 8 – This Chapter discusses the details of Hardware Description Language (HDL) design capture of the ESPR.V1 and ESPR.V2 processor systems.

Chapter 9 – Post-implementation simulation validation of ESPR.V2 is presented here.

Chapter 10 – This chapter concludes the thesis work and gives an insight to possible future research and investigation that can be done in this area.

Chapter Two

Ephemeral State Processing (ESP)

This chapter discusses the basic concepts of the ESP mechanism, a brief explanation of the Ephemeral State Store (ESS), ESP packets - format and processing, network macro instructions, and example practical applications of ESP followed by a brief introduction to the design of an Ephemeral State Processor (ESPR).

2.1 Introduction

Ephemeral State Processing (ESP) has been proposed as a network layer protocol to be implemented in routers to support a range of new scalable end-to-end network services and to improve scalability and performance of existing network services. It gives control to the end systems to support scalable network applications such as collecting network feedback, locating services, identifying 'branch points' [6], topology discovery and other auxiliary functions. The main idea of ESP is to carry service specific instructions (macro instructions) in its specially marked packets, enable the ESP capable router nodes to process the packets and leave a temporary state in the node according to the carried macro instructions and forward the packets to the next node or drop the packets with the state being already set for identification. This leads to the key requirements [8] for ESP development:

- provide means for the packets to leave information at a router for other packets to modify or pick up later as they pass through the path
- having a space-time product of storage for state storing
- having the space-time product of storage consumed as a result of any packet to be bounded
- per packet processing at each node be comparable to that of IP

The ESP protocol and network macro instructions (shown later in this chapter) are designed in such a way to meet the first requirement and it also lies in the hands of application services to meet this requirement by using ESP wisely. The design of an associative Ephemeral State Store (ESS) with a constant lifetime allows meeting the next two requirements. Each ESP packet carries a single macro instruction and so the per-packet processing time is known and bounded and the current goal is to process packets at or near wire speeds of 100 Mbps, which allows nearly a million packets being processed per second. With these requirements the ESP mechanism is based on three building blocks:

- an *Ephemeral State Store (ESS)*, which allows packets to deposit small amounts of arbitrary state at routers for a short time
- the *ESP protocol and packet format*, which defines the way by which the packets are processed and forwarded through the network.
- a *set of network macro instructions,* which defines the computations on ESP packets at the nodes

Ephemeral State Processing is initiated in any ESP-capable router when the router receives an ESP packet. Each router carries out only local operations and the responsibility for controlling and coordinating the system lies in the end-systems. The ESP header carries a network macro instruction out of a set of pre defined macro instructions. An instruction may create or update the contents of the ESS and/or fields in the ESP header and may place some information in the packet. A sequence of network macro instructions carried in ESP packets, form a practical ESP based application.

2.2 Ephemeral State Store (ESS)

Scalability of ESP is provided by the availability of an associative ESS at each network node. The associative ESS will allow data values to be associated with keys or tags for subsequent retrieval and/or update. The ESS will be unique in that it supports only *ephemeral* storage of (tag, value) pairs. Each (tag, value) binding is accessible for only a fixed interval of time after it is created and each tag has at most one value bound to it. Both tags and values are fixed size bit strings, the current design uses 64-bit tags and 64-bit values, to reduce the probability of collision [8].

The *lifetime* of a (tag, value) binding in ESS will be defined by the parameter 'τ', which is assumed to be approximately the same for each node. Once created, a binding remains in the store for 'τ' seconds and then vanishes; the value in the binding may be updated (overwritten and read) any number of times during the lifetime. For scalability, the value of 'τ' should be as short as possible. For robustness, the value of 'τ' needs to be long enough for interesting end-to-end services to be completed. This ESS supports two operations:

- **put** (x, e): bind the value e to tag x. After this operation, the pair (x, e) is in the set of bindings of the store for 'τ' seconds.
- **get** (x): Retrieve the value bound to tag x, if any. If no pair (x, e) is in the store when this operation is invoked or if the associated pair's lifetime is expired, the special value '⊥' meaning failure of the operation, is returned. ('⊥' - indicates the lifetime of the value is expired or the value is not in store).

2.3 ESP Packet Format and Processing

ESP packets are processed in ESP supporting routers as they travel through the network. Whenever an ESP packet arrives at a node, it is recognized as such and passed to the ESPR module for processing. These packets either propagate through to the original destination or are discarded along the path. Many end-to-end applications can be constructed using two steps – the first set of packets from end-systems establish and compute on the state while a second set of packets are used to collect the computed information. Two forms of ESP packets are supported: dedicated and piggybacked. A *dedicated* packet carries the ESP packet in an IP payload and *piggybacked* ESP packets carry ESP opcode and operands in an IP option (IPv4) or extension header (IPv6), as well as the regular application data (e.g., TCP/HTTP data) [8]. The ESP packet format is shown in Figure. 2.1.

FL	OP	LEN	CID	< VAR. FIELD >	CRC
(8)	(8)	(16)	(64)	(From 128 to 3968 bits)	(32)

FL – Flags (8 bits)
OP – Opcode (8 bits)
LEN – Length of the packet (16 bits)
CID – Computation ID (64 bits)
VAR. FIELD – Variable operands field that contains Tag and/or Value and/or a micro opcode (From 128 to 3968 bits, depending on the macro opcode)
CRC – Cyclic Redundancy Check (32 bits)

Figure 2.1. ESP Packet Format

The 8-bit FL (flag) field is organized as follows,

LOC (3)	E (1)	R (1)	U (3)

LOC – Location (3 bits)
E – Error (1 bit)
R – Reflector (1 bit)
U – Unused (3 bits)

Figure 2.2. FLAG field of ESP Packet

The *LOC* field identifies where the ESP processing should occur in the router [8], either the input side, output side or in the centralized ESP location, or any combination of these three locations. The *E* bit is set when an error occurs while processing an ESP packet (e.g., when a tag is not found in the ESS, when ESS is full, etc.). Such packets are forwarded to the destination without further processing allowing the end-systems to discover that the operation failed. *R* is the reflector bit, ESP routers forward packets with the reflector bit set without processing them [8].

CID – Computation ID, is a demultiplexing key: different packets that need to access the same state must have the same CID. The *OP* field identifies the ESP macro instruction to be performed, *LEN* field indicates the length of the ESP packet, *VAR. FIELD* carries the opcode specific operands and *CRC* field carries the Cyclic Redundancy Check code for the entire ESP packet.

2.4 Macro Instructions of ESP

Network macro instructions are the second building block of the ESP service. Each node in the network supports a predefined set of ESP instructions that can be invoked by ESP packets to operate on the ESS. Each ESP macro instruction takes zero or more operands, where each operand is one of the following types:

- a value stored in the local ESS (i.e. identified by a tag carried in the ESP packet)
- an 'immediate value' (i.e. one carried directly in the packet)
- a well known router value (i.e. the node's address)
- an associative or commutative operator (e.g., <, >=, etc)

Each ESP packet initiates exactly one network macro instruction and all macro instructions are carried out locally in the node, may update the state and/or the immediate values in the packet and after completion of execution, the packet that initiated it is either dropped or forwarded towards its original destination. A network macro (high-level language) instruction is implemented by a program comprised of micro (assembly language level) instructions. Macro instructions are combined and executed to implement emerging end-to-end application services. The defined macro instructions [8] are explained as follows:

COUNT:

The COUNT instruction takes two operands (carried in the ESP packet), a tag identifying a '*Count (pkt.count)*' value in the ESS and an immediate value '*Threshold*'. This instruction increments or initializes a counter and forwards or drops the packet, according to whether the resulting value is below or above a threshold value. It is used for counting packets passing through the router. The Ephemeral State Store (ESS) contains a number of (tag, value) pairs. The Ephemeral part of the ESS is that a value bound to a tag is active only for a particular period of time 'τ'. In this operation, if the specified tag in the packet is not currently bound, (i.e.) if there is no such tag found, a location is created for that tag in ESS, the value associated with it is set to '1' initializing it to be the first packet passing through the node. Otherwise if the tag is found, the value associated with it is incremented by one. If the resultant value reaches the '*Threshold*' value, subsequent COUNT packets will increment the counter but will not be forwarded.

This operation was devised based on networking applications such as Finding Path Intersection and Aggregating Multicast receiver feedback. The basis of this operation is to determine the number of members of a particular group and is useful for counting the number of children (nodes) sending packets through a node. COUNT is often used as a 'setup' message for subsequent collection messages. The values set in the ESS based on this packet allow later packets to retrieve useful information in performing network applications. For example, in Finding a Path Intersection the COUNT operation is the first step. The basic idea here is to count the number of router nodes in a particular path. If an ESPR module in a router receives a packet with COUNT operation, this router is observed to be in that path and a 'setup' message is set in that node by creating a (tag, value) pair in ESS. If a tag is not found, a location for this tag is created and the associated value is set to '1' to initiate a 'setup' message. Based on the appropriate '*Threshold*' value the resultant packet is forwarded or dropped to avoid implosion. Figure 2.3 shows the macro level description of the COUNT operation.

```
$t_0$  ⟵ get (pkt.count);
if ($t_0$ != ⊥) { put (pkt.count, $t_0$ +1); }
else { put (pkt.count, 1); }
if ($t_0$ <= threshold) forward;
else drop;
```

Figure 2.3. COUNT Operation

The macro level COUNT operation of Figure 2.3. can be explained on a line-by-line basis as follows.

Line 1: The value corresponding to tag-count in the packet is retrieved to a register **t_0**.

Line 2: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value. If a value is found in ESS and its lifetime has not expired, it is incremented and then placed in the ESS binding it to the corresponding tag-count.

Line 3: If a value is not found in ESS, a location is created for this tag-count in ESS with a value of 1 – meaning counting the initial packet.

Line 4: If the resultant value is less than or equal to the threshold value carried in packet, the packet is forwarded.

Line 5: Else the packet is discarded.

COMPARE:

The COMPARE instruction carries three operands (carried in the ESP packet), a tag '*V*' identifying the value of interest in the ESS, an immediate value '*pkt.value*' that carries the 'best' value found so far, and an immediate value '*<op>*' used to select a comparison operator to apply (e.g., min, max, etc). The COMPARE instruction tests whether the tag '*V*' has an associated value in the ESS within its lifetime and tests whether the relation specified by *<op>* holds between the value carried in the packet and the value in the ESS. If so, the value from the packet replaces the value in the ESS, and the packet is forwarded. If not, the packet is silently dropped. The COMPARE instruction can be used in a variety of ways but is particularly useful in situations where only packets containing the highest or lowest value seen by the node so far are allowed to continue on. This operation is mainly used as a second step in Finding Path Intersection after a COUNT operation. Figure 2.4 shows a macro level description of the COMPARE operation.

```
t0 ←  get (pkt.v);
if (t0 = ⊥)
{ put (pkt.v, pkt.value);
forward; }
else
if (t0 <op> pkt.value)
{ put (pkt.v, pkt.value);
forward; }
else
drop;
```

Figure 2.4. COMPARE Operation

Below is a line-by-line description of the macro level COMPARE operation of Figure 2.4.

Line 1: The value corresponding to tag-v in the packet is retrieved to a register $\mathbf{t_0}$.

Line 2&6: The value is checked for its availability in ESS, its lifetime expiry and it is also checked whether the relation specified by *<op>* holds between this value and the value carried in the packet.

Line 3&7: If so, the value from the packet replaces the value in the ESS.

Line 4&8: The resultant packet is forwarded.

Line 10: If not, the packet is dropped.

COLLECT:

The COLLECT macro instruction carries four operands (carried in the ESP packet), a tag identifying the '*Count*' value in the ESS, a tag identifying a '*Value*' in the ESS to perform an associative or commutative operation on, an immediate value '*pkt.data*', which carries the resultant value from the operation performed from child nodes and an immediate value '*<op>*' that indicates the actual operator to be applied.

The COLLECT macro operation is used by a network node to compute an associative or commutative operation on values sent back by its children nodes. If register

t_0 contains the count for the number of children nodes, each COLLECT packet from a child node is applied to the node's current result and **t_0** is decremented. The parent node holds the current result, which is obtained by performing associative or commutative operations on values sent by its children nodes. After all children have reported their value, the computed result is forwarded to the next hop. Figure 2.5 illustrates the macro level description of the COLLECT operation.

This operation is mainly used in aggregating receiver feedback, for example, loss rate corresponding to a group. After obtaining information back on the number of children in a group from the COUNT operation, this operation is performed on values sent by the children and on corresponding conditions in this operation. This macro operation allows particular feedback information such as loss rate to be determined.

```
$t_0$  $\leftarrow$  get (pkt.count);
if ($t_0$ != $\perp$) {
        $t_1$  $\leftarrow$  get (pkt.value);
        if ($t_1$ != $\perp$) {
                $t_1$  $\leftarrow$  $t_1$ <op> pkt.data; }
        else { $t_1$  $\leftarrow$  pkt.data; }
        put (pkt.value, $t_1$);
        $t_0$  $\leftarrow$  $t_0 - 1$;
        put (pkt.count, $t_0$);
        if ($t_0$ == 0) {pkt.data := $t_1$; forward; }
        else { drop; }
} else abort;
```

Figure 2.5. COLLECT Operation

Below is a line-by-line description of the macro level COLLECT operation of Figure 2.5.

Line 1: The value corresponding to tag-count in the packet is retrieved to a register **t_0**.

Line 2: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value. If the corresponding tag with value is found, it indicates the number of children nodes in a particular group. If there is no such tag found, Line 12 is performed.

Line 3: The value corresponding to tag-value in the packet is retrieved to a register **t_1**. It corresponds to a value sent by a child node.

Line 4: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value.

Line 5: If the corresponding tag with value is found, then an associative or commutative operation indicated by <op> (a micro opcode carried in the packet) is performed on this value and the value (pkt.data) carried in the packet, and the result is placed in $\mathbf{t_1}$.

Line 6: If no such tag with value is found, then the value (pkt.data) carried in the packet is placed in $\mathbf{t_1}$.

Line 7: The resultant value in $\mathbf{t_1}$ is written into ESS with its associated tag-value.

Line 8&9: After performing the operation on one child node, the number of children nodes is decremented by one and this new value is placed in ESS with its associated tag-count.

Line 10: It is now checked to see whether the number of children nodes is zero, (i.e.) whether the operation is completed on all children nodes. If there is no child left, then the final result from $\mathbf{t_1}$ is placed in the packet and the resultant packet is forwarded.

Line 11: But if there are some children left, the packet is dropped.

Line 12: This line indicates an abort statement if the parent node doesn't have the count on number of children. It sets a corresponding 'E' bit to '1' and 'LOC' bits to zero in the 'FLAGS' part of the packet and forwards the packet to the next node.

RCHLD:

The RCHLD macro instruction carries four operands (carried in the ESP packet), a tag specifying the Identifier Bitmap '*tagb*' and an immediate identifier value '*idval*', a tag '*C*' identifying count of forwarded packets and an immediate threshold '*thresh*'. The RCHLD macro instruction is similar to the COUNT macro instruction except that it also records the identifiers in packets received from its children. For example, tree-structured [8] computations for collecting information from the group members can be carried out in two phases:

- The first phase corresponds to a RCHLD instruction, which uses ESP to record the identifiers.
- The second phase corresponds to the RCOLLECT instruction (which will be described next), in which the group members send their identifier values up the

tree (towards destination) and each node uses RCOLLECT to compute and forward the result only after having heard from every child.

Each group member sends the RCHLD instruction towards the root; this instruction causes the interior node or the immediate parent node to receive packets carrying this instruction from each of its children. For some useful computational applications [8], it is useful to determine whether a packet comes from a child that has not been heard from previously. To accomplish this, *Bloom Filters* [8,9] are used to determine a random bit sized identifier for each node called *bitmap identifier*. Figure 2.6 illustrates the macro level description of the RCHLD operation.

```
$t_0 \leftarrow$ get (pkt.tagb);
if ($t_0$ != $\perp$) {    $t_0 \leftarrow$    $t_0$ <OR> pkt.idval;}
else {   $t_0 \leftarrow$   0;}
put (pkt.tagb, $t_0$ );
$t_1 \leftarrow$ get (pkt.C);
if ($t_1$ != $\perp$) { put (pkt.C, $t_1$ +1); }
else { put (pkt.C, 0); }
if ($t_1$ <= thresh)
{ pkt.idval := current node's identifier value;
forward;}
else drop;
```

Figure 2.6. RCHLD Operation

Below is a line-by-line description of the macro level RCHLD operation of Figure 2.6.

Line 1: The value corresponding to tag-tagb in the packet is retrieved to a register $\mathbf{t_0}$.

Line 2: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value. If the corresponding tag with value is found, indicating the bitmap identifier(s) of the other children nodes for an immediate parent, the immediate value carried in the packet is bit wise ORed with the value found in ESS meaning the bit corresponding to the bitmap identifier of the current child is turned on and is also included (added) as children for the immediate parent.

Line 3: If its not found, the value is set to '0'.

Line 4: The resulting new value is written into ESS.

Line 5: The value corresponding to tag-C in the packet is retrieved to a register $\mathbf{t_1}$.

Line 6: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value. If a value is found in ESS and its lifetime has not expired, it is incremented and then placed in ESS binding it to the corresponding tag-count.

Line 7: If a value is not found in ESS, a location is created for this tag-count in ESS with a value of 0.

Line 8, 9&10: If the resultant value is less than or equal to the threshold value carried in packet, the current node's bitmap identifier value is written into the packet, and the resultant packet is forwarded.

Line 5: Else the packet is discarded.

RCOLLECT:

In addition to '*Value*', '*pkt.data*' and '*<op>*' operands carried in COLLECT packet, the RCOLLECT macro instruction carries four more operands in the packet: a tag '*tagb1*' identifying the bloom filter used in the previous RCHLD instruction, a tag '*tagb2'* identifying another bitmap for detecting duplicates, a tag '*D*' for identifying the count of packets forwarded and an immediate threshold value *'thresh'* to control the number of duplicated transmissions.

This instruction is used as a second phase after the RCHLD macro instruction for tree-structured computations. The main difference between COLLECT and RCOLLECT is that in RCOLLECT the condition for forwarding is when the two Bloom filters match, rather than when the count is zero. This packet is sent after a short delay to allow phase one packets to be processed. As each packet arrives, the bit corresponding to its bitmap identifier is set in the second bitmap, and the value is added into the existing binding. If the resulting bitmap is equal to the one from the first phase, it means that all children identified in the first phase have been heard from. In that case the accumulated value is written into the packet, the bitmap identifier in the packet is replaced with that node's identifier, and the packet is forwarded. Otherwise, the packet is discarded. Figure 2.7 illustrates the macro level description of RCOLLECT operation.

```
$t_0$ ← get (pkt.tagb1);
if ($t_0$ != ⊥) {
        $t_1$ ← get (pkt.tagb2);
        if ($t_1$ != ⊥) {    $t_1$ ← $t_1$ <AND> pkt.idval;}
        else {    $t_1$ ← 0;}
        if ($t_1$ != pkt.idval) {    $t_1$ ← $t_1$ <OR> pkt.idval;}
        put (pkt.tagb2, $t_1$ );
        $t_2$ ← get (pkt.value);
        if ($t_2$ != ⊥) {
                $t_2$ ← $t_2$ <op> pkt.data; }
        else { $t_2$ ← pkt.data; }
        put (pkt.value, $t_2$);
        if ($t_0$ == $t_1$) {
                $t_3$ ← get (pkt.D);
                if ($t_3$ != ⊥) { put (pkt.D, $t_3$ +1); }
                else { put (pkt.D, 0); }
                if ($t_3$ <= thresh)
                {pkt.value := $t_2$ ;
                pkt.idval:= node's identifier value;
                forward; }
                else {drop; }
        else {drop; }
else {abort; }
```

Figure 2.7. RCOLLECT Operation

Below is a line-by-line description of the macro level RCOLLECT operation of Figure 2.7.

Line 1: The value corresponding to tag-tagb1 in the packet is retrieved to a register **t_0.**

Line 2: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value. If the corresponding tag with value is found indicating the identifier bitmap(s) obtained from previous phase one (RCHLD) operations, described in Line 3 – Line 22 are executed. If there is no such tag found, Line 23 is performed.

Line 3: The value corresponding to tag-tagb2 in the packet is retrieved to a register **t_1.**

Line 4&5: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value. If the corresponding tag with value is found, indicating the bitmap identifier(s) of the other children nodes for an immediate parent, the value is added to the existing value, and if the value is not found, its set to '0'.

Line 6: Then it is checked whether the resulting value is equal to the one carried in the packet, if its not equal, the immediate value carried in packet is bit wise ORed with the value found in ESS meaning the bit corresponding to the bitmap identifier of the current child is turned on and is also included (added) as children for the immediate parent.

Line 7: The resulting new value is written into ESS.

Line 8: The value corresponding to tag-value in the packet is retrieved to a register $\mathbf{t_2}$. It corresponds to a value sent by a child node.

Line 9: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value.

Line 10: If the corresponding tag with value is found, then an associative or commutative operation indicated by <op> (a micro opcode carried in the packet) is performed on this value and the value (pkt.data) carried in the packet, and the result is placed in $\mathbf{t_2}$.

Line 11: If no such tag with value is found, then the value (pkt.data) carried in the packet is placed in $\mathbf{t_2}$.

Line 12: The resultant value in $\mathbf{t_2}$ is written into ESS with its associated tag-value.

Line 13: The bitmap identifiers are compared for equality to check whether the values from all children nodes have been heard and to forward the packet.

Line 14: If they are equal then, the value corresponding to tag-D in the packet is retrieved to a register $\mathbf{t_3}$ to have the count of packets.

Line 15: The value is checked for its availability in ESS. '⊥' indicates lifetime expiry of this value. If a value is found in ESS and its lifetime has not expired, it is incremented and then placed in ESS binding it to the corresponding tag-D indicating that this packet is counted.

Line 16: If a value is not found in ESS, a location is created for this tag-count in ESS with a value of 0 starting to count the packets.

Line 17, 18, 19&20: If the resultant value is less than or equal to the threshold value (for the maximum number of packets) carried in packet, the resultant value in $\mathbf{t_2}$ is placed in the output packet's 'pkt.value' field, current node's bitmap identifier value is placed in 'pkt.idval' field and the resultant packet is forwarded.

Line 21: Else the packet is discarded.

Line 22: If the bitmap identifiers do not match meaning there's still some child nodes to hear from, then the packet is silently dropped.

2.5 Example End-to-End Applications using ESP

End systems utilize ESP to perform various applications. Many applications can be constructed using two-step network macro instructions. For example, in *Finding Path Intersection* as shown in Figure 2.8, the first step utilizes the COUNT macro instruction in determining the number of nodes along the path and defining a state in each node's ESS as it travels. The next step, the COMPARE instruction, examines the value left by the previous COUNT instruction and determines the nearest intersection node along the path.

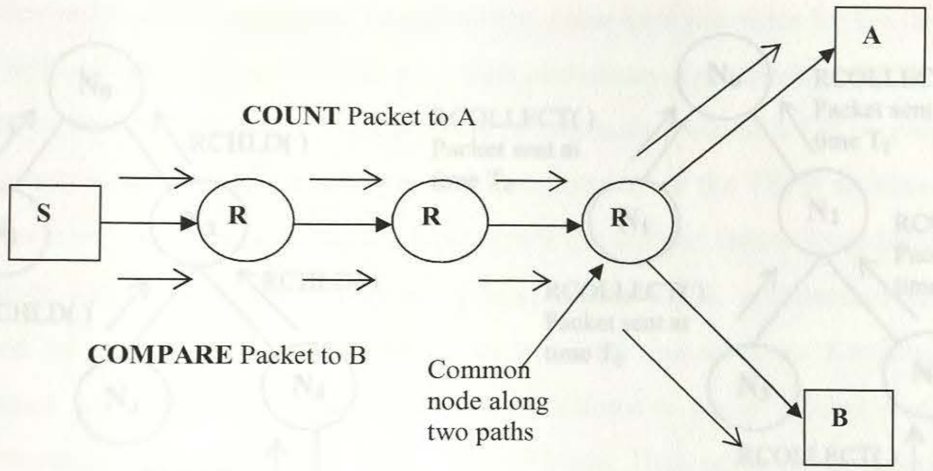


Figure 2.8. Finding Path Intersection

Another example is in *Aggregating Multicast Receiver Feedback* in which first the number of children maintaining a state are counted and then some operation is performed on the values collected to deliver some useful information like maximum loss rate etc., ESP facilitates such operations without the risk of implosion (see Figure 2.9). These computations are viewed as tree structured computations by ESP and are generally carried out in two phases.

In the first phase each group member sends an RCHLD macro-instruction towards the root; this instruction causes the interior node or the immediate parent node to receive packets carrying this instruction from each of its children and records their identifiers (Figure 2.9a). The identifiers are helpful in determining whether the parent has heard from all children nodes, and this information is useful in some specific applications

[8,26]. After a short delay (for processing RCHLD packets), phase two RCOLLECT packets are sent towards the destination root (Figure 2.9b). The parent node receives packets from each of its children one by one, and sets the bit corresponding to its bitmap identifier in the second bitmap, and the value carried in the packet is added to the existing binding in ESS. If the resulting bitmap is equal to the one from the first phase, it means that the parent has heard from all its children and the accumulated value is written into the packet, the bitmap identifier in the packet is replaced with that node's identifier, and the packet is forwarded. Otherwise, the packet is discarded.

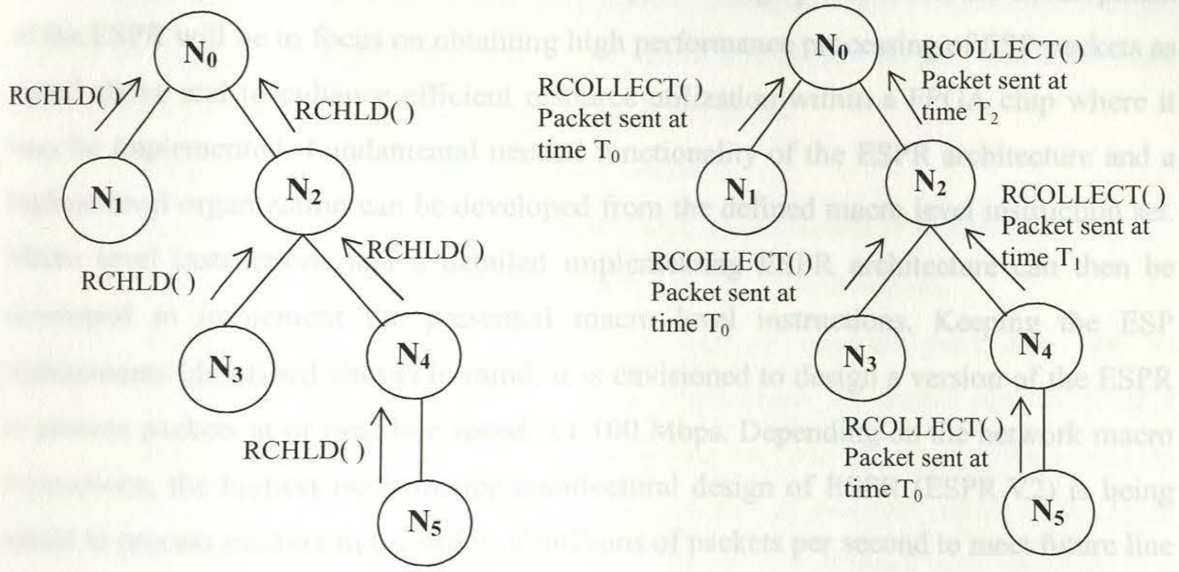


Figure 2.9a. Phase 1 (RCHLD Packet to Root Node)

Figure 2.9b. Phase 2 (RCOLLECT Packet to Root Node)

Figure 2.9. Reducing Implosion using Two-Phase Tree Structured Computations

Various other applications may be implemented such as thinning group feedback within a network allowing prevention of the implosion problem [10], simple distributed computations requiring data gathering across the network, identifying network topology information [8] and network bottleneck identification [2].

2.6 Prologue to Ephemeral State Processor (ESPR)

It is envisioned that a special purpose programmable processor architecture can be developed which will allow ESP to be programmed into programmable logic within network level routers to support functional applications. This architecture can be described in a hardware descriptive language (HDL) and then simulated using an appropriate simulator for its first level of architectural functionality validation. Final architectural functionality, design correctness and performance can be verified by implementing the design in a Field Programmable Gate Array (FPGA) chip through virtual prototype implementation and testing.

Beyond correct operational functionality, other high priorities for the development of the ESPR will be to focus on obtaining high performance processing of ESP packets as stated above and to enhance efficient resource utilization within a FPGA chip where it may be implemented. Fundamental needed functionality of the ESPR architecture and a highest-level organization can be developed from the defined macro level instruction set. Micro level instructions and a detailed implementing ESPR architecture can then be developed to implement the presented macro level instructions. Keeping the ESP requirements (described above) in mind, it is envisioned to design a version of the ESPR to process packets at or near line speeds of 100 Mbps. Depending on the network macro instructions, the highest performance architectural design of ESPR (ESPR.V2) is being aimed to process packets in the order of millions of packets per second to meet future line speeds.

Chapter Three

Ephemeral State Processor (ESPR)

This chapter discusses the characterization and requirements needed for designing an Ephemeral State Processor (ESPR), highest level functional organization and its basic micro instruction set formats and types. Finally, it also presents the equivalent micro instruction implementation of the already defined high level macro instructions.

3.1 ESPR Requirements Summary

Based on the given ESP mechanism, to be implemented into Programmable Logic Device (PLD) technology, the basic processor building blocks and processor characteristics/requirements of ESP service can be given as,

- *An Ephemeral State Store (ESS) is needed.*
- *Compatibility with the ESP protocol and packet format is required.*
- *Must support a predefined set of network macro-instructions.*
- *Develop an ESP architecture that has an upgrade path and can be performance boosted via systematic steps (such as deeper pipelining of a pipelined architecture, move from issuing one instruction per-clock-cycle to two instructions per-clock-cycle, instantiation of multiple copies of the basic ESPR architecture to a single PLD platform in a network node resulting in a multiprocessor configuration).*
- *Support an in-field upgrade path (e.g., via a software upload)*

Following the above ESP requirements, the ESP processor requires a reduced latency ESS, which is designed as an associative memory to store ephemeral (tag, value) pairs. A packet storage unit is required to store and send packets to the output, and a way of indicating the state of this packet to the next node in the path is done using a "code register". The third requirement of being able to implement the network macro-instructions in the node requires development of a set of micro-instructions, supporting high-level architectural configuration and the Instruction Set Architecture (ISA) which will be used to implement ESP macro-instructions.

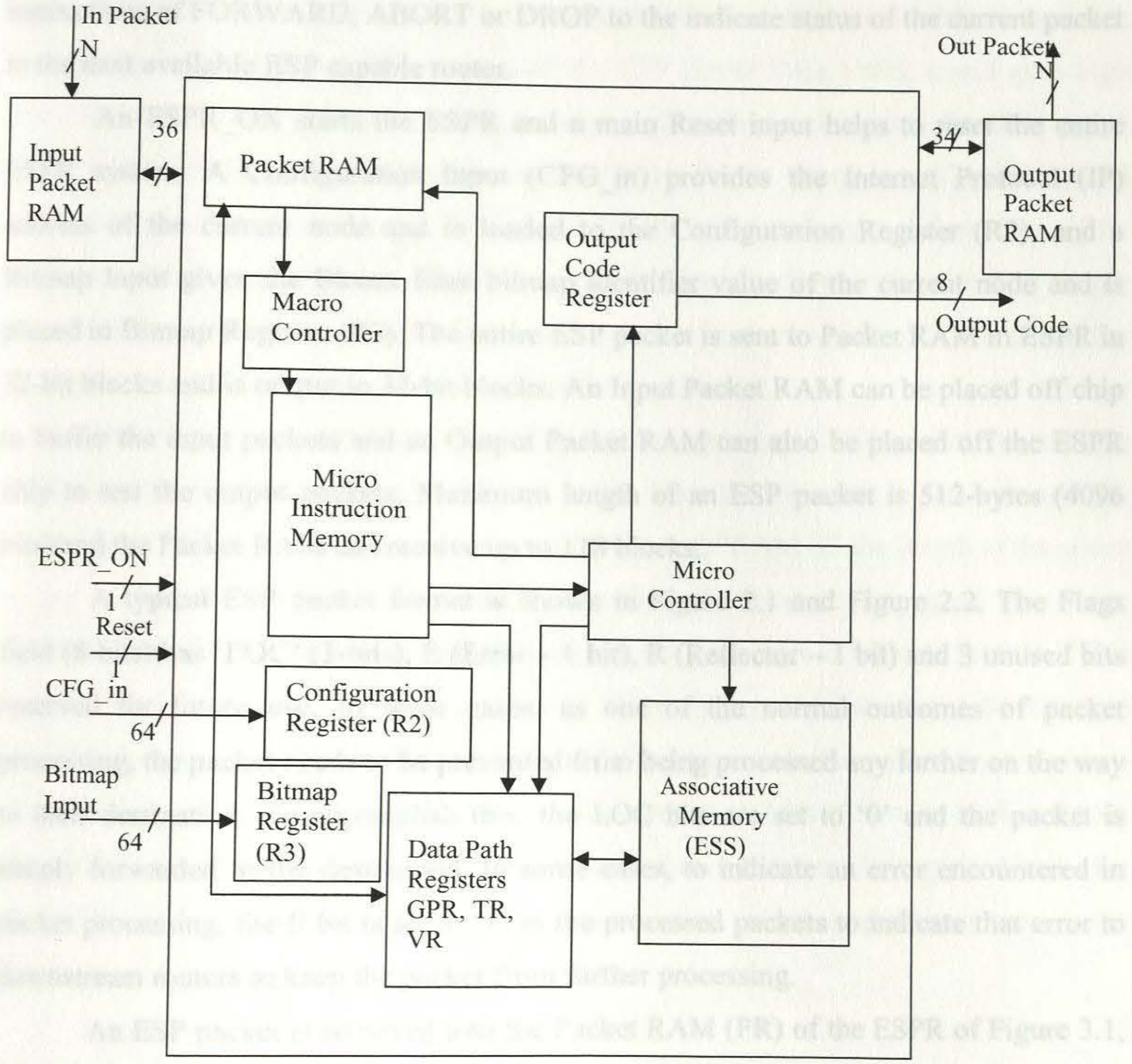
Requirements/characteristics of the ISA of an ESPR can be high-lighted as follows:

- *The number of micro-instructions should be minimized in support of the concept of "lightweight" ESP.*
- *The number of instruction formats should be minimized.*
- *All instructions should be of the same length allowing simplification of the architecture.*
- *A minimum number of addressing modes should be utilized within the instructions.*
- *Most data to be processed is in 64-bit format.*
- *The architecture should offer high performance but yet it should be kept simple (pipelined initial version issuing one instruction per-clock-cycle in the spirit of being "lightweight").*
- *The ESS should be integrated into the ESPR pipelined architecture in a manner to hide latency.*

3.2 Highest Level Functional Organization of ESPR

Based on the previously-presented macro instructions of ESP, an initial ESPR functional organization can be developed as follows. The required functional units of an ESPR will be the ESS, Instruction Memory, Packet Storage RAM, Macro and Micro Controllers, Register blocks and basic processor modules. A high level view of an ESPR illustrating its main functional units is shown in Figure 3.1. Primary inputs and outputs of the system are also shown.

The overall operation of the ESPR within a network node will be as follows. The distinction between an ESP packet and other packets is carried out by external logic inside the router and the ESPR sees only the ESP packets. When ESP is activated, the Packet RAM in ESPR receives the ESP Packet and the Macro Controller decodes the macro opcode in the packet to point to a sequence of micro level instructions held in the Micro Instruction Memory, which must be executed to implement the incoming macro instruction. The remaining ESPR functional modules implement the sequence of micro instructions required to implement a macro operation.



GPR – General Purpose Registers

TR – Tag Registers

VR – Value Registers

ESS – Ephemeral State Store

Figure. 3.1. Functional Units of the ESPR system

Thus, the ESPR processes an incoming packet and the resultant packet is either forwarded or dropped. Primary outputs of the ESPR are the resultant Output Packet if it is forwarded and the resultant Output Code for either DROP, FORWARD or ABORT. The 8-bit Output Code Register (OCR) generates Output Code for the corresponding instructions of FORWARD, ABORT or DROP to the indicate status of the current packet to the next available ESP capable router.

An ESPR_ON starts the ESPR and a main Reset input helps to reset the entire ESPR system. A Configuration Input (CFG_in) provides the Internet Protocol (IP) address of the current node and is loaded to the Configuration Register (R2), and a Bitmap Input gives the Bloom filter bitmap identifier value of the current node and is placed in Bitmap Register (R3). The entire ESP packet is sent to Packet RAM in ESPR in 32-bit blocks and is output in 32-bit blocks. An Input Packet RAM can be placed off chip to buffer the input packets and an Output Packet RAM can also be placed off the ESPR chip to test the output packets. Maximum length of an ESP packet is 512-bytes (4096 bits) and the Packet RAM can receive up to 128 blocks.

A typical ESP packet format is shown in Figure 2.1 and Figure 2.2. The Flags field (8-bits) has 'LOC' (3-bits), E (Error – 1 bit), R (Reflector – 1 bit) and 3 unused bits reserved for future use. In some cases, as one of the normal outcomes of packet processing, the packet needs to be prevented from being processed any further on the way to their destination. To accomplish this, the LOC bits are set to '0' and the packet is simply forwarded to the destination. In some cases, to indicate an error encountered in packet processing, the E bit is set to '1' in the processed packets to indicate that error to downstream routers to keep the packet from further processing.

An ESP packet is retrieved into the Packet RAM (PR) of the ESPR of Figure 3.1, as 32 bit blocks from Input Packet RAM (IPRAM) placed off the ESPR chip and the output processed ESP packet is given to Output Packet RAM (OPRAM). This is shown, focusing on the involved functional units, in Figure 3.2.

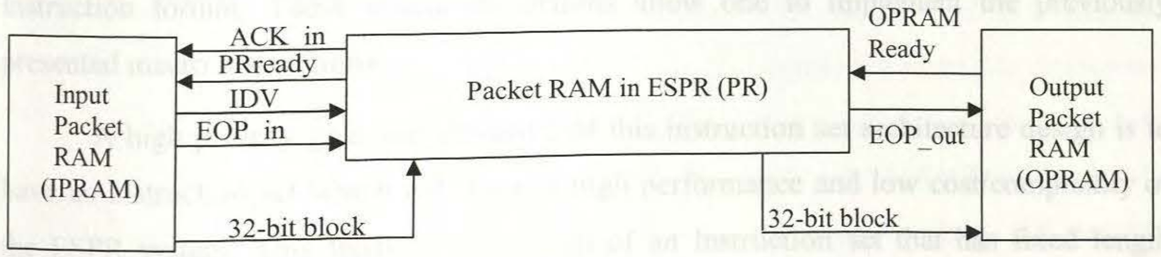


Figure 3.2. Packet Processing in Packet RAM of ESPR

When ESPR is switched on, it is ready to receive and process packets, and then the PR (Packet RAM) in ESPR waits until the IDV (Input Data Valid) signal goes high from IPRAM. When IPRAM is ready to send packet blocks, it asserts the IDV signal high and places 32 bit packet blocks onto the 32-bit Packet Block bus. The IDV signal should remain high for at least 2 blocks and the PR starts receiving packet blocks. The ACK_in (Acknowledge input) signal goes high for every packet block indicating proper receipt of a packet block. When the end of a packet is reached, the IPRAM sends the EOP_in signal with the CRC value for the entire packet.

Similarly, when OPRAM is ready to receive a processed ESP packet, it sends the OPRAMready (Output RAM ready) signal to PR. Then the PR sends the address and 32-bit blocks to OPRAM. The PR sends packet blocks to OPRAM till the length of the entire packet is reached and then sends the EOP_out (End of Packet output) signal to indicate the end of the ESP packet. Then the PRready signal goes high to indicate that the PR is ready to receive the next packet. This packet processing module also has a 64-bit input (not shown here) from a multiplexer so it may choose between the values from registers or different pipeline stages of ESPR when executing the STPR (Store To Packet RAM) micro instruction. It also has a 64-bit output (not shown here) to the register blocks for the LFPR (Load From Packet RAM) micro instruction, which is explained later in the description of the pipelined ESPR architecture.

3.3 Micro Instruction Format, Types, Architecture and Definition

In this section the goals, objectives and approach in designing a basic micro instruction set architecture on basis of the defined macro instructions are discussed. It also covers different instruction types (classes) of micro instructions and their basic instruction format. These micro instructions allow one to implement the previously presented macro instructions.

A high priority goal and objective of this instruction set architecture design is to have an instruction set which will lead to high performance and low cost/complexity of the ESPR system. This leads to the design of an Instruction set that has fixed length instructions, a minimum number of formats and classes. It provides the ESPR with a potential for easy decoding and implementation of the instructions and with less time in decoding and implementation, potentially leading to a high performance and low cost/complexity system.

The use of a 64-bit width for the micro instructions is addressed as follows. The 'Branch' type instructions are identified to use the most number of bits (32) in their instruction format for implementation of the existing ESP macro instructions. The micro instruction sequences required to represent the above presented macro instructions exceed 256 address locations in memory and so a convenient number (16-bit) is used to represent the instruction memory address locations. Considering the evolving growth of ESP and the future possibility of arising additional complicated macro instructions, the micro instruction width was felt to be best set at 64-bits. Also, to achieve the goal of designing a lightweight ESPR, to avoid complexities like register renaming etc., in the design of future ESPR versions and to support the growth of micro opcode and register file(s) size, 64-bit instructions will be supported by the ESPR.

3.3.1. Micro Instruction Format

The basic instruction format for all micro instructions will be as shown in the following Figure 3.3 with the individual field definitions given in Figure 3.4.

OP	RD	RS1	RS2	TR	T	VR	V	U	W	L	S	AIO (16 bit Br./Jmp Addr/Imm/Offset)	SHAMT

Figure 3.3. Micro Instruction Format

The opcode specifies the micro operation. Some of the defined macro instructions require arithmetic, associative and commutative operations that are performed in these micro instructions using operands specified by RS1 and RS2 and the result is written into RD. T and V fields indicate whether TR and VR is used either as a source or destination. The W field specifies the general-purpose register write which indicates that the instruction uses destination register operand RD. The AIO and SHAMT fields are also sometimes required in these operations. ESS is accessed using the fields TR and VR. LMOR is set to '1' when the ESPR encounters an operator <op> in the COMPARE, COLLECT or RCOLLECT macro instruction.

OP – Opcode (6 bits)
RD –Register Destination (5 bits)
RS1 –Register Source 1 (5 bits)
RS2 –Register Source 2 (5 bits)
TR – Tag Register (5 bits)
T – Tag Register Source or Destination (1 bit)
0 – Source, 1 - Destination
VR – Value Register (5 bits)
V – Value Register Source or Destination (1 bit)
0 – Source, 1 - Destination
U – Unused
W – General Purpose Register Write (1 bit)
L – LMOR (Load Micro Opcode Register) (1 bit)
S – Sign bit used in Immediate Type Instructions to denote the sign of the immediate value.
AIO – Address, Immediate, Offset [Address, Immediate Value and Offset (16 bits)]
SHAMT – Shift Amount (6 bits)

Figure 3.4. Field Definitions

3.3.2. Micro Instruction Types (Classes)

The basic micro instruction types (classes) are designed based on fundamental micro operations required to implement the macro instructions and are developed to implement the macro operations correctly and completely. A description of the instruction types and their functionality is as follows. Detailed descriptions and formats of individual micro instructions are described in Appendix A.

3.3.2.1.ALU / SHIFT Type Instructions

The necessity of this type of instruction arises from the COLLECT macro operation, which needs associative and commutative operations. Other macro instructions also need increment and decrement operations. The instructions under this type are, **ADD, SUB, INCR, DECR, OR, AND, EXOR, COMP, SHL, SHR, ROL and ROR.** The instruction format for this type of instruction is shown in the following Figure 3.5.

3.3.2.2.Immediate Type Instruction

The one instruction of this type is **MOVI**. It loads immediate values into registers and its format and definition is as shown in the following Figure 3.6.

63 58 57 53 52 48 47 43 42 25 24 23 6 5 0

OP	RD	RS1	RS2	U	W	U	SHAMT

Instruction	Operation	Description
ADD	Addition	Computes Sum of two operands
SUB	Subtraction	Computes Difference of two operands
INCR	Increment	Increments an operand by 1
DECR	Decrement	Decrements an operand by 1
OR	Logical OR	Logical OR of two operands
AND	Logical AND	Logical AND of two operands
EXOR	Logical EXOR	Logical EXOR of two operands
COMP	Complement	Logical NOT of two operands
SHL	Shift Left	Logical Left Shift
SHR	Shift Right	Logical Right Shift
ROL	Rotate Left	Logical Rotate Left
ROR	Rotate Right	Logical Rotate Right

Figure 3.5. ALU/SHIFT Type Instruction Format and Definition

63 58 57 53 52 24 23 22 21 6 5 0

OP	RD	U	W	S	16 bit Imm Val	U

Instruction	Operation	Description
MOVI	Move Immediate	Moves immediate value to register

Figure 3.6. Immediate Type Instruction Format and Definition

3.3.2.3.Branch / Jump Type Instructions

These instructions check conditions and conditionally execute instructions based on the checked conditions. All macro instructions involve checking conditions based on high-level language constructs such as IF…ELSE. These micro instructions perform similar functions at a lower level. The instructions of this type are **BRNE, BREQ, BRGE, BLT, BNEZ, BEQZ, JMP and RET.** Figure 3.7 shows the format and definition.

63	58 57	53 52	48 47	43 42	22 21	6 5	0
OP		RS1	RS2	U	16 bit Br./Jmp Addr	U	

Instruction	Operation	Description
BRNE	Branch on NOT Equal	Branches to a different location specified by 16-bit Address on inequality of two operand values
BREQ	Branch on Equal	Branches to a different location specified by 16-bit Address on equality of two operand values
BRGE	Branch on Greater or Equal	Branches to a different location specified by 16-bit Address on greater than or equality of two operand values
BLT	Branch on Less Than	Branches to a different location specified by 16-bit Address on comparison of less than operation of two operand values
BNEZ	Branch on NOT Equal to Zero	Branches to a different location specified by 16-bit Address, if the operand value is not equal to zero
BEQZ	Branch on Equal to Zero	Branches to a different location specified by 16-bit Address, if the operand value is equal to zero
JMP	Jump	Jumps to a location specified by 16-bit Address
RET	Return	Returns from a location to the normal PC value

Figure 3.7. Branch/Jump Type Instruction Format and Definition

3.3.2.4.LFPR / STPR Type Instructions

LFPR (Load From Packet RAM) and STPR (Store To Packet RAM) instructions are mainly useful in retrieving/placing information from/to the packet to/from registers. All macro operations require (tag, value) operands in the packet to be retrieved/placed from/to separate registers/Packet RAM. The retrieved values are used to perform local calculations and operations in modules of ESPR. These instructions are used to get/put tag or value from/to specific fields at a particular offset of the packet to/from local General Purpose, Tag or Value Registers (GPR/ TR/ VR). The instructions of this type are **LFPR** and **STPR** which have the format as shown below in Figure 3.8.

63 58 57 53 52 43 42 383736 323130 24232221 6 5 0

OP	RD	U	TR	T	VR	V	U	W	L	U	16 bit Offset	U

Instruction	**Operation**	**Description**
LFPR	Load From Packet RAM	Load value at a particular offset from the packet to register
STPR	Store To Packet RAM	Stores values to a particular offset in packet from a register

Figure 3.8. LFPR/STPR Type Instruction Format and Definition

3.3.2.5.GET / PUT Type Instructions

These instructions are directly equivalent to macro get/put instructions and are useful in detailed accessing of ESS. The **GET** instruction checks to see whether the specified tag exists in ESS, if so checks validity of the value and returns the value if found. The **PUT** instruction places the (tag, value) pair in ESS. The **BGF and BPF** instructions branch to a different location specified by Br.Addr on failure of GET and PUT operations respectively. Figure 3.9 shows the format and definition for GET and PUT instructions.

63 58 57 43 42 383736323130 22 21 6 5 0

OP	U	TR	T	VR	V	U	16 bit Br. Addr	U

Instruction	**Operation**	**Description**
GET	Get	Retrieves the value bound to a tag in ESS
PUT	Put	Places a (tag, value) pair in ESS
BGF	Branch on GET Failed	Branches to a different location specified by 16-bit address on Failure of GET operation
BPF	Branch on PUT Failed	Branches to a different location specified by 16-bit address on Failure of PUT operation

Figure 3.9. GET/PUT Type Instruction Format and Definition

3.3.2.6.Packet Related Instructions

The instructions of this type are **IN, OUT, FWD, DROP, SETLOC, ABORT1 and ABORT2.** These instructions are used to Input, Output, Forward or Drop a packet respectively and the ABORT instructions sets the LOC bits to zero and set/unset the E bit in the packet and then forwards the resultant packet. Its format and definition is shown in Figure 3.10.

63 58	57 0
OP	U

Instruction	Operation	Description
IN	Input	Inputs a packet to Packet RAM
OUT	Output	Outputs resultant code for either DROP or FWD
FWD	Forward	Forwards the packet
DROP	Drop	Drops the packet
ABORT1	Abort	Sets LOC bits to zero in packet and forwards
ABORT2	Abort	Sets LOC bits to zero and E bit to '1' in packet and forwards
SETLOC	Set LOC bits	Sets LOC bits to a specified LOC (Location) value

Figure 3.10. Packet Related Instruction Format and Definition

3.3.3. Further ESPR Architecture Definition

Based on the above-defined Instruction Types/Classes and their formats, additional specific functional units and components of an ESPR system required to complete the definition of its Instruction Set Architecture (ISA) can be defined as follows:

- The ESPR architecture will be Register / Register (R/R), Reduced Instruction Set Computer (RISC) type architecture.
- 32 General purpose 64 bit registers (R0, R1……R31) – 28 available to Programmer (R4, R5……R31)
  - Restricted registers

  - R0 – loaded with '000…..0'
  - R1 - loaded with '000……1'
  - R2 – Configuration Register which holds the node's IP address
  - R3 – Bitmap Register which holds the current node's bitmap identifier value
- PR - Packet RAM to store Input Packets
- 32 – Sixty Four (64) bit Tag Registers (TR) and 32 – Sixty Four (64) bit Value Registers (VR) – 31 available to Programmer (TR1, TR2…….R31), (VR1, VR2……VR31)
  - TR0, VR0 – loaded with '000…..0'
- 8 bit Output Code Register (OCR) to indicate status of the packet in current node
  - 0 – No status, 1 – FWD code, 2 – ABORT1, 3 – DROP, 4 – ABORT2
- 8 bit Flag Register (FLR). FLR consists of the bit pattern to set in flags field of the packet.
- 8-bit Micro Opcode Register (MOR) to store the micro opcode in packet (<op>) instructions particularly used in a defined 'COMPARE', 'COLLECT' and 'RCOLLECT' operation
- Associative Memory – Ephemeral State Store (ESS)
- 64 bit wide Instruction Memory addressed by 16-bit pointer (MAX – 2**16 locations)
- CRC block – To calculate Cyclic Redundancy Check (CRC) of the received packet and to place it back at the end of the packet in PR.
- 2-bit Condition Code Register (CCR) to indicate the Failure of GET and PUT operations in ESS.
- PC – Program Counter
- Macro Controller – Decodes the macro opcode and generates the equivalent micro code location address to PC.
- Micro Controller – Controls the ESPR system at the micro level by generating control signals
- 64 bit ALU and SHIFTER – used in the arithmetic and logical computations
- Decoders and Multiplexers.

3.4 Micro Instruction Implementation of ESP Macro Instructions

The previously presented five ESP macro instructions can now be implemented with sequences of the presented micro instructions which can be shown in the following figures – Figure 3.11 through Figure 3.15.

```
        LFPR <Offset - 3> TR1
        GET VR1, TR1
        BGF Addr1
        INCR R4, VR1
        MOV VR1, R4
        PUT TR1, VR1
        BPF Addr2
Addr3:  LFPR <Offset - 5> R4
        MOV R5, VR1
        BGE R4, R5, Addr4
        DROP
Addr1:  MOV VR1, R0
        PUT TR1, VR1
        BPF Addr2
        JMP Addr3
Addr2:  ABORT2
        OUT
Addr4:  FWD
        OUT
```

Figure 3.11. Equivalent Micro Instruction Sequence for 'COUNT'

```
        LFPR <Offset - 3> TR1
        GET VR1, TR1
        LFPR <Offset - 5> R5
        BGF Addr1
        MOV R4, VR1
        LFPR <Offset - 7> MOR
        NOP
        R4 <OP> R5 Addr1
        DROP
Addr1:  MOV VR1, R5
        PUT TR1, VR1
        BPF Addr2
        FWD
        OUT
Addr2:  ABORT2
        OUT
```

Figure 3.12. Equivalent Micro Instruction Sequence for 'COMPARE'

```
        LFPR <Offset - 3> TR1
        GET VR1, TR1
        BGF Addr1
        LFPR <Offset - 7> TR2
        GET VR2, TR2
        LFPR <Offset - 5> R4
        BGF Addr2
        MOV R5, VR2
        LFPR <Offset - 9> MOR
        NOP
        VR2 ←— R5 <op> R4
        JUMP Addr3
Addr1:  ABORT2
        OUT
Addr2:  MOV VR2, R4
Addr3:  PUT TR2, VR2
        BPF Addr1
        DECR R6, VR1
        MOV VR1, R6
        PUT TR1, VR1
        BPF Addr1
        BEQZ VR1, Addr4
        DROP
Addr4:  STPR <Offset - 5> VR2
        FWD
        OUT
```

Figure 3.13. Equivalent Micro Instruction Sequence for 'COLLECT'

```
        LFPR <Offset - 3> TR2
        GET VR2, TR2
        BGF Addr5
        LFPR <Offset - 7> R8
        MOV R6, VR2
        OR R7, R6, R8
        MOV VR2, R7
        PUT TR2, VR2
        BPF Addr2
Addr0:  LFPR <Offset – 5> TR1
        GET VR1, TR1
        BGF Addr1
        INCR R4, VR1
        MOV VR1, R4
        PUT TR1, VR1
        BPF Addr2
Addr3:  LFPR <Offset – 9> R4
        MOV R5, VR1
        BGE R4, R5, Addr4
        DROP
```

Figure 3.14. Equivalent Micro Instruction Sequence for 'RCHLD'

```
Addr1:  MOV  VR1, R1
        PUT  TR1, VR1
        BPF Addr2
        JUMP Addr3
Addr2:  ABORT2
        OUT
Addr4:  STPR <Offset – 7> R3
        FWD
        OUT
Addr5:  MOV  VR2, R0
        PUT  TR2, VR2
        BPF  Addr2
        JUMP Addr0
```

Figure 3.14. Equivalent Micro Instruction Sequence for 'RCHLD' (continued)

```
        LFPR  <Offset - 3>  TR1
        GET  VR1, TR1
        BGF  Addr1
        LFPR  <Offset - 5>  TR2
        GET  VR2, TR2
        LFPR  <Offset - B>  R4
        BGF  Addr2
        MOV  R5, VR2
        AND  R6, R5, R4
        BEQ  R6, R4,  Addr3
Addr4:  OR  R7, R5, R4
        MOV  VR2, R7
        PUT  TR2, VR2
        BPF  Addr1
Addr3:  LFPR  <Offset – 7> TR3
        GET  VR3, TR3
        LFPR  <Offset – D> R8
        BGF  Addr5
        MOV  R9, VR3
        LFPR  <Offset - F>  MOR
        NOP
        VR3  ←   R8 <op> R9
        JUMP  Addr6
Addr2:  MOV  VR2, R0
        MOV  R5, VR2
        BNEZ  R4, Addr4
        JUMP Addr3
Addr5:  MOV  VR3, R8
Addr6:  PUT  TR3, VR3
        BPF  Addr1
        MOV  R10, VR1
        MOV  R11, VR2
        BEQ  R10, R11, Addr7
        DROP
```

Figure 3.15. Equivalent Micro Instruction Sequence for 'RCOLLECT'

```
Addr7:  LFPR  <Offset – 9> TR4
        GET  VR4, TR4
        BGF  Addr8
        INCR  R12, VR4
        MOV  VR4, R12
        PUT  TR4, VR4
        BPF  Addr1
Addr10:LFPR  <Offset – 10> R13
        MOV  R14, VR4
        BGE  R13, R14, Addr9
        DROP
Addr8:  MOV  VR4, R0
        PUT  TR4, VR4
        BPF  Addr1
        JUMP Addr10
Addr1:  ABORT2
        OUT
Addr9:  STPR  <Offset – B> R3
        FWD
        STPR  <Offset – D> VR3
        OUT
```

Figure 3.15. Equivalent Micro Instruction Sequence for 'RCOLLECT' (continued)

The above presented micro instruction sequences for the five defined macro instructions utilize most of the Packet Related instructions, all of the GET / PUT type instructions, LFPR/STPR type instructions, most of the JUMP/BRANCH type instructions, some (INCR, DECR, OR, AND) of ALU/SHIFT type instructions and the MOV instruction. The rest of the ALU/SHIFT type instructions are included to be utilized in the COMPARE, COLLECT AND RCOLLECT macro instruction. The rest of the unused micro instructions are reserved for future macro instructions.

Chapter Four

Associative Ephemeral State Store (ESS)

A unique requirement of ESP is a temporal Ephemeral State Store (ESS) associative memory where values are bound to tag fields and a (tag, value) pair is active only for a given time period resulting in a reduced capacity store allowing a more light weight and scalable processing system. The ephemeral part of ESS is that the value corresponding to the tag is accessible only for a fixed amount of time and bindings disappear after the Expiration time, 'τ' seconds. Packets leave useful information in the ESS after computations for later packets to retrieve which help in implementing various end-to-end network services. This Chapter discusses the detailed design of ESS and its individual components.

4.1 ESS Design

The ESS design is based on the method of combining some extra logic with a normal random access memory to create associative access. Each location is stored with a Value, Expiration time and a control bit (empty bit – E) for the associated logic. Tags are stored in separate storage space and are used to find whether the required value exists in the ESS. It supports two operations, GET and PUT.

- GET (x): Retrieves the value bound to tag x, if any.
- PUT (x, e): Bind the value e to the tag x.

Depending on the result of GET and PUT, the ESS gives way to support two more operations.

- BGF addr: Branch on GET Failed to address location indicated by 'addr'
- BPF addr: Branch on PUT Failed to address location indicated by 'addr'

The functional blocks of ESS are,

- Block Select Random Access Memory (RAM) used as Content Addressable Memory (CAM)
- Random Access Memory (RAM)
- Expiration time Calculating block
- Empty Location Calculating block
- ESS Controller

A functional level block diagram is shown in Figure 4.1. Primary Inputs to the ESS are TAG, VALUE and GET or PUT operation and the primary outputs are the value (for GET operation) and GET Failed (GF) or PUT Failed (PF) depending on the operation. The main operation of the ESS is as follows. The CAM is used as a storage space for tags and is used to find whether there is a match for the incoming tag. On a match it gives the address for the RAM where the values are stored with its respective expiration time. Depending on the match, values are accessed based on expiration time. The RAM also has a separate empty bit (E) to indicate which location in RAM is empty. This is helpful for the PUT operation when writing a new (tag, value) pair. The empty location-calculating block is used to determine the empty location in RAM and CAM to write a new value and tag based on the empty bits from RAM. A global clock register in the expiration time calculating block is used to check for validity of the (tag, value) pair by comparing its value with the expiration time field in the ESS. The ESS controller generates control signals to all the blocks depending on a GET or PUT operation. Components of ESS can be described as follows.

4.2 Content Addressable Memory (CAM)

A Content Addressable Memory is a storage array designed to quickly find the location of a particular stored value. By comparing the input against the data in memory, a CAM determines if an input value matches a value stored in the array. The basic core of a CAM has a storage location value and a comparator between the storage location value and the input data. The main advantage of a CAM is that its memory size is not limited by its address lines and can be easily expanded. It offers increased data search speed by finding a match and address in a single clock cycle [17].

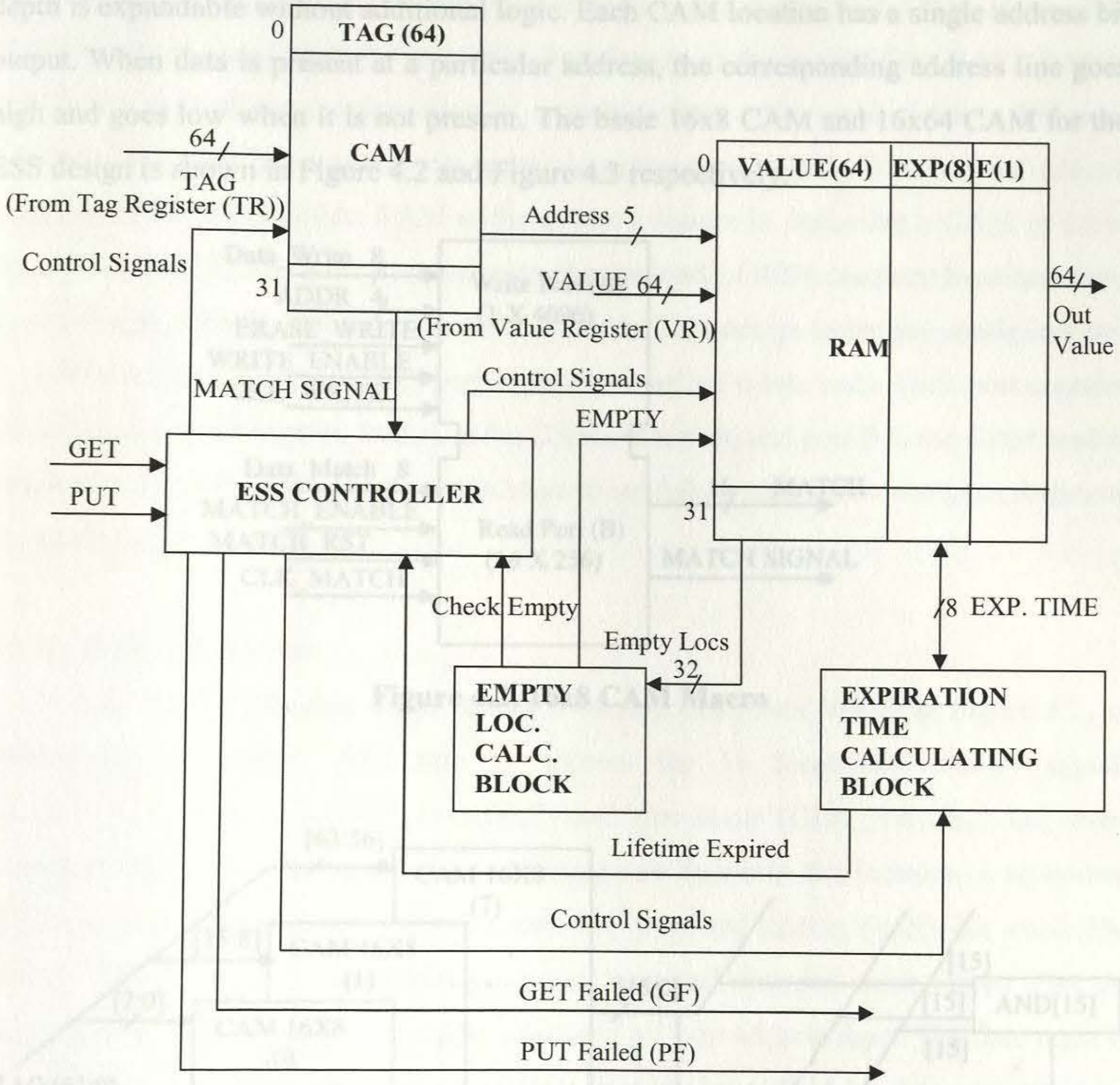


Figure 4.1. Functional Block Diagram of ESS

CAM is used in the ESS design to check whether the (tag, value) pair resides in the ESS by comparing the incoming tag with the tags stored in it. To obtain efficient search of tags and for high performance GET and PUT operations, a Dual-Port Block Select RAM of Virtex FPGA devices will be used in the later presented experimental model of ESPR to operate as a CAM. As per the current design, the CAM is 32x64 (built using two 16x64 CAMs) and the depth can be increased if need be. It is built (width wise) using 8 basic 16x8 block RAM macros and the depth can also be increased in a similar manner by including more basic blocks. As the CAM output is a decoded address, the depth is expandable without additional logic. Each CAM location has a single address bit output. When data is present at a particular address, the corresponding address line goes high and goes low when it is not present. The basic 16x8 CAM and 16x64 CAM for the ESS design is shown in Figure 4.2 and Figure 4.3 respectively.

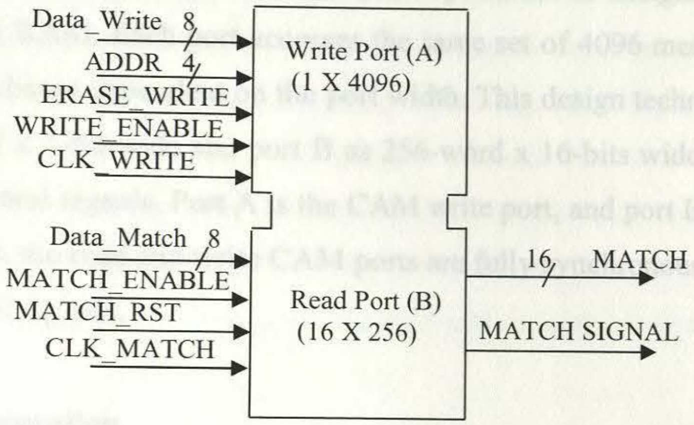


Figure 4.2. 16x8 CAM Macro

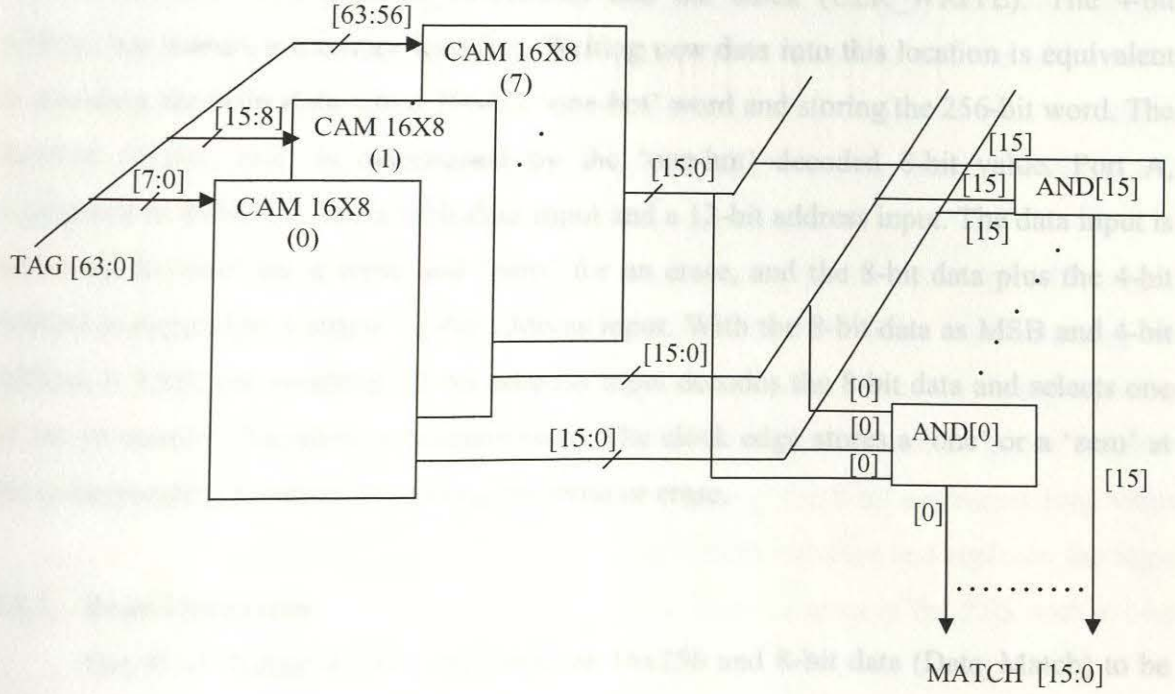


Figure 4.3. 16X64 CAM using 8 16x8 CAMs

The unique Virtex block RAM approach is used to build the 16x8 CAM block. This methodology is based upon the true Dual-Port feature of the block Select RAM. Ports A and B can be configured independently, anywhere from 4096-word x1-bit to 256-word x16-bit. Each port has separate clock inputs and control signals. The internal address mapping of the block Select RAM is the primary feature in designing a CAM in a true Dual-Port block RAM. Each port accesses the same set of 4096 memory locations using an addressing scheme dependent on the port width. This design technique configures port A as 4096-word x 1-bit wide and port B as 256-word x 16-bits wide. Each port contains independent control signals. Port A is the CAM write port, and port B is the CAM read or match port. Both the read and write CAM ports are fully synchronous and have dedicated clock and control signals.

4.2.1. Write Operation

The CAM write port inputs are an 8-bit data bus (Data_Write) in Figure 4.2, an address bus (ADDR - four bits to address the 16 locations), control signals (ERASE_WRITE and WRITE_ENABLE) and the clock (CLK_WRITE). The 4-bit address bus selects a memory location. Writing new data into this location is equivalent to decoding the 8-bit data into a 256-bit 'one-hot' word and storing the 256-bit word. The location of the 'one' is determined by the 'one-hot' decoded 8-bit value. Port A, configured as 4096 x 1, has a 1-bit data input and a 12-bit address input. The data input is addressed to 'one' for a write and 'zero' for an erase, and the 8-bit data plus the 4-bit address is merged in a single 12-bit address input. With the 8-bit data as MSB and 4-bit address as LSB, the resulting 12-bit address input decodes the 8-bit data and selects one of the 16 memory locations simultaneously. The clock edge stores a 'one' or a 'zero' at the corresponding location depending on write or erase.

4.2.2. Read Operation

Port B of Figure 4.2 is configured as 16x256 and 8-bit data (Data_Match) to be searched is connected as an 8-bit address bus. Using the fact that a particular location corresponds to the decoded 8-bit data, the matching operation is equivalent to searching 16 locations for specific 8-bit data at the same time and port B generates the matches concurrently. The MATCH SIGNAL is asserted high when a match occurs and the 16-bit output is the decoded value. MATCH_ENABLE and MATCH_RST are the control signals for port B.

The base 16x64 CAM for this ESS design (32x64) can be obtained by using eight 16x8 CAMs and extra AND gates. Eight 16x8 CAMs allow for a 64-bit width, with the first 8 bits stored in CAM0, next 8 bits in CAM1 and so on. A match is found only if all 8-bit locations match the specified incoming 64-bit tag. An 8-input AND gate for each CAM output signal provides the final decoded address. The 16-bit MATCH output is then encoded to provide the 4-bit Address for ESS where value and expiration time are stored. Currently the ESS design has only 32 locations. This is a sufficient depth to validate the functionality and design of the ESS and to later experimentally validate the ESPR with the ESS included.

4.3 Random Access Memory (RAM) for Storage of Value, Expiration Time and Empty Bit

The RAM storage of Figure 4.1 is used to store value (64-bits), expiration time (8-bits) and an empty bit (1-bit). The current design has 32 locations and the address bits come from the CAM and are used to store and retrieve value, expiration time and empty bit. The empty bit in all locations is set to '1' initially indicating the location is empty and is changed to '0' whenever the location is written with a value.

In case of creation of a new (tag, value) binding in the ESS, the 1-bit empty location value is checked for the availability of space in ESS rather than checking the already existing 8-bit expiration time values of all locations. Thus this compromising solution of an additional 1-bit space for each location in the ESS and an empty location check on these 1-bit values are preferred over comparing the 8-bit expiration time value of all locations with a value of zero. It also significantly reduces and replaces the logic overhead involved in having 8-bit comparators for each location of the ESS with a 1-bit value check on each location.

4.4 Expiration Time Calculating Block

This block is shown in Figure 4.1 and 4.4. A counter operating as a very low frequency clock functions as a global clock register (see Figure 4.4). Whenever a value is written (corresponding to any new Tag) to any location, expiration time is calculated by adding the global clock register value with the lifetime 'τ' and it is written in the expiration time field in the RAM. The validity of the value in the RAM is checked by comparing whether the entry in the expiration time field is less than the global clock register value.

When 8-bits are used to represent the expiration time values, there may be possibilities of 'wrap around' situations, in which case, the values may incorrectly be in the ESS for a longer time. As this is an initial functionality testing version of ESPR, 8-bits is used to represent the expiration time value to check the functionality of the expiration time calculating block where [8] suggests 10-bit values are sufficient for the assumption of a 10 second lifetime and a 0.1 resolution clock. In order for correct functional operation of ESS in an experimental deployment, expiration time should be represented by a larger number of bits (e.g.: 10 bits or more) to avoid the possibility of 'wrap around' situations.

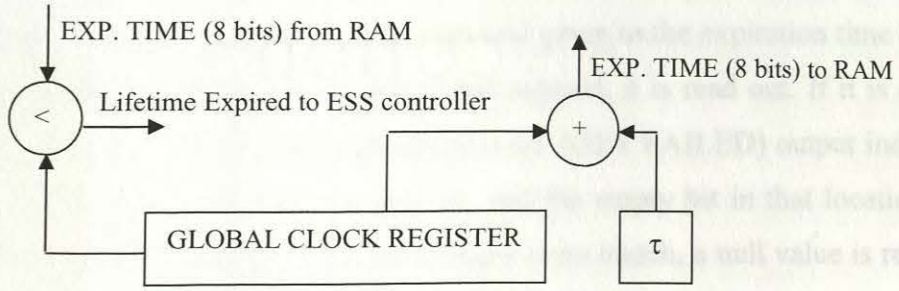

EXP. TIME (8 bits) from RAM
EXP. TIME (8 bits) to RAM
<
Lifetime Expired to ESS controller
+
GLOBAL CLOCK REGISTER
τ

Figure 4.4. Expiration Time Calculating Block

4.5 Empty Location Calculating Block

This block (see Figure 4.1) is used to determine the new location to write value and expiration time in RAM and tag in CAM. The empty bits from RAM are input to this block and it determines the output address bits to write a new (tag, value) pair.

4.6 ESS Controller

Depending on the GET or PUT operations, the controller of Figure 4.1 generates control signals to all the blocks in ESS. The inputs to the controller are GET or PUT operation from the instruction decode stage of ESPR, the check empty signal from the empty location calculating block, the lifetime expired signal from the expiration time calculating block and the MATCH SIGNAL from the CAM. Two outputs are the control signals - GF (GET FAILED) or PF (PUT FAILED). WRITE_ENABLE, ERASE_WRITE, MATCH_RST and MATCH_ENABLE are output control signals to the CAM, we (write enable) is a control signal to RAM and cnt (count) is a control signal to the Expiration time calculating block.

4.7 Operations Performed in ESS and Flowchart

The ESS operations – Flowchart shown in Figure 4.5 describes GET and PUT operation as will be described here and below.

4.7.1. GET Operation

The incoming tag from the tag register is given to the CAM to check for a match and for the availability of a value in the ESS. If a match occurs, the CAM asserts the MATCH SIGNAL high and gives the address to the RAM to get the value. The expiration time in that address is read out and given to the expiration time block to check the validity of the data. If the value is not expired, it is read out. If it is expired, a null value is returned and the controller gives a GF (GET FAILED) output indicating a GET failure. This location is then cleaned up, and the empty bit in that location is set to '1' indicating that the location is empty. If there is no match, a null value is returned and the controller generates a GF (GET FAILED) output indicating a GET failure.

4.7.2. PUT Operation

The incoming tag from the tag register is given to the CAM to check whether the (tag, value) binding already exists in the ESS. If a match occurs, the CAM asserts the MATCH SIGNAL high and gives the address to the RAM to get the value. The expiration time in that address is read out and given to the expiration time block to check the validity of the data. If the value is not expired, a new value is written in that location and the empty bit is set to '0'. If the value is expired, a new value is written, the empty bit is set to '0' and the expiration time is reset again in that location. If there is no match, the empty location-calculating block checks to find the empty location to write a new tag and value. If there is an empty location, the address of the new location is given to the CAM to write the tag and given to RAM to write a value, expiration time and the empty bit is set to '0'. If there is no empty location, the controller generates a PF (PUT FAILED) output indicating a PUT failure. Whenever the PUT operation is completed successfully the empty bit of the location is reset to '0' indicating that the location is filled.

4.7.3. Branch on GET Failed (BGF) / Branch on PUT Failed (BPF) Operation

The BGF / BPF micro instructions branch to a 16-bit address specified in the instruction, on failure of GET / PUT respectively. These micro instructions are actually performed by the Branch Detection Unit of the ESPR depending on the result of ESS operations.

4.8 ESS Scalability, Size and Performance

The ESS organization, architecture and design is a key functional unit related to the performance and scalability of the ESP service. We now discuss the scalability and performance of the current ESS design. The main components of the ESS described above are CAM and RAM, and the scalability has to be defined in terms of these memories. The CAM and RAM memories described above can be implemented using core block RAM of PLD technology chips which the ESPR would be implemented to. The size of these memories can be expanded by adding the required core RAM on-chip, by adding the required bits in the existing design, without any change to the existing controller design.

Thus the presented design for ESS is scalable, dependent upon the capacity of block RAM memory available in PLD chips and the depth of ESS can be extended accordingly to that. Under this limitation the same organization, architecture and design for ESS can be used to implement ESS off chip. Therefore, an off-chip implementation of ESS is also scalable. The price paid here is a slight decrease in performance of ESS because of the time required by the signals to travel through the additional circuitry in reconfigurable PLD chips.

The Current ESS design only has 32 (32x137) locations. This was sufficient to allow its functional validation. The same design can be expanded to a deeper ESS assuming sufficient on-chip core RAM. For ESPRs implemented to PLD technology, the core RAM determines the size and performance of ESS, and it can be tuned as necessary by the utilizing application by including additional block RAM in the design.

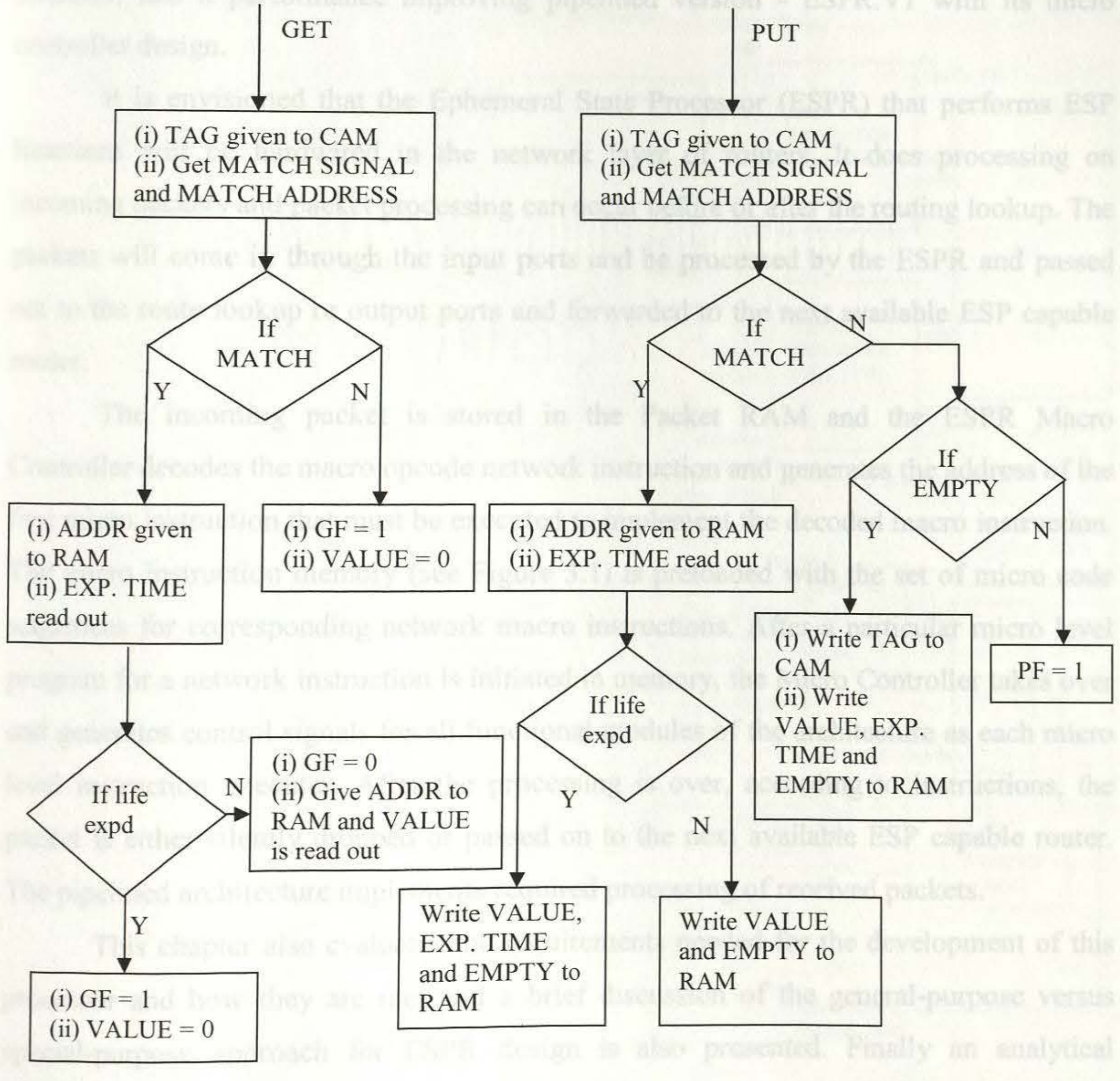


Figure 4.5. ESS Operations – Flow Chart

Chapter Five

Ephemeral State Processor Version 1 (ESPR.V1) Architecture

This chapter deals with the development of the Ephemeral State Processor Architecture – Version 1. Later in this thesis, to improve the performance needed, a second version of ESPR will be developed and tested. The chapter describes the overall system architecture design of ESPR including all connectivity between functional modules, and a performance improving pipelined version - ESPR.V1 with its micro controller design.

It is envisioned that the Ephemeral State Processor (ESPR) that performs ESP functions will be hardwired in the network layer of routers. It does processing on incoming packets and packet processing can occur before or after the routing lookup. The packets will come in through the input ports and be processed by the ESPR and passed out to the route lookup or output ports and forwarded to the next available ESP capable router.

The incoming packet is stored in the Packet RAM and the ESPR Macro Controller decodes the macro opcode network instruction and generates the address of the first micro instruction that must be executed to implement the decoded macro instruction. The micro instruction memory (see Figure 3.1) is preloaded with the set of micro code sequences for corresponding network macro instructions. After a particular micro level program for a network instruction is initiated in memory, the Micro Controller takes over and generates control signals for all functional modules of the architecture as each micro level instruction executes. After the processing is over, according to instructions, the packet is either silently dropped or passed on to the next available ESP capable router. The pipelined architecture implements required processing of received packets.

This chapter also evaluates the requirements needed for the development of this processor and how they are met and a brief discussion of the general-purpose versus special-purpose approach for ESPR design is also presented. Finally an analytical performance model for ESPR is devised and presented.

5.1 Basic Register/Register Architecture Development with ESPR Components

The Micro Instruction Memory of the basic ESPR architecture shown in Figure 5.1 is preloaded with the micro code sequences for corresponding network macro instructions. The incoming packet is loaded into the Packet RAM (PR in Figure 5.1). The

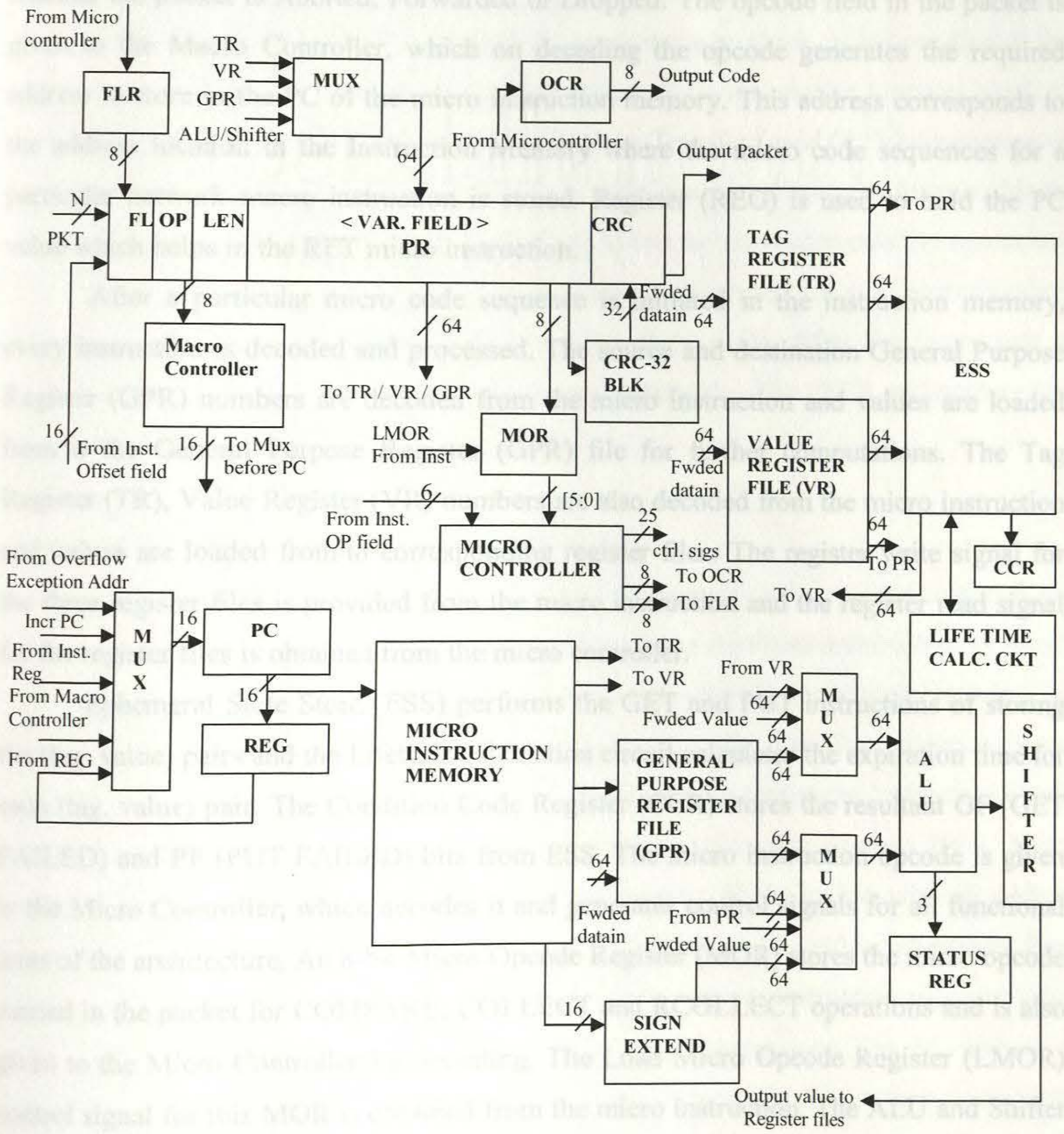


Figure 5.1. Basic ESPR Architecture

Flag Register (FLR) is an 8-bit register which holds the corresponding bit patterns for the setting of 'LOC' and 'E' (Error) bits in the packet and it is always given to the Flag field in the first location of Packet RAM. The CRC-32 block calculates the Cyclic Redundancy Check code (CRC) using CRC-32 polynomial and places the resultant CRC code back in the packet. The Output Code Register (OCR) generates output code depending on whether the packet is Aborted, Forwarded or Dropped. The opcode field in the packet is given to the Macro Controller, which on decoding the opcode generates the required address to store in the PC of the micro instruction memory. This address corresponds to the address location in the Instruction Memory where the micro code sequences for a particular network macro instruction is stored. Register (REG) is used to hold the PC value which helps in the RET micro instruction.

After a particular micro code sequence is initiated in the instruction memory, every instruction is decoded and processed. The source and destination General Purpose Register (GPR) numbers are decoded from the micro instruction and values are loaded from/to the General Purpose Register (GPR) file for further computations. The Tag Register (TR), Value Register (VR) numbers are also decoded from the micro instruction and values are loaded from/to corresponding register files. The register write signal for the three register files is provided from the micro instruction and the register read signal for the register files is obtained from the micro controller.

Ephemeral State Store (ESS) performs the GET and PUT instructions of storing the (tag, value) pairs and the Lifetime calculation circuit calculates the expiration time for each (tag, value) pair. The Condition Code Register (CCR) stores the resultant GF (GET FAILED) and PF (PUT FAILED) bits from ESS. The micro instruction opcode is given to the Micro Controller, which decodes it and generates control signals for all functional units of the architecture. An 8-bit Micro Opcode Register (MOR) stores the micro opcode carried in the packet for COMPARE, COLLECT and RCOLLECT operations and is also given to the Micro Controller for decoding. The Load Micro Opcode Register (LMOR) control signal for this MOR is obtained from the micro instruction. The ALU and Shifter perform the arithmetic and logical computations and store the result back into registers. The ALU provides overflow, sign and zero status signals that are stored in a 3-bit status register. On an overflow exception, the PC is loaded with a specific address by which the microcode sequence aborts and the processing stops. The SIGN EXTEND unit is used to extend the 16-bit value to 64-bit value which helps in the MOVI micro instruction.

The Macro level system flowchart in Appendix B shows the overall macro level operation of the ESPR of Figure 5.1. as packets arrive at the input and the appropriate macro level instruction is executed. Each step in the macro instruction flowchart corresponds to the execution of a micro instruction. The Micro level system flow charts in Appendix C and Appendix D represents the clock cycle by clock cycle operation of the micro instructions as they are fetched and executed. Each rectangular function block of the micro level system flow chart contains a Register Transfer Level (RTL) description of the micro operations executed during the clock cycle associated with the function block. The first two function blocks common to all micro instructions represents the instruction fetching and each micro operation(s) in each block represent activation of corresponding functional modules in the architecture for every clock cycle. After the instruction is fetched each instruction is decoded and executed separately. The ESPR architecture and system flow charts are developed concurrently.

5.2 Four Stage Pipelined Architecture (ESPR.V1)

To improve the performance of the ESPR by increasing the processing speed, the basic ESPR architecture of Figure 5.1 is transformed to a pipelined architecture as shown in Figure 5.2. It is a 4-stage pipeline with Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute (EX) and Write Back (WB) stages. All instructions advance during each clock cycle from one pipeline register to the next. The first stage, Instruction Fetch, is common to all instructions. This stage contains the Program Counter (PC), Micro Instruction Memory, Register (REG) and a multiplexer. PC is loaded with an address from the multiplexer depending on micro/macro instructions, overflow exception from ALU or incremented PC value. Instructions are read from the instruction memory using the address in PC and then placed in the IF/ID pipeline register. The PC address is incremented by one and then loaded back into the PC to be ready for the next clock cycle. REG is used to hold the address from PC whenever the JMP micro instruction is encountered and used to restore the address back in PC whenever the RET micro

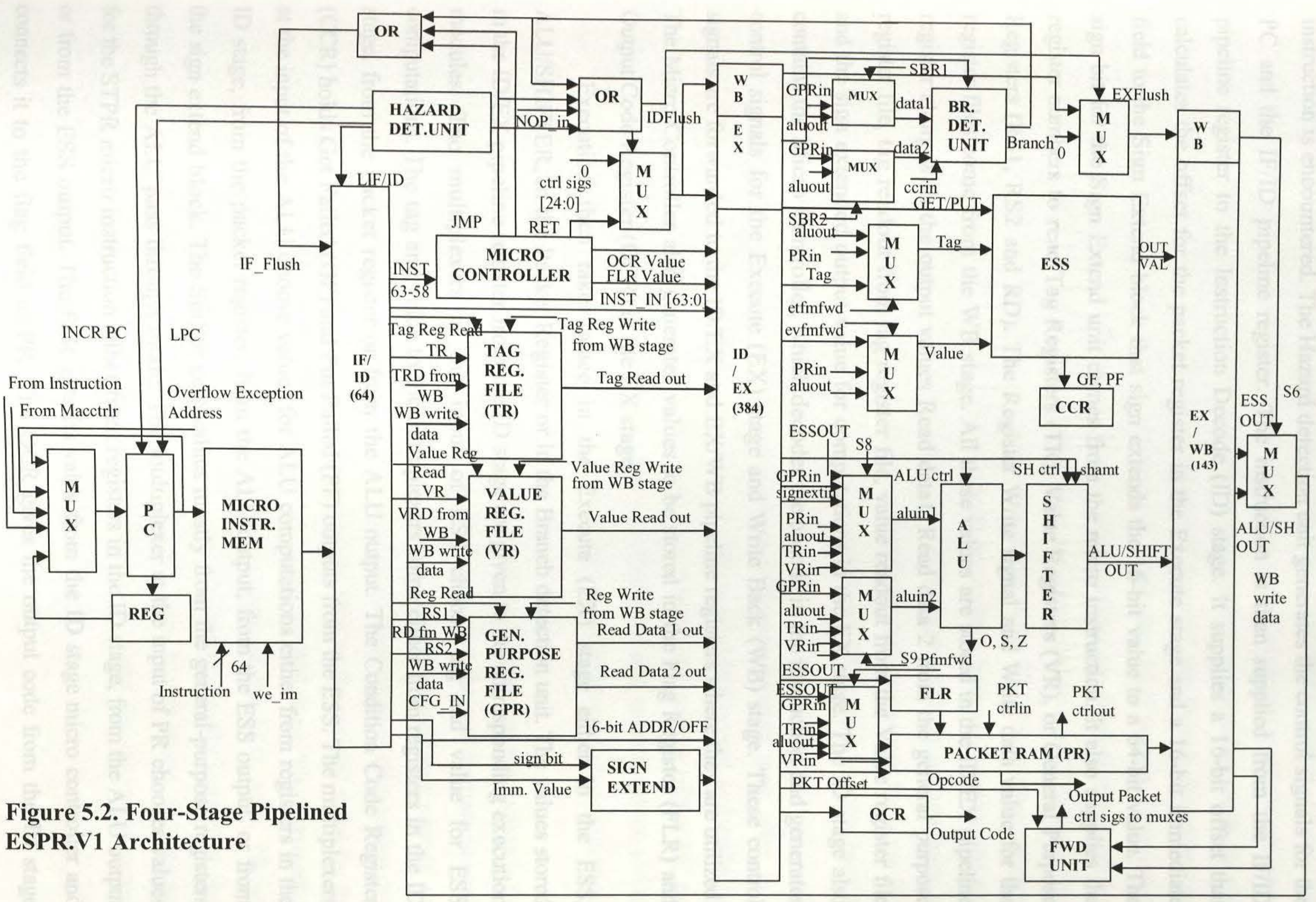


Figure 5.2. Four-Stage Pipelined ESPR.V1 Architecture

instruction is encountered. The Hazard detection unit generates the control signals for the PC and the IF/ID pipeline register. The instruction is then supplied from the IF/ID pipeline register to the Instruction Decode (ID) stage. It supplies a 16-bit offset that calculates the offset for the packet register in the Execute stage and a 16-bit immediate field to the Sign Extend block that sign extends the 16-bit value to a 64-bit value. The sign bit for the Sign Extend unit comes from the micro instruction. It also supplies the register numbers to read Tag Registers (TR), Value Registers (VR), or General Purpose Registers (RS1, RS2 and RD). The Register Write signal and Write data value for the register files come from the WB stage. All these values are stored in the ID/EX pipeline register along with the output values Read data1, Read data 2 from the general purpose register file, tag readout from tag register file, value readout from the Value register file and the sign extended output value for computations in the EX stage. The ID stage also contains the Micro Controller, which decodes the opcode in the instruction and generates control signals for the Execute (EX) stage and Write Back (WB) stage. These control signals are forwarded to the ID/EX and EX/WB pipeline registers where they are utilized. The Micro Controller also generates values to be stored in the Flag Register (FLR) and Output Code Register (OCR) in the EX stage.

Execution then takes place in the Execute (EX) stage either in the ESS, ALU/SHIFTER, in the Packet Register or in the Branch detection unit. The values stored in the ID/EX pipeline register from the ID stage are given to the corresponding execution modules. The multiplexers at the input of ESS choose tag and value for ESS computations. The tag and value to the multiplexers come either from registers in the ID stage, from the packet register or from the ALU output. The Condition Code Register (CCR) holds Get Failed (GF) and Put Failed (PF) outputs from the ESS. The multiplexers at the input of the ALU choose values for ALU computations either from registers in the ID stage, from the packet register, from the ALU output, from the ESS output, or from the sign extend block. The Shifter gets values mostly from the general-purpose registers through the ALU pass through mode. The multiplexer at the input of PR chooses values for the STPR micro instruction either from registers in the ID stage, from the ALU output or from the ESS output. The FLR gets its value from the ID stage micro controller and connects it to the flag field of PR. The OCR gives the output code from the ID stage micro controller to an output port. The jump and conditional branch type micro instructions are executed using the Branch detection unit. Two register values are given as input to the branch detection unit to check for the equality or inequality depending on the type of micro instructions. The multiplexers in front of the Branch detection unit choose value from general-purpose registers or from the ALU output. The micro controller generated control signals for the execution modules are given to the respective modules and the control signals for the WB stage are forwarded to the EX/WB pipeline register. The resultant values of execution are also stored in the EX/WB pipeline register.

After the execution of instructions, results are written back to registers and this takes place in the Write Back (WB) stage. The WB stage result is written back to registers using a multiplexer. The control signal for this multiplexer comes from the WB stage control signal and it chooses between ALU output and ESS output to write back to registers in the ID stage.

Potential hazards such as Data hazards and Branch hazards may arise in a pipelined architecture. The hazard detection unit detects any data hazard and stalls the pipeline when necessary. This hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexers that choose between the real control values and all 0s. A multiplexer in the ID stage and EX stage is used to reset the control signals to '0' for stalls.

A data hazard is detected when the write register of the previous instruction is the same as the read register of the next instruction. So in this case, the next instruction reads the wrong value of the read register because the write register would not contain the correct value in this stage. The forwarding unit in the EX stage helps in eliminating data hazards by forwarding the result from the ALU output back as the register value for the next instruction instead of waiting to get the result from the WB stage. This forwarding unit generates control signals for the multiplexers in front of the ALU, ESS, PR and Branch detection unit to choose the value from the ALU output directly instead of from the register input. The WB control signals, opcode from the ID stage and register numbers are given to the forwarding unit that helps to forward the result for correct execution.

One solution to resolve a branch hazard is to stall the pipeline until the branch is complete. But on the other hand a common improvement over stalling upon fetching a branch is to assume the branch will not be taken and so will continue to execute down the sequential instruction stream. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. To discard the instructions the controller flushes the instructions in the IF, ID and EX stages of the pipeline. After the execution of a branch condition in the Branch detection unit and if the branch has to be taken, multiplexer in front of the PC helps in choosing the new branch target address. To flush instructions in the IF stage, a control line called IF Flush is added, which resets the instruction field of the IF/ID pipeline register to '0' to flush the fetched instruction. A control signal called IDFlush is used to flush instructions in the ID stage. The EXFlush control signal is used to flush the already executed instructions in the EX stage. The micro controller determines whether to send a flush signal depending on the instruction opcode and the value of the branch condition being tested.

The pipelined architecture system flow chart in Appendix C shows the stage-by-stage operation of all the micro instructions in a pipelined architecture. Most of the instructions take a single execution phase. The ESS (GET / PUT) instructions may take more than one clock cycle (at most 5 clock cycles) to execute. So the ESS has to operate at five times the frequency of the overall ESPR.

5.3 Micro Controller

The Micro Controller is located in the ID stage of the pipeline and will be required to generate 25 control signals to implement all defined micro instructions. The final Micro Controller may be predominantly pipelined combinational logic whose input is the Opcode (6 bits) and whose outputs are the control signals identified within this section. It generates control signals for the ID stage, EX stage and WB stage. The ID stage control signals are REGREAD, JMPINST and RETINST. REGREAD is supplied to the General Purpose, Tag and Value Register files. JMPINST and RETINST are used to determine the flushing of pipeline stage registers.

The EX stage control signals are given to the Packet processing unit for the PR, ESS controller in ESS, ALU, Shifter and Branch detection unit. The control signals for the Packet processing unit are LFPRINST, STPRINST, ININST, OUTINST, LDPKREG, LDOCR and LDFLR. LFPRINST, STPRINST and ININST correspond to the micro instructions LFPR, STPR and IN. The control signal OUTINST corresponds to the OUT micro instruction. LDPKEG, LDOCR and LDFLR are given to the Packet RAM (PR), Output Code Register (OCR) and Flag Register (FLR) respectively. The control signals for the ESS unit are GETINST, PUTINST and LDCCR. GETINST and PUTINST signals are given to the ESS controller to perform GET and PUT operations. LDCCR is the control signal for the Condition Code Register (CCR). The Shifter control signals are S0, S1 and S2 and ALU control signals are S3, S4, S5 and Ci. The function table for the Shifter, ALU and Branch detection unit are shown in Tables 5.1, Table 5.2 and Table 5.3 respectively.

Table 5.1. Function Table for Shifter

CTRL SIGS (S0, S1, S2)	OPERATION
000	PASS THROUGH
001	SHIFT LEFT (SHL)
010	SHIFT RIGHT (SHR)
011	ROTATE LEFT (ROL)
100	ROTATE RIGHT (ROR)

Table 5.2. Function Table for ALU

CTRL SIGS (S3, S4, S5, Ci)	OPERATION
0000	PASS THROUGH for a
0001	PASS THROUGH for b
0010	ONES COMPLEMENT for a
0011	ONES COMPLEMENT for b
0100	ADD
0101	SUB
0110	INCR a
0111	DECR a
1000	INCR b
1001	DECR b
1010	OR
1011	AND
1100	EXOR
1101	NEGATIVE of a
1110	NEGATIVE of b

Table 5.3. Function Table for Branch detection unit

CTRL SIGS (BRANCH TYPE)	OPERATION
000	BLT
001	BRNE
010	BREQ
011	BRGE
100	BNEZ
101	BEQZ
110	BGF
111	BPF

The WB stage control signals generated by the micro controller are S6 and REGWRITE. The S6 control signal is given to the multiplexer in the WB stage to choose between the ALU and ESS outputs. The REGWRITE control signal is connected back to the General Purpose Register file in the ID stage. Two additional control signals for the WB stage, TAG REGISTER WRITE and VALUE REGISTER WRITE, comes from the micro instruction and are given to the Tag Register file and Value Register file respectively. The active control signals involved in the proper execution of each micro instruction are shown below in Table 5.4. For each micro instruction the remaining control signals apart from the active ones are interpreted as inactive during its execution. Each control signal is a control point and identified within the final ESPR.V1 architecture shown in Figure 5.2.

5.4 ESPR.V1 Requirements Evaluation

In designing a processor to handle special functions there comes the question of choosing between the options available for design: either designing a special functionality Coprocessor to perform special functions which can be connected to a general purpose processor to handle other general purpose functions or designing a stand alone special purpose processor. This section discusses requirements evaluation of the Ephemeral State Processor and the next section discusses the options available in designing ESPR. The components and functional unit requirements of the ESPR system defined in Chapter 3 – Section 3.1 can be evaluated as follows to give support to the architectural design of ESPR.

Table 5.4. Control Signals for Micro Instructions

MICRO INSTRUCTIONS	CONTROL SIGNALS
NOP	No Active Signals
IN	ININST, LDPKREG
OUT	OUTINST
FWD	LDOCR, OCR = 0000 0001
ABORT1	LDOCR, LDFLR, OCR=00000010, FLR=00000000
DROP	LDOCR, OCR = 0000 0011
CLR	REGREAD, REGWRITE, S6
MOVE	REGREAD, REGWRITE, S6
MOVI	REGWRITE, S6
ADD	S3, S4, S5, Ci, REGREAD, REGWRITE, S6
SUB	S3, S4, S5, Ci, REGREAD, REGWRITE, S6
INCR	S3, S4, S5, Ci, REGREAD, REGWRITE, S6
DECR	S3, S4, S5, Ci, REGREAD, REGWRITE, S6
OR	S3, S4, S5, Ci, REGREAD, REGWRITE, S6
AND	S3, S4, S5, Ci, REGREAD, REGWRITE, S6
EXOR	S3, S4, S5, Ci, REGREAD, REGWRITE, S6
COMP	S3, S4, S5, Ci, REGREAD, REGWRITE, S6
SHL	S0, S1, S2, REGREAD, REGWRITE, S6
SHR	S0, S1, S2, REGREAD, REGWRITE, S6
ROL	S0, S1, S2, REGREAD, REGWRITE, S6
ROR	S0, S1, S2, REGREAD, REGWRITE, S6
LFPR	LFPRINST, REGREAD, REGWRITE, S6
STPR	STPRINST, REGREAD
BRNE	BRANCH TYPE, REGREAD
BREQ	BRANCH TYPE, REGREAD
BRGE	BRANCH TYPE, REGREAD
BNEZ	BRANCH TYPE, REGREAD
BEQZ	BRANCH TYPE, REGREAD
JMP	JMPINST
RET	RETINST
GET	GETINST, REGREAD, LDCCR
PUT	PUTINST, REGREAD, LDCCR
BGF	BRANCH TYPE, REGREAD
BPF	BRANCH TYPE, REGREAD
ABORT2	LDOCR, LDFLR, OCR=00000100, FLR=00000001
BLT	BRANCH TYPE, REGREAD
SETLOC	LDFLR

The conceptual description of Ephemeral State Processing (ESP) requires no data memory in the design except the Ephemeral State Store (ESS), and so the ESPR is designed as a basic Register/Register Architecture. Based on the ESP requirements described in Chapter 2, the main component of design in designing the ESPR is the ESS, and a scalable associative memory is designed to meet this requirement. To differentiate special operations carried out on tags and values, a separate tag register file and value register file are included along with the general-purpose register file for normal operations. With the current set of macro level instructions, the number of registers is designed to be 32 for validation purposes. This set may grow over time depending upon the ESP design requirements. The existing unused bits in the micro instruction can be used to add register numbers. The tags and values in ESP are 64-bit wide and so the registers are designed to hold 64-bit wide values. All the basic operations in the ESPR are carried out on 64-bit wide operands and so the ALU, Shifter and rest of the components are designed to handle operands in this manner.

A RAM (Packet RAM - PR) is designed for storing and processing incoming packets. PR is designed to have 128 locations of each 32 bits wide. This is designed based on maximum packet size and incoming packet block width. Also offset handling is easy in RAM because the operands in a packet are either 8 or 64 bits wide. Thus a RAM is used to store packets to provide efficient PLD resource utilization. As the status of the current packet, after processing, has to be indicated to the next node, an Output Code Register is used to store the status of the current ESP packet. The Flag Register is used to operate on flag fields individually. The Micro Opcode Register is used to store the micro opcode carried in a packet which may be further used for 'COMPARE', 'COLLECT' or 'RCOLLECT' macro instructions. The Status Register is used to hold the status after ALU operation and the Condition Code Register is used to hold the status after ESS operation.

A separate block is needed for calculation of Cyclic Redundancy Check (CRC) which may be helpful to detect whether an error occurred when the packet is received before processing (Refer Appendix – B). Macro and Micro Controllers are necessary for their respective control operations. As network macro instructions grow over time, there might be a future necessity to have further more additional micro instructions and additional registers. To reflect this and also that basic operations are over 64-bit values, the micro instruction width is chosen to be 64-bits wide. Additional components such as the program counter, decoders and multiplexers meet basic requirements for a processor design.

5.5 Special-Purpose Versus General-Purpose approach to ESP

The two options available in designing ESPR are,

- Designing a special-purpose processor for ESP as the one described in this Chapter
- Designing a special functionality coprocessor that can be linked to a general purpose processor

A special-purpose processor can be designed as the pipelined architecture as described in Section 5.2 in this Chapter. The second alternative is to first design a coprocessor that handles functions corresponding mainly to ESS and the Packet RAM with its control unit. And then have a general-purpose processor connected to this coprocessor module, both combined together to perform ESP.

Referring to the ESPR system requirements, most of the main components are special functional units, which involves almost all of ESS, Packet RAM, Tag and Value register file, Macro controller, Output Code Register, Flag Register, Micro Opcode Register, Condition Code register and CRC block. And also, most of the micro instructions in the micro instruction sequence representation of macro instructions utilize most of the special functional units described above, mainly ESS and the Packet Register. The micro instruction sequences for macro instructions involving general-purpose components are few. The instruction set is defined to handle a lot of general-purpose type instructions only in considering future necessity. Considering the above ESPR requirements and also to eliminate I/O overhead between the general-purpose processor and coprocessor, an initial step was taken in designing a special-purpose pipelined processor for ESP over a general-purpose approach. A comparison of cost and performance of the special-purpose vs general-purpose ESPR has not been conducted. The special purpose ESPR design is an initial step towards implementing ESP in routers.

5.6 Analytical Performance Model for ESPR

The performance of any processor can be measured by the time it takes to complete a specific task, which is commonly described as CPU execution time. CPU execution time depends on how fast the hardware can complete basic functions, which in turn can be a function of the clock frequency at which the processor performs its operations in addition to other factors. A simple formula that defines the basic performance measure – CPU execution time [27], for a processor that performs its operations sequentially can be given as,

CPU execution time for a program =

*(Instruction Count for a program) * (avg. clock cycles per instruction) **

(clock cycle time)……………………………………………………………(1)

A program is comprised of a number of instructions and in a sequential processor, each instruction takes a different number of clock cycles (more than one) to complete its required function, so the term average clock cycles per instruction is used, and clock cycle time is the basic clock cycle period for the processor. The above equation makes it clear that, the performance can be improved by reducing either the clock cycle period or the number of clock cycles required for a program. As per this, performance improvement can be obtained by pipelined implementation of the processor – as was done for the ESPR Thus pipelining reduces the average execution time per instruction; however there is degradation in the expected performance of pipelined processors due to pipeline stalls. And if the stages of the pipeline are perfectly balanced, then the time per instruction in a pipelined machine [27] is equal to,

Time per instruction on unpipelined machine / Number of pipe stages………………………………………………………………………… (2)

Under these conditions, the speed up from pipelining equals the number of pipe stages. The ideal CPI (clock cycles per instruction) for a pipelined machine [27] can be given as,

Ideal CPI =

Number of clock cycles per instruction on an unpipelined machine / Number of pipe stages……………………………………………………………………… (3)

The ideal CPI is almost always 1. The pipelined CPI [27] is the sum of the base CPI and a contribution from stalls.

Pipeline CPI = Ideal CPI + Pipeline stall cycles per instruction…………. (4)

By reducing the terms on the right hand side, the overall Pipeline CPI can be reduced and thus increasing the instruction throughput per clock cycle. The CPU execution time of a program for a pipelined processor [27] can now be given as,

(CPU execution time) $_{pipelined}$ *=*

*(Number of clock cycles for instruction count + stall cycles) * (Ideal CPI) **

(pipelined clock cycle time)……………………………………………………. (5)

Both versions of the Ephemeral State Processor (ESPR - ESPR.V1 and the to be presented ESPR.V2) are pipelined processors and their performance model can be derived from the basic pipelined performance equation as described in (5). Hereafter we refer to the performance model in terms of the ESPR architecture which refers to both versions. The ideal CPI for ESPR can be given as,

CPI_{ESPR} = No. of clock cycles for an instruction in unpipelined ESPR / No. of pipe stages……………………………………………………………………………(6)

All instructions except the IN and OUT instruction can be included in the above CPI equation. The IN instruction gets the input packet into ESPR in 32-bit blocks and it takes more than one pipelined ESPR clock cycle to complete it depending on the input packet size. The number of clock cycles for completion of the IN instruction can be determined from studying the macro system flow chart in Appendix B and it includes the basic count of clock cycles for the IN instruction and additional cycles to check whether the EOP is reached to stop getting input. Similarly, the OUT instruction also takes more than one pipelined ESPR clock cycle to output the packet. Depending on the packet size, IN and OUT takes an equal number of clock cycles to complete their respective operation. The number of pipe stages does not have effect in determining the CPI for IN and OUT. CPI for IN and OUT are given in equations (7) and (8) respectively.

CPI_{IN} = *No. of clock cycles for getting input packet using IN instruction……… (7)*

CPI_{OUT} = *No. of clock cycles for OUT instruction……………………………… (8)*

The performance of ESPR is determined based on the time it takes to complete the processing of a particular Ephemeral State Processing (ESP) packet after the ESPR is switched on. The CPU execution time for the packet is comprised of different execution times, which are described below, and it depends on type of input packet and type of resultant packet and also depends on whether the packet and Ephemeral State Store (ESS) checking failed or succeeded. The complete performance equation for ESPR can be derived from the following equations. The base CPU execution time is given as,

CPU_{BASE} =

*((avg. micro instruction count for a macro instruction * CPI_{ESPR}) + total number of stall cycles in a macro instruction) * (CLK_{ESPR})…………………………………………………………………………(9)*

Any ESP packet carries a macro opcode for performing a particular macro instruction and so the performance of ESPR is measured in terms of per packet processing and is determined by the execution time of a particular macro instruction. A macro instruction consists of a sequence of micro instructions in instruction memory, and the number of micro instructions executed for a particular macro instruction depends on the type of input packet and the availability of the required (tag, value) pair in ESS, as can be seen from the macro level system flow chart in Appendix B. So the term *avg. micro instruction count* is used in the above CPU_{BASE} equation and it excludes any IN and OUT micro instruction. Stalls in ESPR arise due to branch or jump instructions. CLK_{ESPR} is the basic clock cycle period for ESPR. Similarly the CPU execution times for the IN and OUT micro instructions are given in equations (10) and (11) as follows.

$$CPU_{IN} = CPI_{IN} * CLK_{ESPR}\ldots\ldots(10)$$

$$CPU_{OUT} = CPI_{OUT} * CLK_{ESPR}\ldots\ldots(11)$$

The instruction count for the IN or OUT micro instruction is 1, and there are no stall cycles during their execution. After the packet is received in ESPR, it is checked for any errors and conditions and then the ESS is checked for its availability before the packet gets processed. So the CPU execution time spent in checking the packet and ESS, is given by,

$$CPU_{CHECK} = (\textit{No. of clock cycles for packet and ESS checking}) * (CLK_{ESPR})\ldots(12)$$

$$CPU_{FA} = (\textit{No. of clock cycles for FWD or ABORT1 or ABORT2}) * (CLK_{ESPR})\ldots\ldots(13)$$

CPU_{FA} is the CPU execution time for either the FWD or ABORT1 or ABORT2 micro instruction. In case of failure of any checking as described above, eq. (13) helps in determining the total execution time. The CPU execution time of the resultant packet, depending on the FWD/DROP of packets or failure of checking, can be obtained by the combination of any of the above five different CPU execution time equations. CPU execution time for forwarded and dropped packets is given as,

$$CPU_{FWD\text{-}PACKET} = CPU_{BASE} + CPU_{IN} + CPU_{OUT} + CPU_{CHECK}\ldots\ldots(14)$$

$$CPU_{DROP\text{-}PACKET} = CPU_{BASE} + CPU_{IN} + CPU_{CHECK}\ldots\ldots(15)$$

If any packet or ESS checking fails, the CPU execution time in that case is given as,

$$CPU_{CHK\text{-}FAILED} = CPU_{IN} + CPU_{OUT} + CPU_{FA} + ((\textit{1, 2 or 3 clock cycles (depending on which checking fails)}) * CLK_{ESPR})\ldots\ldots(16)$$

Thus, the above performance model is designed to measure the time taken by ESPR to complete processing of an incoming packet, depending on the type of packet and the processed results of packet. This is a theoretical description of the performance of ESPR. Real performance results can be obtained from the post implementation simulation results to be described and seen in later chapters.

Chapter Six

Post-Synthesis and Post-Implementation Simulation Validation of ESPR.V1 Architecture

The hardware design of any system starts with the design specifications, design architecture and design description using Hardware Description Languages (HDL). The next step in the design cycle is simulation and design verification using CAD tools to synthesize and implement the hardware system to a desired programmable logic technology. Field Programmable Gate Array (FPGA) technology using reconfigurable logic is the widely used programmable logic as it offers many advantages of cost effectiveness, flexibility, ability to reconfigure easily, large number of gate counts in a single chip and ability for rapid prototyping and design iteration. This Chapter discusses the Post-Synthesis and Post-Implementation validation testing of the ESPR.V1 architecture synthesized to a Virtex FPGA chip.

6.1 Introduction

Post synthesis simulation and Post Implementation simulation are two major essential phases in the design process of a hardware system in terms of organization, architecture and design validation. Post synthesis simulation can functionally validate the design architecture for its implementation to a specific FPGA chip. This simulation offers output results without considering the specific gate and other logic resource delays while configuring the chip. The gate and all other logic resource delays in the chip are included by the CAD tool while performing Post Implementation Simulation.

Post Synthesis and Implementation simulation and design validation of the final ESPR architecture was done using Xilinx Foundation series 3.1i software. It provides logic simulators for post synthesis and post implementation testing. One can monitor all the inputs, outputs and intermediate signals using this logic simulator. Because of its inability to accept test vectors from a file, test vectors for validation are input manually and exhaustive testing was not performed. But in all cases the simulation results were compared with valid outputs for validation.

The CAD tool provided by Xilinx for Post Synthesis Simulation validation used the environment of a PC (Personal Computer) system – Pentium III 550 MHz Processor, with Windows 98 platform and 384 MB (Megabytes) of RAM memory. The design capture of ESPR.V1 was synthesized into a Xilinx Virtex 800 FPGA chip and implemented on a Virtex2 – 4000 FPGA chip. Post Implementation simulation verification with desired timing constraints on the design is performed on a PC (Personal Computer) system – Pentium III 550 MHz Processor, with Windows 2000 platform and 640 MB (Megabytes) of RAM memory. The logic resources utilized in the Xilinx Virtex2 – 4000 FPGA chip to implement the described ESPR.V1 architecture is given in the following Table 6.1.

Table 6.1 Logic Resources Utilization for ESPR.V1 Architecture

Resources	Utilization
4 Input LUTs	13, 840
Flip flops	7, 388
Block RAMs	16
Equivalent System Gates	1,386,196

6.2 Post-Synthesis Simulation Validation of ESPR.V1 Architecture

Post synthesis simulation validation provides for the functional validation of the ESPR.V1 architecture on a FPGA chip. All the components of ESPR.V1 are first developed, synthesized and verified separately for functional correctness. Then the whole system of ESPR.V1 is connected using individual modules, synthesized and tested for its functional validation. Most of the micro instructions are tested on a clock cycle by clock cycle basis for proper generation of internal and external signals, and outputs. All the individual components and the whole ESPR.V1 system are not tested exhaustively but are tested for a large set of varying input conditions.

The pipelined ESPR.V1 system is synthesized to run at a clock cycle (clk_pipe) of 10 nanoseconds (frequency of 100 MHz). The simulation validation was first done for individual micro instructions and then for small programs using the micro instructions. For this simulation the instruction memory is first written with specific micro instructions to be tested and then simulated for proper execution in corresponding clock cycles. Synthesis was not optimized to run at a maximum clock rate since achieving this is a long drawn out process. Our top priority was functional validation of the architecture. Also, the Xilinx Virtex FPGA chip used for implementation is an older FPGA chip with long propagation delays through its logic resources.

6.2.1. Simulation Validation of Single Micro Instructions

Most of the micro instructions were tested and verified individually and then programs with sequences of micro instructions were tested for correct execution. The first micro instruction to be verified was the ‘ADD’ (ALU/SHIFTER type instructions) instruction and the next instruction to be verified was the ‘GET’ (GET/PUT type instruction for ESS) instruction. Both of these micro instructions exercise most functional units in the processor.

6.2.1.1.Validation of ‘ADD’ Micro Instruction

To verify the ADD micro instruction, the bit pattern for this instruction was written to the instruction memory. This micro instruction utilizes PC, micro instruction memory, General Purpose Register File, Controller, ALU and Shifter functional units. The ADD instruction is interpreted as,

ADD RD, RS1, RS2

The ADD instruction verified here was,

ADD R5, R4, R3

– 001001 00101 00100 00011 00000 0 00000 0 000000 100 0000000000000000 000000

- 24A4180001000000 (Hex. Equivalent for binary representation)

The source registers R3 and R4 were initially loaded with values 6 and A respectively using the MOVI micro instruction. The verification of this instruction is shown in Figure 6.1. The verification starts with the IF stage and the instruction (instchk) is fetched during the first clock cycle.

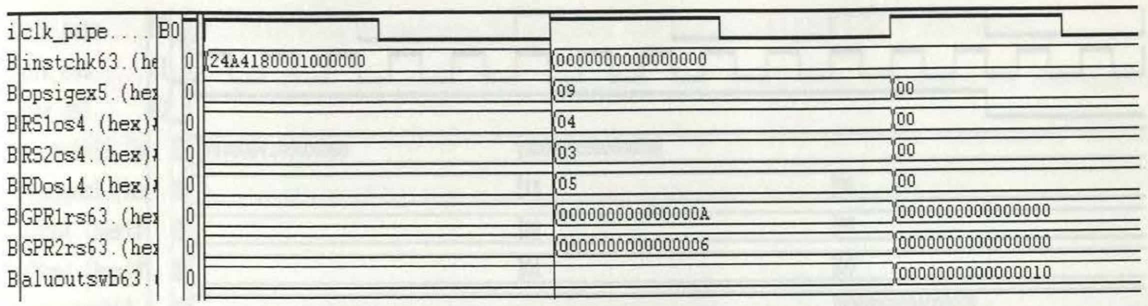


Figure 6.1. Validation of 'ADD' Micro Instruction

During the second clock cycle of ID stage, the opcode for ADD is decoded as '9' (opsigex) and the source and destination register are decoded as '4' (RS1os), '3' (RS2os) and '5' (Rdos1) respectively. In the same ID stage the source registers R4 (GPR1rs) and R3 (GPR2rs) are read as 'A' and '6' respectively. In the next clock cycle of EX stage, the ALU output (aluoutswb) for the addition of A + 6 = 16, is obtained in hex as "0000000000000010". This validates the ADD micro instruction.

6.2.1.2.Validation of 'GET' Micro Instruction

To validate the GET micro instruction, the bit pattern was written in the instruction memory. This micro instruction utilizes PC, micro instruction memory, Tag Register File, Controller, and ESS functional units. The GET instruction is interpreted as,

GET TR, VR

The GET instruction validated here was,

GET TR1, VR1

- 011110 00000 00000 00000 00001 0 00001 1 000000 000 0000000000000000 000000

- 7800004180000000 (Hex Eq.)

The source register TR1 was initially loaded with a Tag of 5 using the MOVI micro instruction. A Value of 2 was already associated with this Tag and stored in ESS using the PUT micro instruction. The GET instruction is executed after sometime to validate whether it checks the ESS for this Tag 5 and the lifetime of its Value. The validation of this instruction is shown in Figure 6.2.

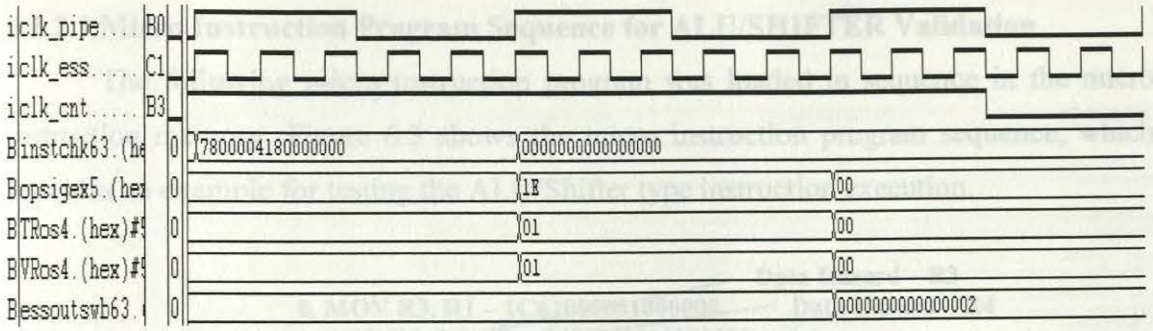


Figure 6.2. Validation of 'GET' Micro Instruction

The first clock cycle of the IF stage fetches the instruction (instchk) for GET. The ID stage decodes the opcode (opsigex) for GET and the source Tag Register (TRos) and the destination Value Register (VRos). ESS in the EX stage runs with the clock (clk_ess) of 2 nanoseconds and at a frequency five times the frequency of the processor. The clock (clk_cnt) for the counter in ESS which counts for the lifetime of the value runs at a very low frequency and has a period of 80 nanoseconds. The value is read out of the ESS from the EX stage as "0000000000000002". This proves a successful validation of the GET micro instruction.

6.2.2. Micro Instruction Program Sequence Validation of ESPR.V1 Architecture

After functional validation of the ESPR.V1 architecture executing single micro instructions, the pipelined ESPR.V1 architecture was validated by loading the micro instruction memory with sequences of micro instructions that form small programs. Three example programs are shown in this section with its simulation validation. These programs are written in a way to show each of the features and components of the pipelined processor at work. Most of all these programs have data hazards, which are automatically reduced/eliminated by the forwarding unit. One program with a Branch type instruction uses the hazard detection unit to control branch hazards as explained in the architectural description of ESPR.V1. This section briefly explains pipelined execution of the micro instruction sequences and shows the simulation validation results. The first program is explained in a detailed manner and the remaining two programs are briefly explained.

6.2.2.1.Micro Instruction Program Sequence for ALU/SHIFTER Validation

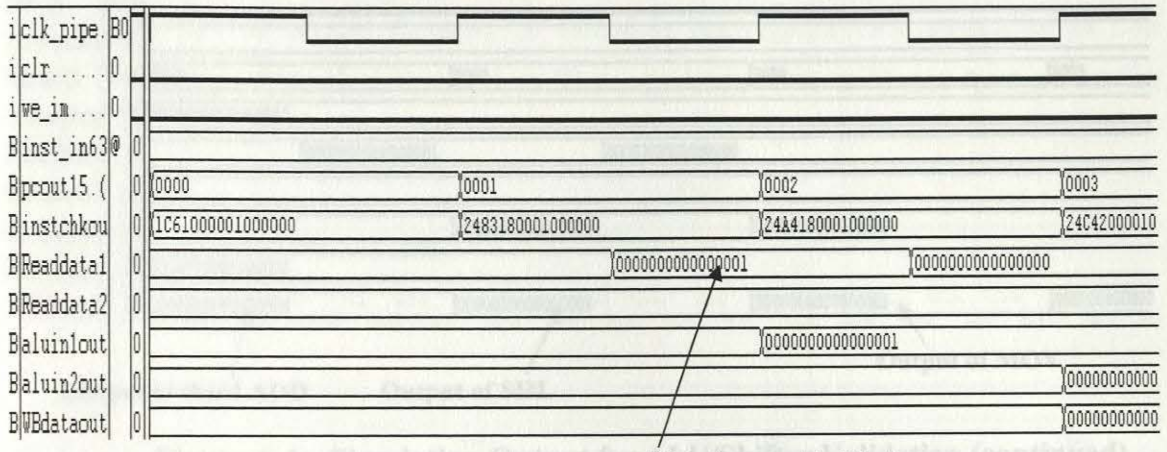
The following micro instruction program was loaded in sequence in the micro instruction memory. Figure 6.3 shows the micro instruction program sequence, which provides an example for testing the ALU/Shifter type instruction execution.

```
                                              Data Hazard – R3
0. MOV R3, R1 – 1C61000001000000          Data Hazard – R4
1. ADD R4, R3, R3 – 2483180001000000
2. ADD R5, R4, R3 – 24A4180001000000
3. ADD R6, R4, R4 – 24C4200001000000
4. SHL R7, R4, 2 – 44E4000001000001
5. MOV R8, R3 – 1D03000001000000
```

Figure 6.3. Program for Validating ALU/Shifter

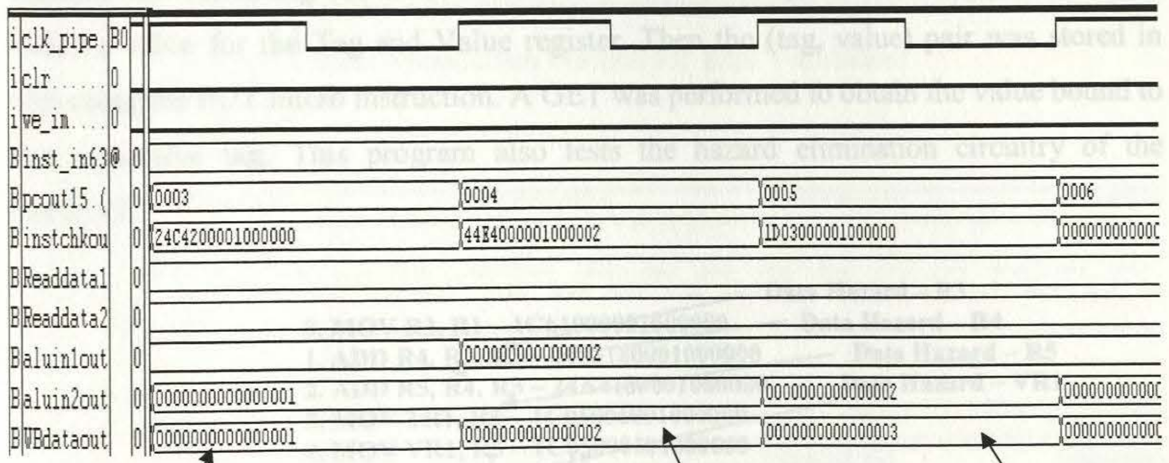
This program also tests the forwarding unit to eliminate the data hazards that arise in the ADD instructions. In the first two ADD instructions there is a danger of the old data being read instead of the new value for R3 from the MOV instruction and the new value for R4 from the first ADD instruction. This is a called a Data Hazard. This is because the new values are not written until the WB stage but the next sequencing instruction in the ID stage wants to read the new value before it is written. The forwarding unit allows for the proper execution of the instruction without any hazard by forwarding the required data to the computational unit inputs from the ALU output. Figures 6.4a, 6.4b and 6.4c show the post synthesis simulation validation results for the above program.

All the instructions take a single clock cycle to execute. Figure 6.4a shows the IF stage for the first three instructions, ID stage for the first two instructions and EX stage for the first instruction. Figure 6.4b shows the fetching of the remaining three micro instructions, WB stage for first three instructions, EX stage for second, third and fourth instructions and ID stage for third, fourth and fifth instruction. Figure 6.4c shows the WB stage for the fourth, fifth and sixth instruction and the remaining stages for the remaining instructions. All instructions were executed correctly and their execution also validated the functional operation of the forwarding units.



R1 being read in ID stage

Figure 6.4a. Simulation Output for ALU/Shifter Validation



Output of first MOV instruction in 4th clock cycle

Output of first ADD instruction in 5th clock cycle

Output of the second ADD instruction in 6th clock cycle

Figure 6.4b. Simulation Output for ALU/Shifter Validation (continued)

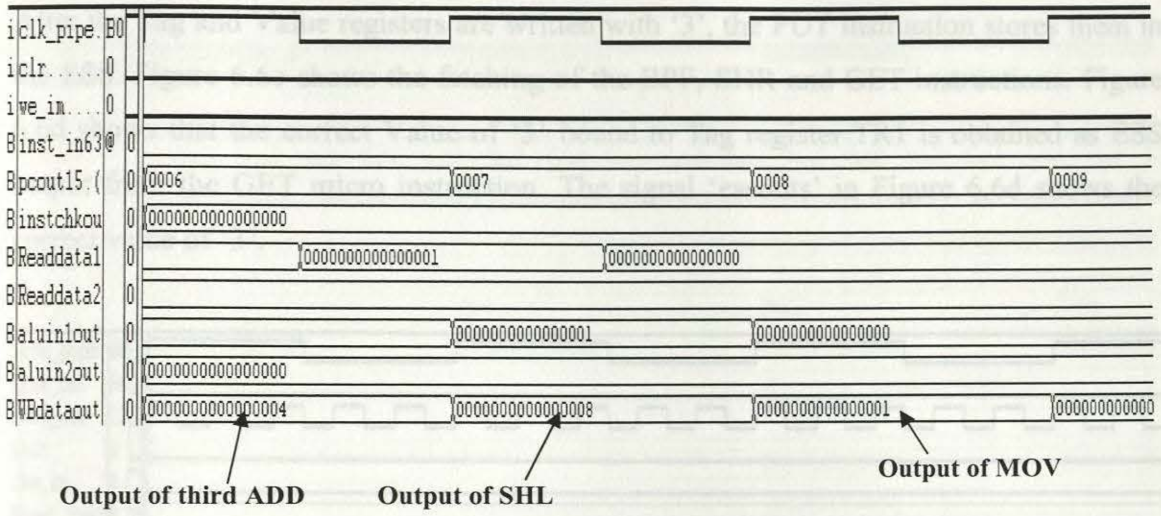
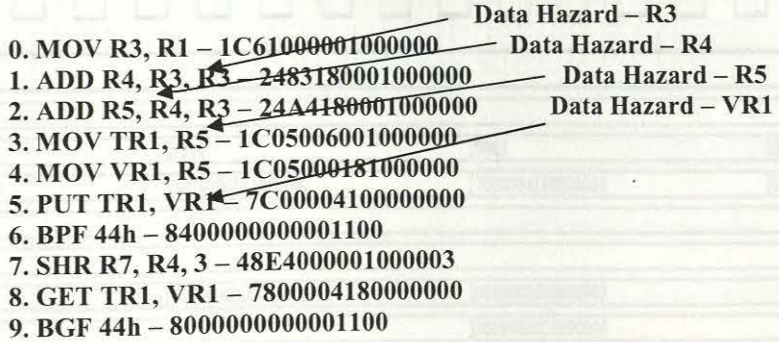


Figure 6.4c. Simulation Output for ALU/Shifter Validation (continued)

6.2.2.2.Micro Instruction Program Sequence for ESS Validation

Figure 6.5 shows the micro instruction program sequence, which provides an example for validating the ESS. First a series of ALU operations were performed to obtain a value for the Tag and Value register. Then the (tag, value) pair was stored in ESS using the PUT micro instruction. A GET was performed to obtain the value bound to that respective tag. This program also tests the hazard elimination circuitry of the ESPR.V1.



0. MOV R3, R1 – 1C61000001000000
1. ADD R4, R3, R3 – 2483180001000000
2. ADD R5, R4, R3 – 24A4180001000000
3. MOV TR1, R5 – 1C05006001000000
4. MOV VR1, R5 – 1C05000181000000
5. PUT TR1, VR1 – 7C00004100000000
6. BPF 44h – 8400000000001100
7. SHR R7, R4, 3 – 48E4000001000003
8. GET TR1, VR1 – 7800004180000000
9. BGF 44h – 8000000000001100

Figure 6.5. Program for Validating ESS

Figures 6.6a, 6.6b, 6.6c and 6.6d show the post synthesis simulation validation results for the above program. Figure 6.6a shows a series of ALU operations similar to the previous program. Figure 6.6b shows the operation and result of the ALU operations and fetching of the PUT instruction. It also shows a value of ‘3’ being written to the Tag register.

After the Tag and Value registers are written with '3', the PUT instruction stores them in the ESS. Figure 6.6c shows the fetching of the BPF, SHR and GET instructions. Figure 6.6d shows that the correct Value of '3' bound to Tag register TR1 is obtained as ESS output from the GET micro instruction. The signal 'essouts' in Figure 6.6d shows the correct value of '3'.

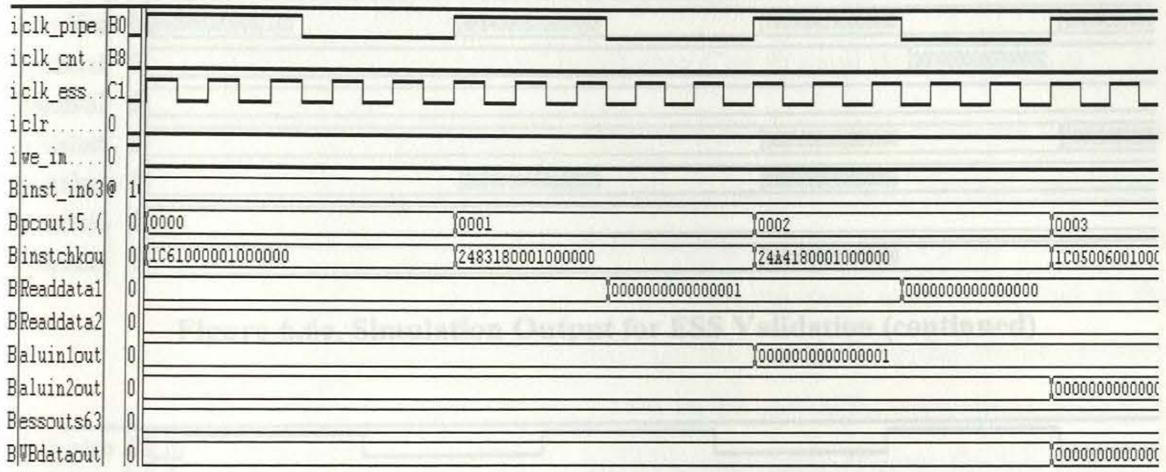
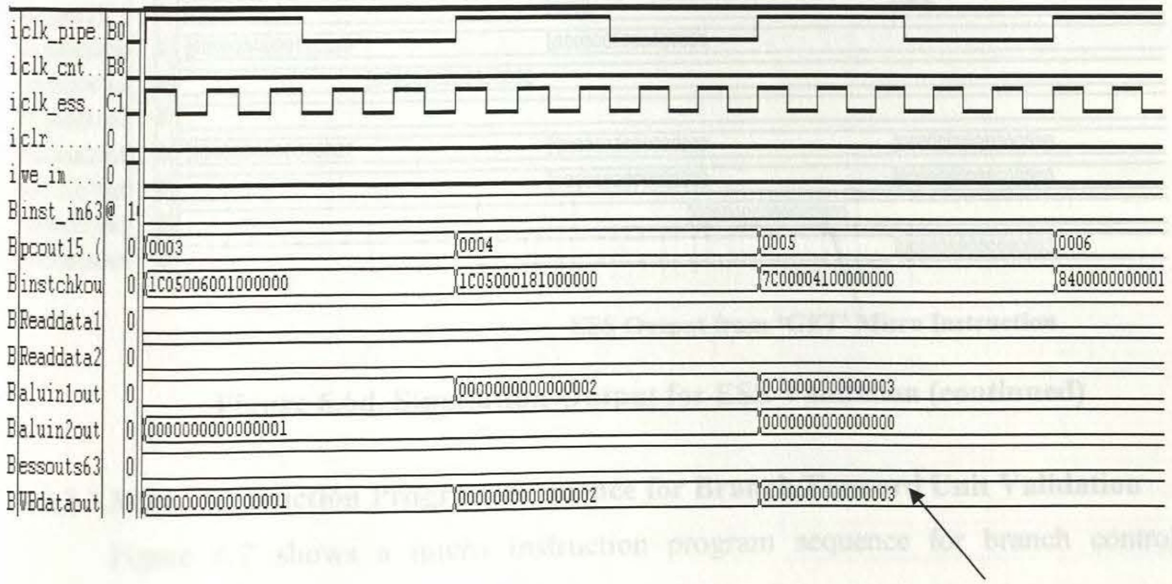


Figure 6.6a. Simulation Output for ESS Validation



Output Value '3' given to Tag Register TR1

Figure 6.6b. Simulation Output for ESS Validation (continued)

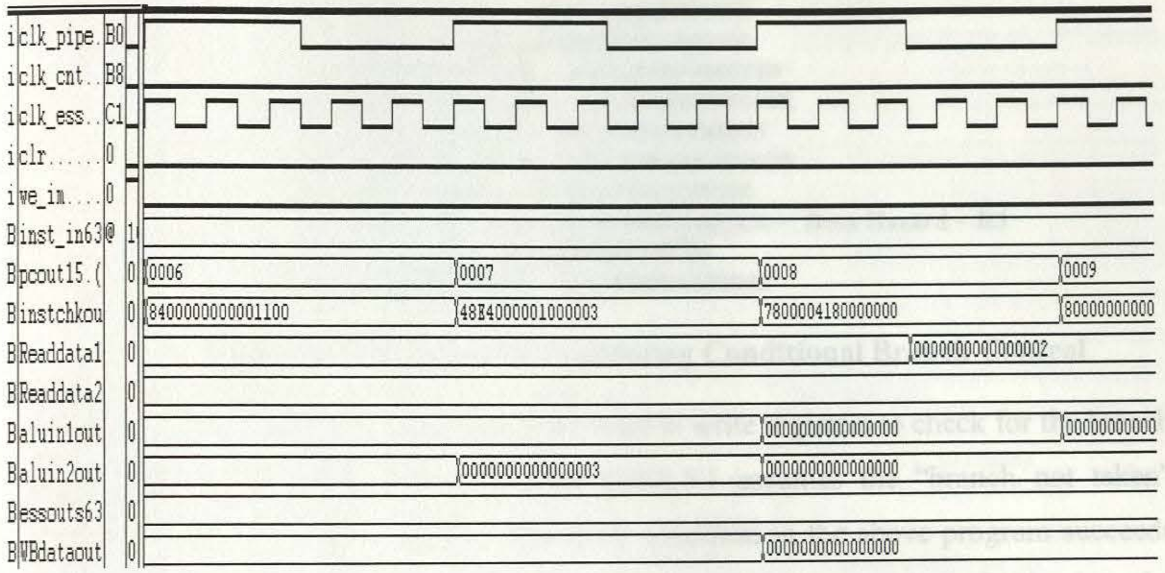
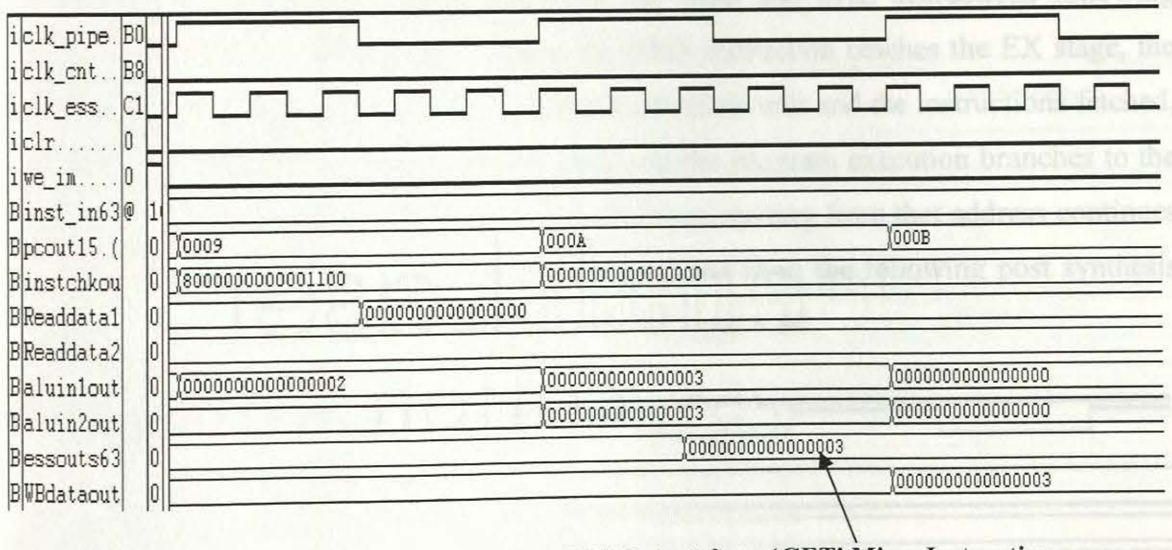


Figure 6.6c. Simulation Output for ESS Validation (continued)



ESS Output from 'GET' Micro Instruction

Figure 6.6d. Simulation Output for ESS Validation (continued)

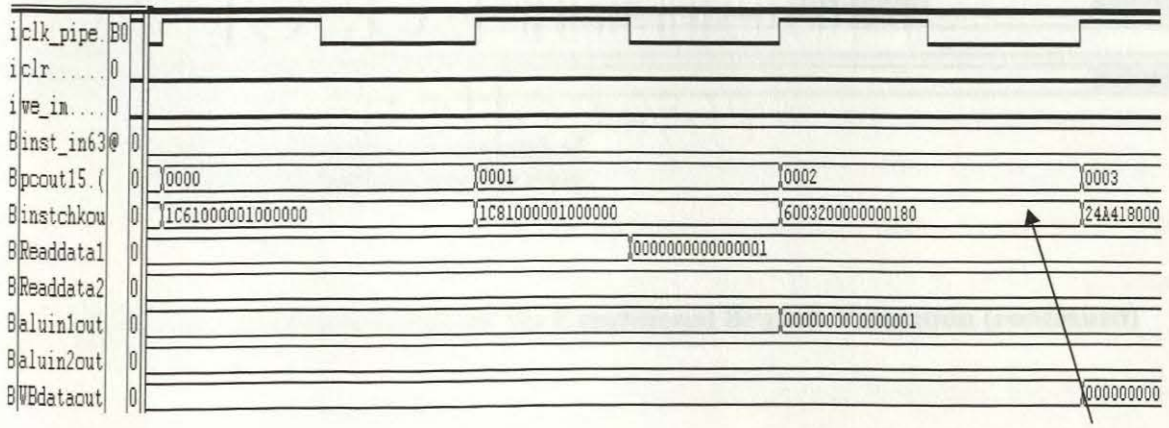
6.2.2.3.Micro Instruction Program Sequence for Branch/Forward Unit Validation

Figure 6.7 shows a micro instruction program sequence for branch control validation.

```
                                        Branch Hazard and Data Hazard – R4
0.  MOV R3, R1 – 1C61000001000000
1.  MOV R4, R1 – 1C81000001000000
2.  BEQ R3, R4, 6h – 6003200000000180
3.  ADD R5, R3, R4 – 24A4180001000000
4.  SHL R7, R4, 5 – 44E4000001000003
5.  AND R6, R5, R7 – 38C5380001000000
6.  MOV R8, R6 – 1D06000001000000
7.  MOVI R7, Fh – 20E00000010003C0 — Data Hazard – R3
8.  INCR R3 – 2C63000001000000
9.  ROR R9, R3, 2 – 5123000001000002
```

Figure 6.7. Program for Validating Conditional Branch Control

The first two MOV micro instructions were used to write registers to check for the branch equality (BEQ) condition. The pipelined ESPR.V1 assumes the "branch not taken" condition to reduce branch hazards. The BEQ condition in the above program succeeds because the two registers R3 and R4 are written with the same value of '1'and so the program execution has to branch to the address (6h) specified by the branch address. Because of the branch not taken condition, the ADD and SHL instructions following BEQ will be fetched and decoded. When the BEQ instruction reaches the EX stage, the correct branch decision is taken by the branch detection unit and the instructions fetched, decoded or executed after BEQ were flushed and the program execution branches to the address specified by BEQ and the micro instructions starting from that address continues to execute in normal sequence. This can be validated from the following post synthesis simulation results shown in Figures 6.8a, 6.8b, 6.8c and 6.8d.



Fetching 'BEQ' Micro Instruction

Figure 6.8a. Simulation Output for Conditional Branch Validation

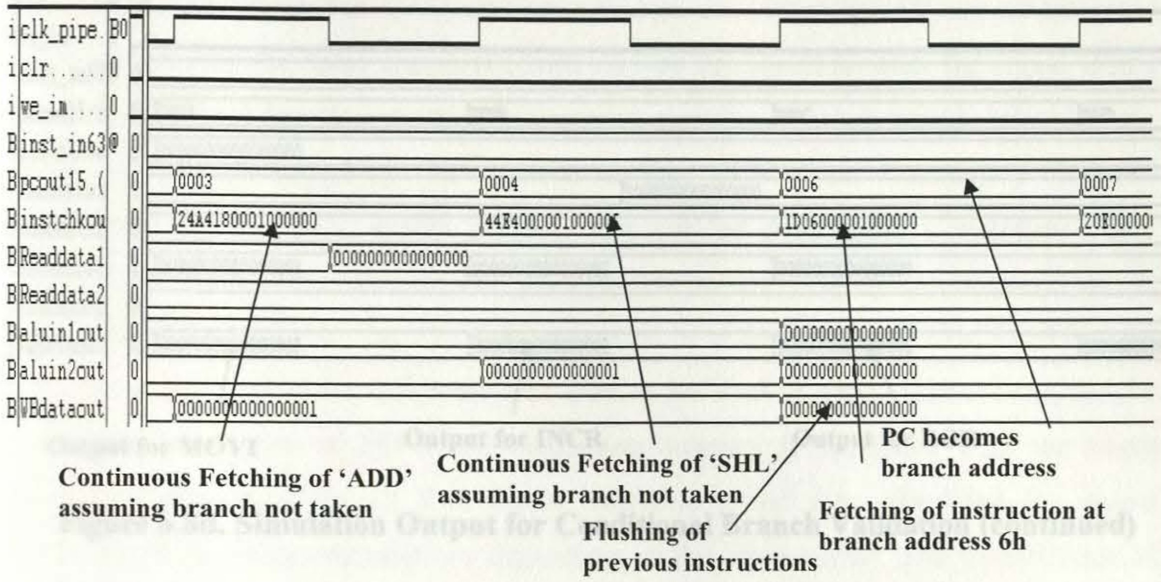


Figure 6.8b. Simulation Output for Conditional Branch Validation (continued)

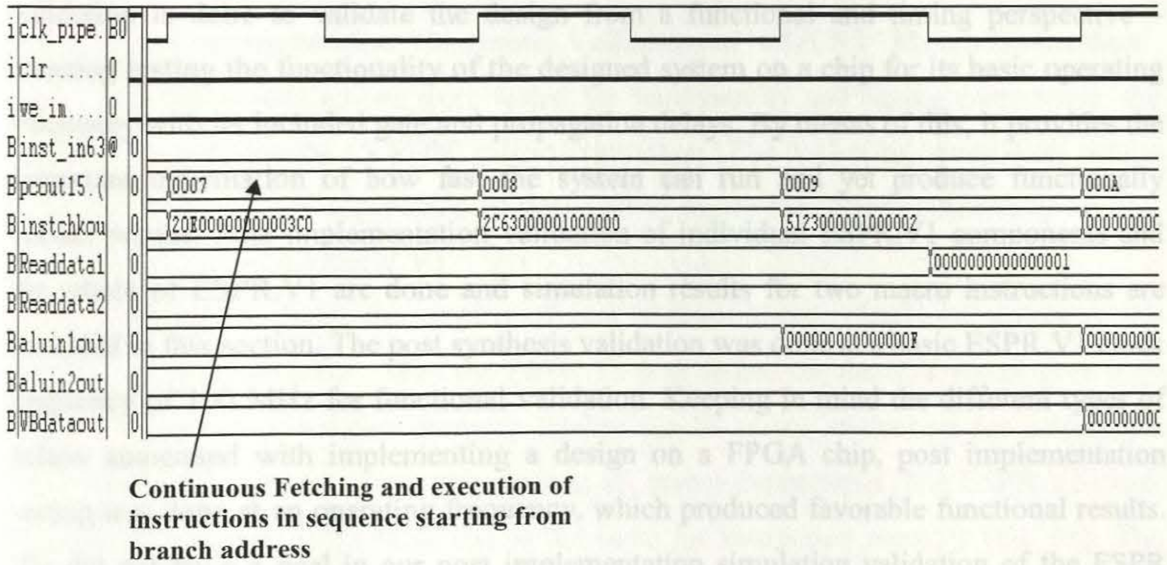


Figure 6.8c. Simulation Output for Conditional Branch Validation (continued)

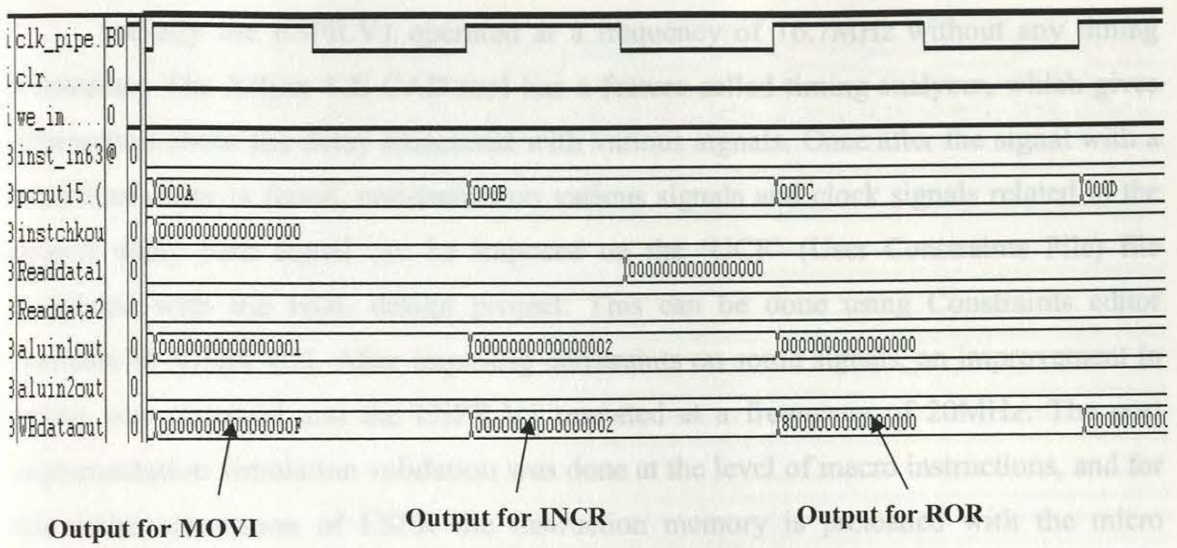


Figure 6.8d. Simulation Output for Conditional Branch Validation (continued)

6.3 Post-Implementation Simulation Validation of ESPR.V1 Architecture

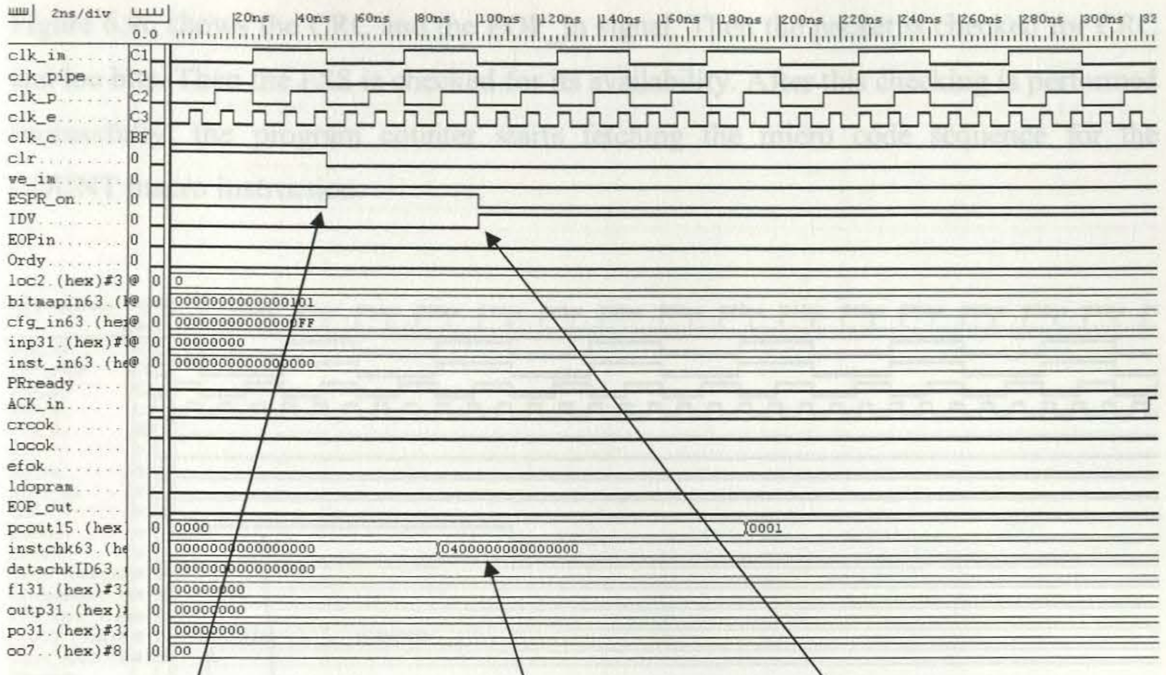
Unlike Post synthesis validation of any HDL system design, Post implementation validation is done to validate the design from a functional and timing perspective – meaning testing the functionality of the designed system on a chip for its basic operating frequency with its included gate and propagation delays. By means of this, it provides the important information of how fast the system can run and yet produce functionally correct results. Post implementation validation of individual ESPR.V1 components and the whole of ESPR.V1 are done and simulation results for two macro instructions are provided in this section. The post synthesis validation was done at a basic ESPR.V1 clock frequency of 100 MHz for functional validation. Keeping in mind the different types of delays associated with implementing a design on a FPGA chip, post implementation testing was done at an operating frequency, which produced favorable functional results. We did not have a goal in our post implementation simulation validation of the ESPR architecture to maximize system clock frequency. How to do that is known but time consuming. Our goal was simply to determine the frequency at which the ESPR would perform functionally correct. This frequency becomes the base frequency which can be improved upon via synthesis and VHDL coding optimizations in addition to deeper pipelining of the ESPR.V1 architecture.

Initially the ESPR.V1 operated at a frequency of 16.7MHz without any timing constraints. The Xilinx 4.2i CAD tool has a feature called timing analyzer, which gives information about the delay associated with various signals. Once after the signal with a maximum delay is found, constraints on various signals and clock signals related to the longest delay path signal can be imposed on the 'UCF' (User Constraints File) file associated with the HDL design project. This can be done using Constraints editor available in Xilinx 4.2i. After imposing constraints on some signals, an improvement in timing was obtained and the ESPR.V1 operated at a frequency of 20MHz. The post implementation simulation validation was done at the level of macro instructions, and for this entire simulation of ESPR the instruction memory is preloaded with the micro instruction sequences for all five macro instructions and then simulated for proper execution of the macro instructions depending on the input packet. The initialization of the instruction memory was done by writing a constraints file (NCF) in the desired format and saving this file under the specific HDL design project in Xilinx 4.2i.

6.3.1. Post-Implementation Simulation Validation of 'COUNT' Macro Instruction

All macro instructions were tested for functionality and timing correctness; the first one tested was the COUNT macro instruction. The following simulation results show the execution and completion of the COUNT macro instruction initiated by an ESP (COUNT) packet in the ESPR.V1.

Figure 6.9a shows the starting of post implementation simulation output of the COUNT macro instruction. After the ESPR.V1 is switched on, the IDV signal goes high and the preloaded instruction memory fetches the IN micro instruction for getting the input packet. This initial step is the same in all macro instructions. The ESPR.V1 clock (clk_pipe) frequency is 20 MHz and is the same for instruction memory (clk_im). The clock (clk_p) for the packet processing modules is 2 times faster than the clk_pipe. Clock (clk_e) for ESS is five times faster than the pipeline clock. All this can be seen from Figure 6.9a. After the IN instruction is fetched, the ESPR.V1 starts getting the input packet in 32-bit blocks as shown in Figure 6.9b. The input packet for COUNT has the following format – 00070004, 00000001, 00000000, 00000001, 00000000, 00000002, 00000000, CRC.

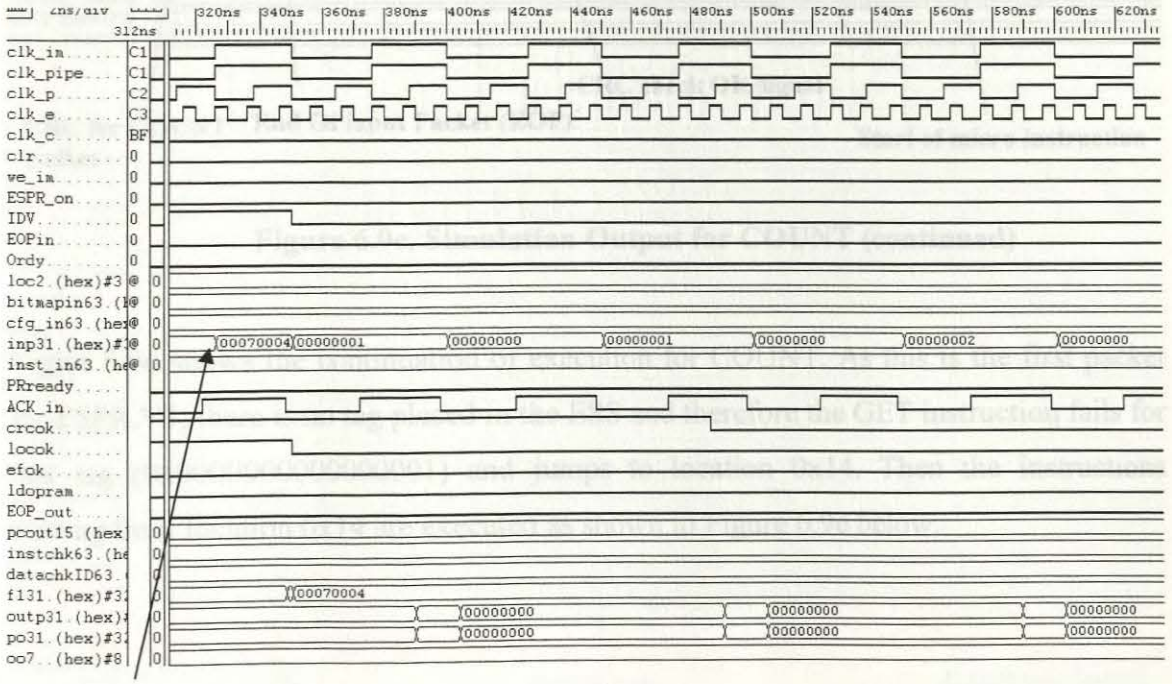


ESPR_ON is high indicating ESPR is switched on

Opcode for IN instruction is fetched

IDV signal goes high

Figure 6.9a. Simulation Output for COUNT



Start of input COUNT

Figure 6.9b. Simulation Output for COUNT (continued)

Figure 6.9c shows the CRC and the EOP_in signal. Then the packet is checked for CRC and loc bits. Then the ESS is checked for its availability. After this checking is performed successfully, the program counter starts fetching the micro code sequence for the COUNT macro instruction.

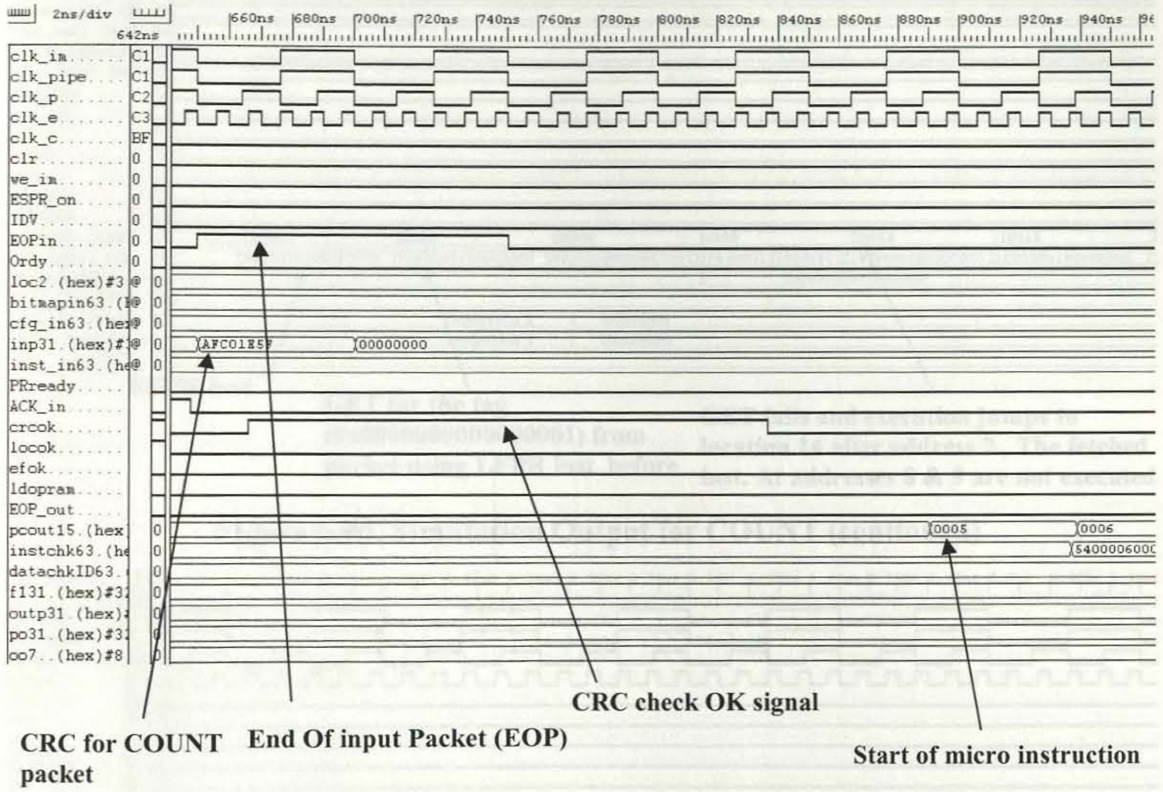
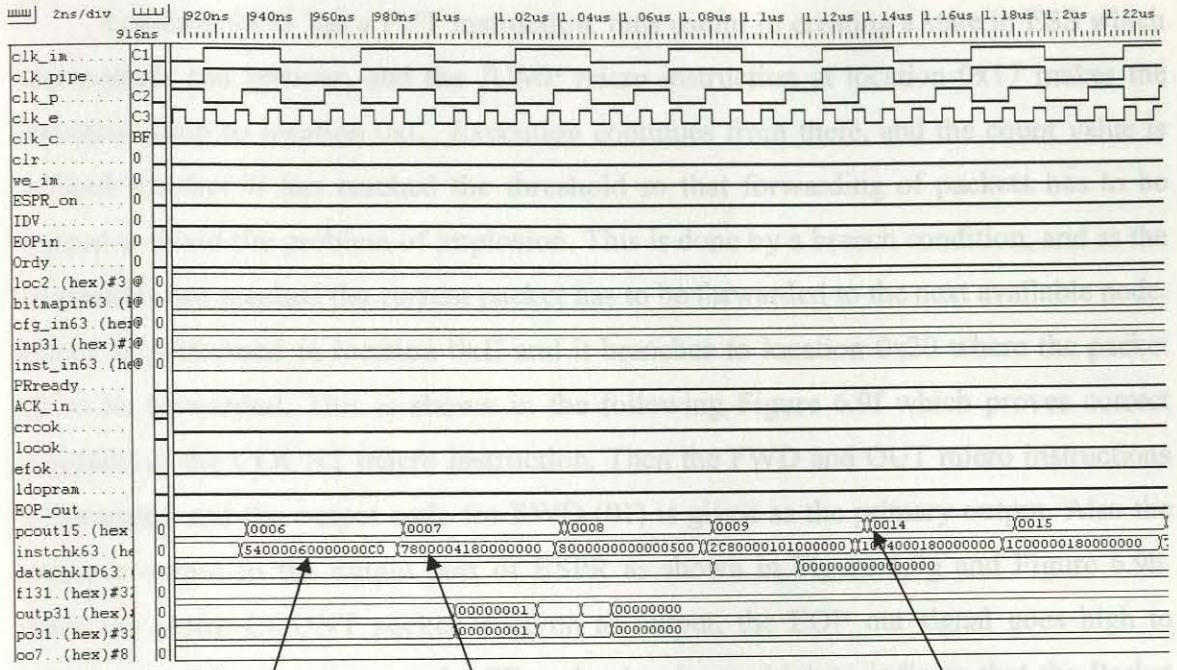


Figure 6.9c. Simulation Output for COUNT (continued)

Figure 6.9d shows the continuation of execution for COUNT. As this is the first packet for ESPR.V1, there is no tag placed in the ESS and therefore the GET instruction fails for that tag (0x0000000000000001) and jumps to location 0x14. Then the instructions starting from location 0x14 are executed as shown in Figure 6.9e below.

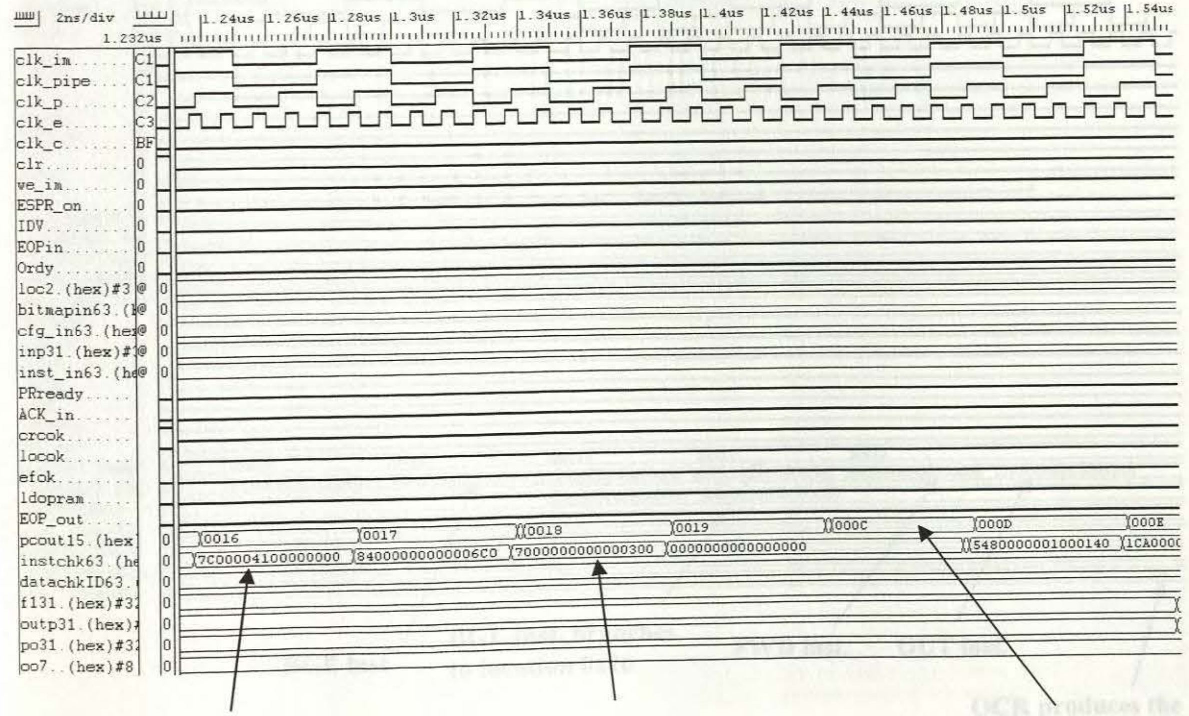


LFPR inst

GET for the tag (0x0000000000000001) from packet using LFPR inst. before

GET fails and execution jumps to location 14 after address 7., The fetched inst. At addresses 8 & 9 are not executed

Figure 6.9d. Simulation Output for COUNT (continued)



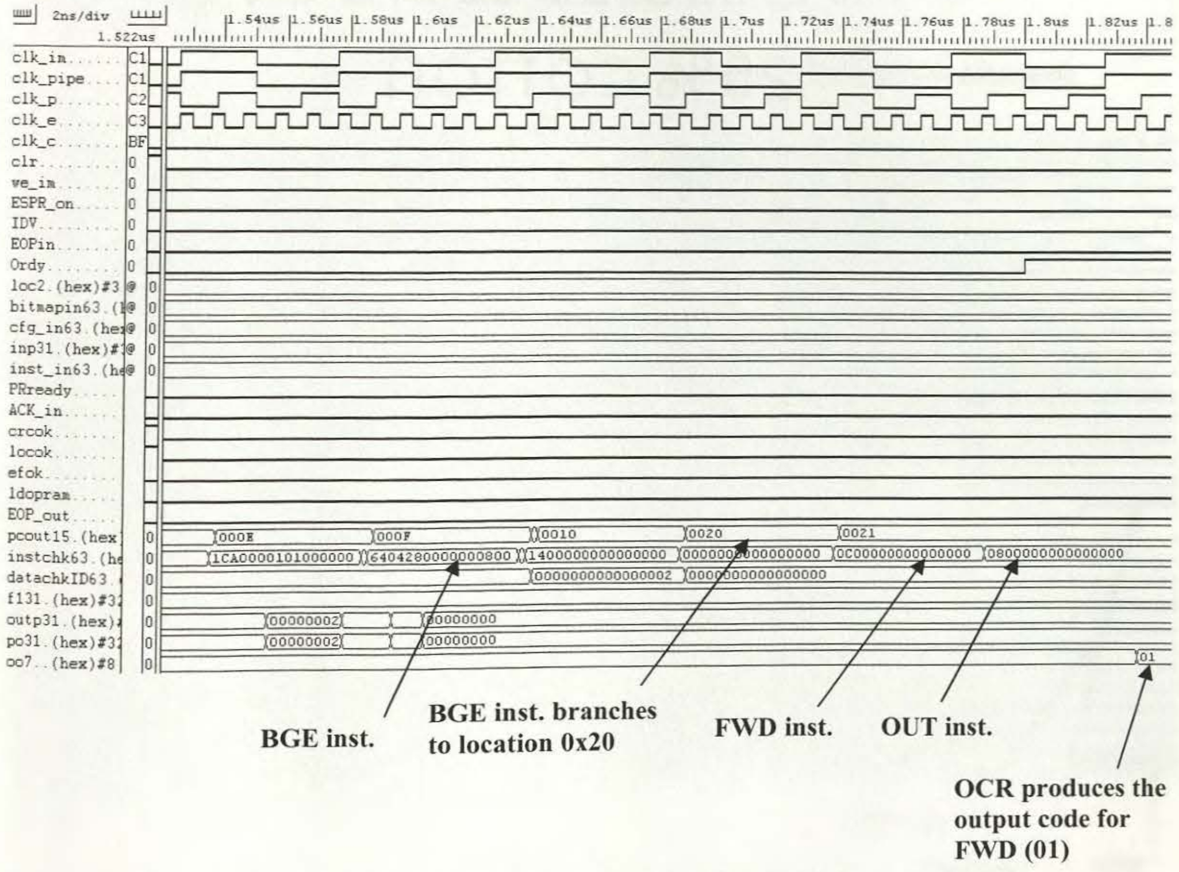
PUT inst. for creating an ESS state

JUMP inst.

JUMP inst. jumps to location 0xC

Figure 6.9e. Simulation Output for COUNT (continued)

Location 0x15 has a PUT instruction, thus useful in creating a state in ESS which later packets can retrieve, and the JUMP micro instruction at location 0x17 makes the execution jump to location 0xC. Execution continues from there, and the count value is checked whether it has reached the threshold so that forwarding of packets has to be stopped to avoid the problem of implosion. This is done by a branch condition, and as the threshold is not reached the current packet has to be forwarded to the next available node. Branch is performed in location 0xE and it branches to location 0x20 where the packet has to be forwarded. This is shown in the following Figure 6.9f which proves correct execution of the COUNT macro instruction. Then the FWD and OUT micro instructions are executed and the output code for FWD (01) is given as the primary output. Also the packet is output to the output port of ESPR as shown in Figure 6.9g and Figure 6.9h. When the entire COUNT packet is given as output, the EOP_out signal goes high to indicate the end of packet and the PRready signal goes high to indicate that the Packet RAM (PR) is ready to accept the next packets as can be shown in Figure 6.9h.



BGE inst.

BGE inst. branches to location 0x20

FWD inst.

OUT inst.

OCR produces the output code for FWD (01)

Figure 6.9f. Simulation Output for COUNT (continued)

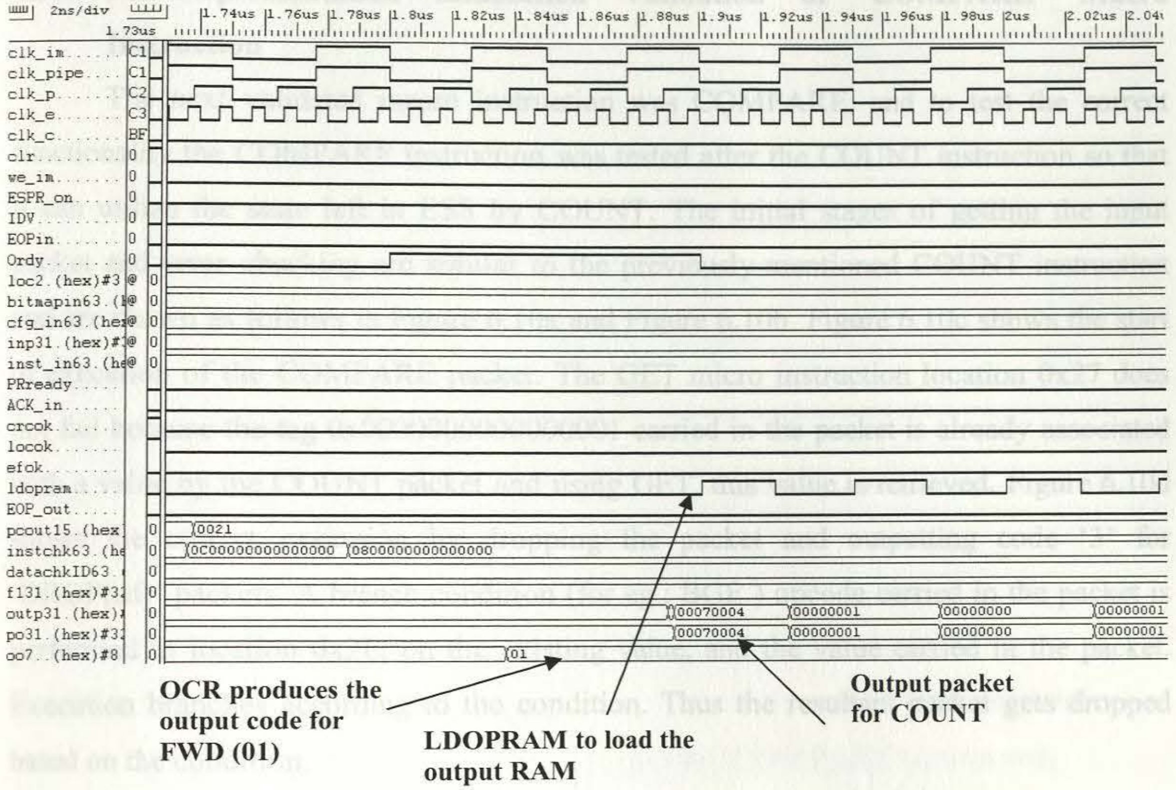


Figure 6.9g. Simulation Output for COUNT (continued)

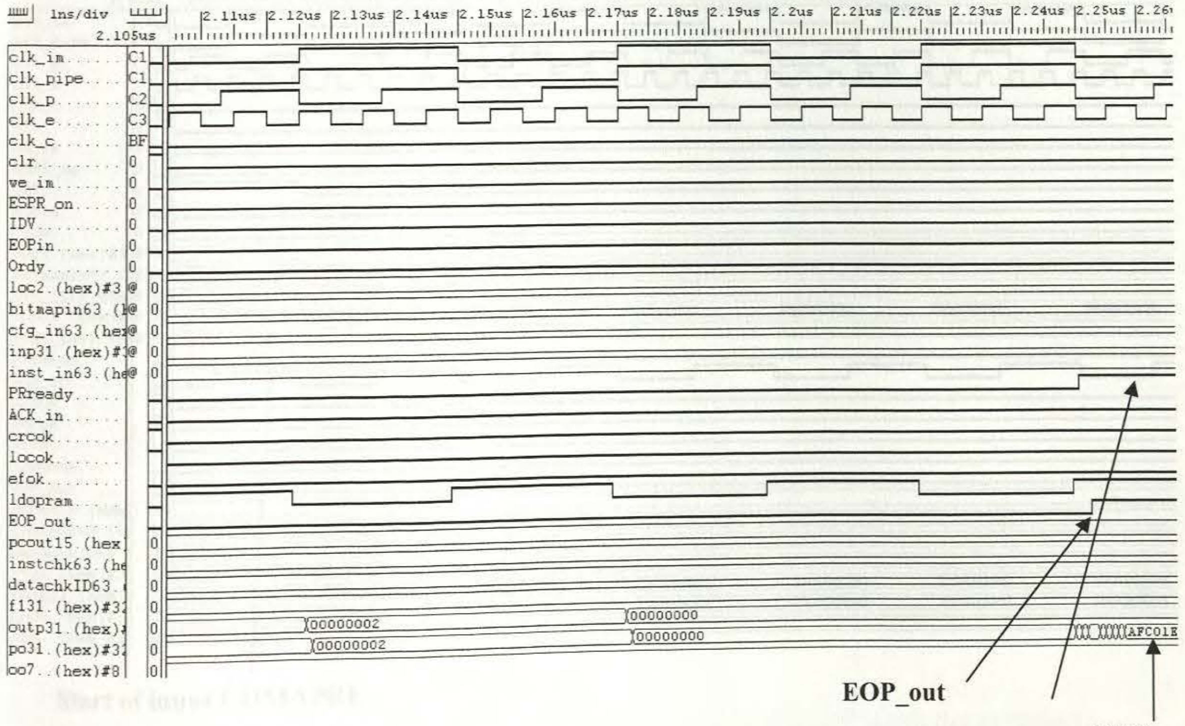
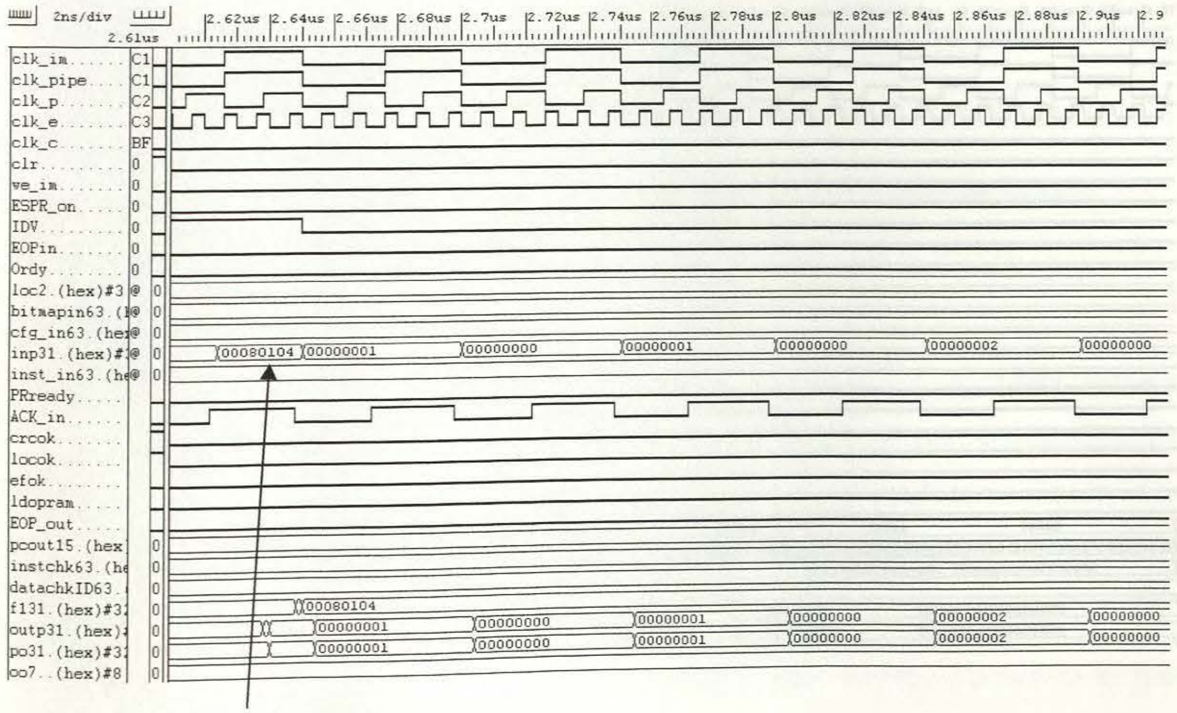


Figure 6.9h. Simulation Output for COUNT (continued)

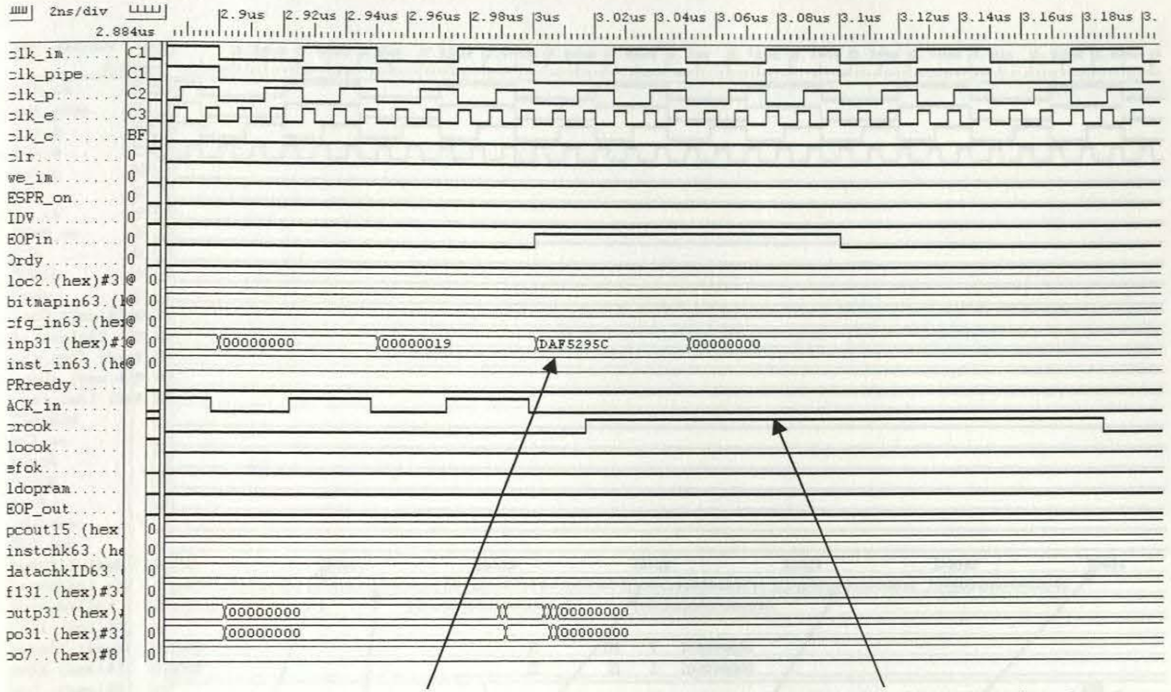
6.3.2. Post-Implementation Simulation Validation of 'COMPARE' Macro Instruction

The next validated macro instruction was COMPARE, and to test the correct functionality the COMPARE instruction was tested after the COUNT instruction so that it can utilize the state left in ESS by COUNT. The initial stages of getting the input packet and error checking are similar to the previously-mentioned COUNT instruction and are shown as follows in Figure 6.10a and Figure 6.10b. Figure 6.10c shows the start of execution of the COMPARE packet. The GET micro instruction location 0x27 does not fail because the tag 0x0000000000000001 carried in the packet is already associated with a value by the COUNT packet and using GET, this value is retrieved. Figure 6.10d shows the correct execution by dropping the packet and outputting code '3' for 'DROP(ed)' packets. A branch condition (for eg.: BGE) opcode carried in the packet is performed in location 0x2E, on the existing value, and the value carried in the packet. Execution branches according to the condition. Thus the resultant packet gets dropped based on the condition.



Start of input COMAPRE

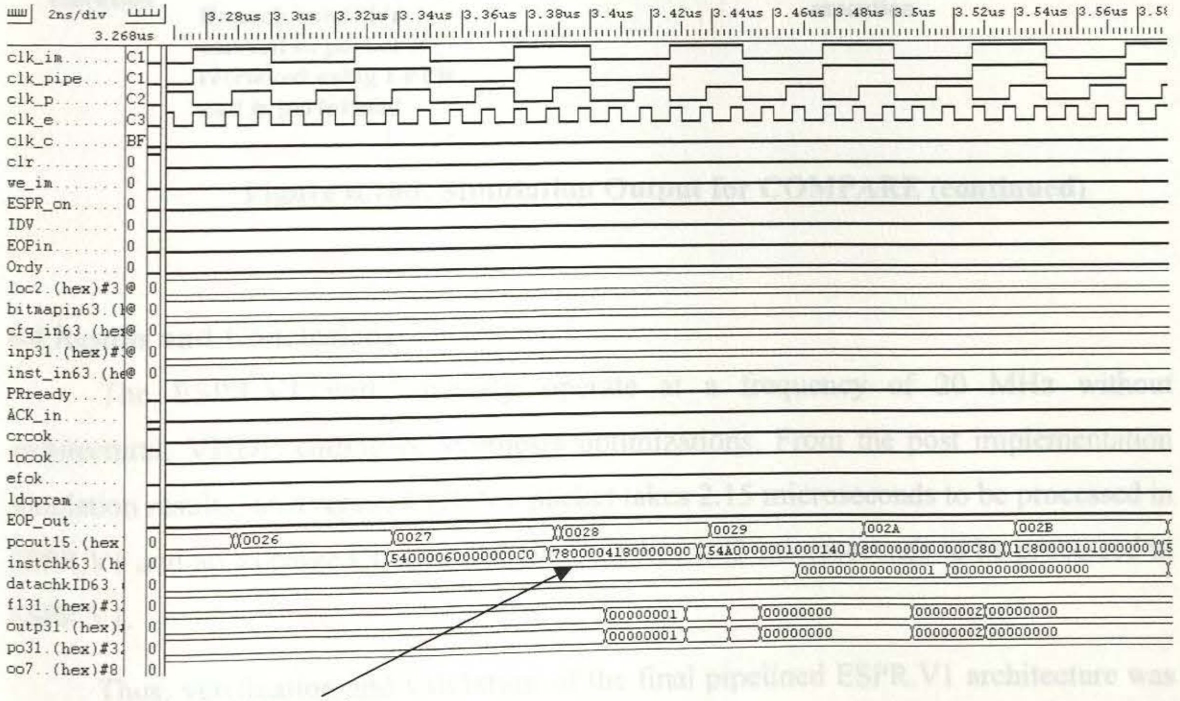
Figure 6.10a. Simulation Output for COMPARE



CRC for COMPARE packet

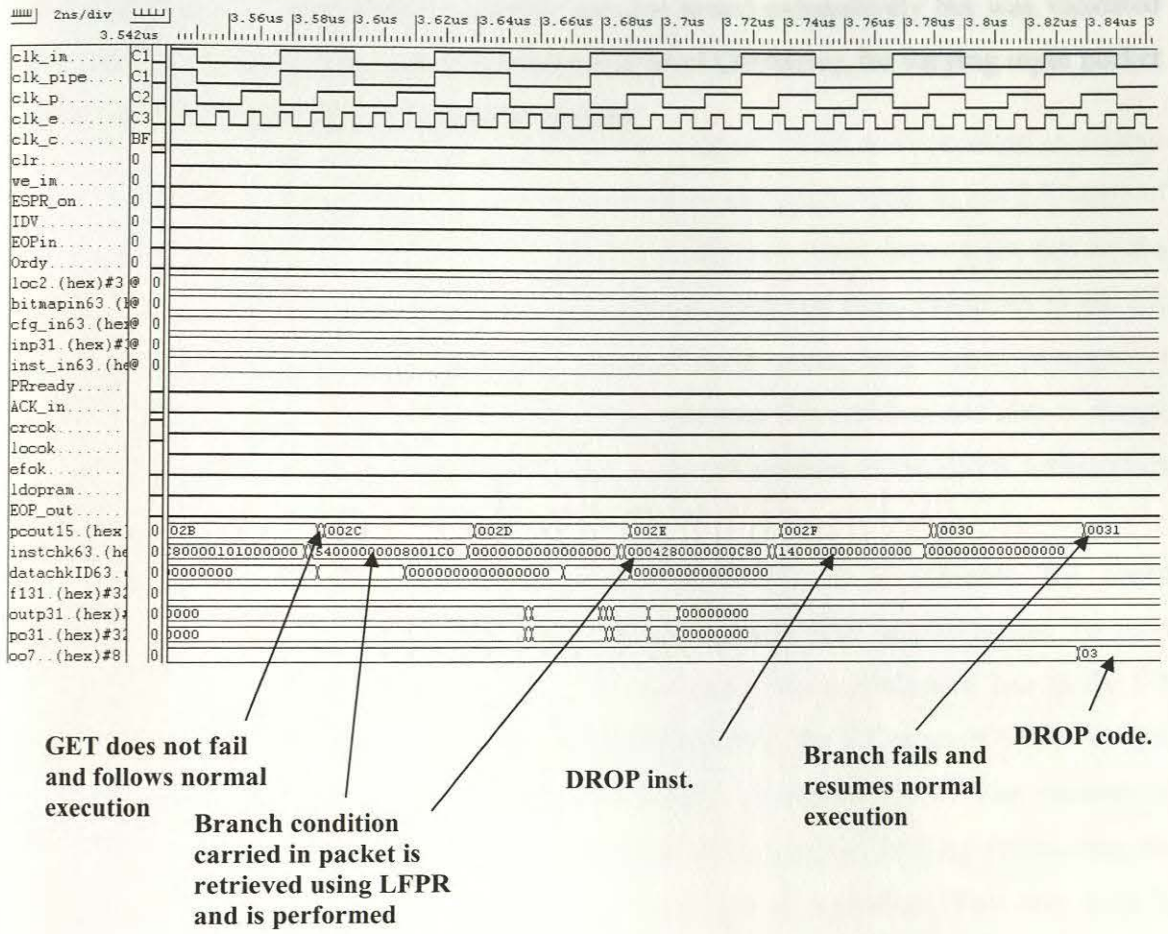
CRC check OK signal

Figure 6.10b. Simulation Output for COMPARE (continued)



GET microinstruction

Figure 6.10c. Simulation Output for COMPARE (continued)



GET does not fail and follows normal execution

Branch condition carried in packet is retrieved using LFPR and is performed

DROP inst.

Branch fails and resumes normal execution

DROP code.

Figure 6.10d. Simulation Output for COMPARE (continued)

6.4 Results and Conclusions

The ESPR.V1 will correctly operate at a frequency of 20 MHz without architectural, VHDL coding or synthesis optimizations. From the post implementation simulation results, an average COUNT packet takes 2.15 microseconds to be processed in ESPR.V1 and an average COMPARE packet takes 1.43 microseconds to be processed in ESPR.V1.

Thus, verification and validation of the final pipelined ESPR.V1 architecture was achieved by testing ESPR.V1 with example packets and testing of macro instructions using post synthesis and post implementation simulation verification/validation techniques. Performance results have been calculated for the COUNT and COMPARE macro instructions. The ESPR.V1 system was not tested exhaustively but was validated for correct functionality for a given performance level (20 MHz), for varying input packet formats with different macro instruction opcode.

Chapter Seven

Ephemeral State Processor Version 2 (ESPR.V2) Architecture

The ESPR.V1 architecture described in Chapter 5 is a Four-Stage pipelined architecture with the ESS being staged with all other execution units in the EX stage of the pipeline. The ESS operated at a clock frequency five times faster than that of the pipeline clock. Because of the number of functional units and their structures in the EX stage and because of the complexity of some of these units, long signal propagation delays (latency) can occur within this stage. To overcome this problem and also to design an overall performance enhanced architecture, a second version of the ESPR architecture is designed.

The main objective in the design of ESPR.V2 is to increase the speed (performance) by which the processor can operate to meet ESP service needs. To meet this objective, study and analysis of ESPR.V1 revealed that a bottleneck lies in the EX stage of ESPR.V1 as anticipated. To reduce the bottleneck, the EX stage of ESPR.V1 was partitioned to multiple stages resulting in a deeper pipelined ESPR. The essence of improving the performance of ESPR.V1 is to hide the latency of ESS by partitioning the ESS such that it can be implemented over three stages of a pipeline. This was done in addition to other architectural adjustments resulting in a Five-Stage pipelined ESPR.V2.

The performance enhancing pipelined design of Ephemeral State Store (ESS) and hence the five stage pipelined architectural design of ESPR – ESPR.V2, is discussed in the following chapter.

7.1. Pipelined ESS

In order to partition the work being done in the EX stage of ESPR.V1, the ESS, now fully in the EX stage, was transformed into a pipelined version. The flow of operations to be performed in the ESS for the 'GET' and 'PUT' instructions can be seen from Figure 4.5 of Chapter 4. To hide the latency and to achieve the functionality needed for 'GET' and 'PUT', the operational work of ESS is distributed into three pipeline stages. The functional block diagram of ESS can be seen in Figure 7.1 and the three-stage pipelined version of ESS can be seen from Figure 7.2. Figures 7.1 and 7.2 show only a high-level view of ESS and its pipelined version with its primary inputs and outputs. Detailed description of each stage of the pipelined ESS with supplementing diagrams and additional control signals is shown in the following sub section with Figures 7.3, 7.4 and 7.5.

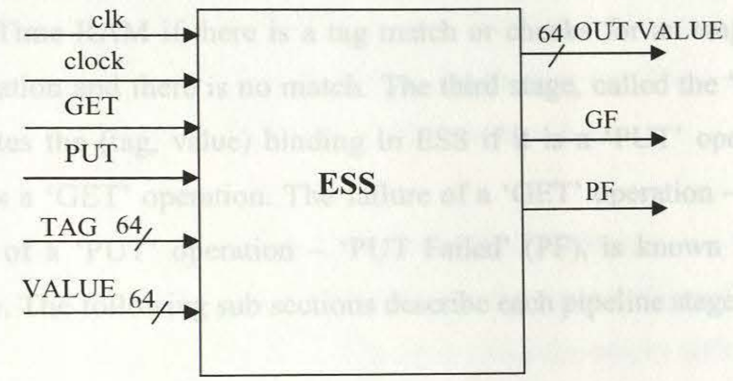
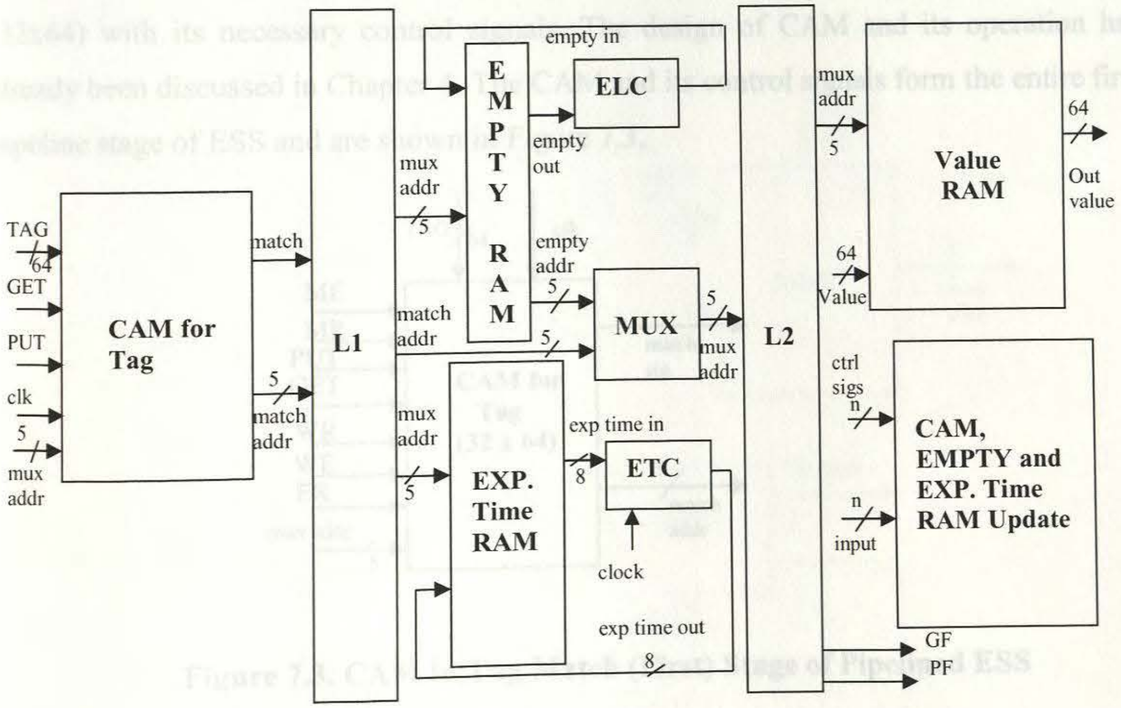


Figure 7.1. High-Level Block Diagram of ESS



L1 – TM / ELTC Stage Latch
L2 – ELTC / EUD Stage Latch
ELC – Empty Location Calculating block
ETC – Expiration Time Calculating block

Figure 7.2. High-Level View of Three-Stage Pipelined ESS

The first stage is the 'Tag Match (TM)' stage in which the tag to match is given as input along with the necessary operations ('GET' or 'PUT') to be performed. As all other components of ESS such as Value, Expiration Time and Empty values are placed in RAM, the second pipeline stage, called the 'Empty Location and Lifetime Check (ELTC)' stage, checks for the lifetime of the corresponding (tag, value) binding in the Expiration Time RAM if there is a tag match or checks for an empty location if it is a 'PUT' operation and there is no match. The third stage, called the 'ESS Update (EUD)' stage, updates the (tag, value) binding in ESS if it is a 'PUT' operation or retrieves a value if it is a 'GET' operation. The failure of a 'GET' operation – 'GET Failed' (GF), and failure of a 'PUT' operation – 'PUT Failed' (PF), is known from stages 1 and 2 respectively. The following sub sections describe each pipeline stage separately in detail.

7.1.1 Tag Match (TM) Stage

The first stage, called the 'Tag Match (TM)' stage, contains only the CAM (32x64) with its necessary control signals. The design of CAM and its operation has already been discussed in Chapter 4. The CAM and its control signals form the entire first pipeline stage of ESS and are shown in Figure 7.3.

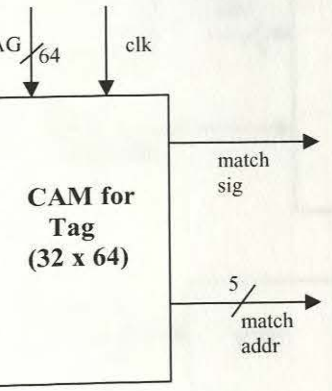
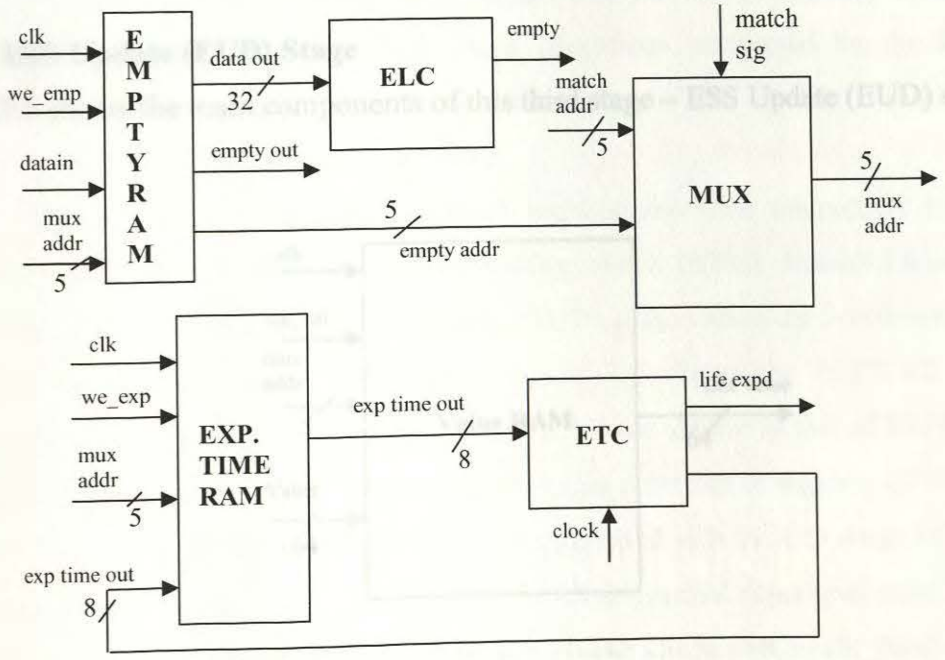


Figure 7.3. CAM in Tag Match (First) Stage of Pipelined ESS

For both 'PUT' and 'GET' functionality, the tag to match is given to CAM along with the specified operation, and if there is a match, the match signal (match sig) goes high along with the corresponding 5-bit match address (match addr) in one clock cycle. The control signals Match Enable (ME), Match Reset (MR), Write Ram (WR), Write Enable (WE) and Erase Ram (ER) are generated internally in this pipeline stage with the existing signals from this stage and control signals from the remaining two stages. 'ME' is activated on either of 'GET' or 'PUT'. This CAM is also staged in the third stage of the pipeline for updating the new tag if it is a 'PUT' operation. So the 'WR' and 'WE' signals get activated on the third stage if it is a 'PUT' operation and there is a no match in the first stage and there is an empty location for the new tag. The 'mux addr' comes from choosing between the empty address and the match address.

7.1.2 Empty Location and Lifetime Check (ELTC) Stage

The ELTC stage, shown in Figure 7.4 is the second stage of the pipelined ESS. It consists of a Multiplexer (MUX) for choosing either the empty address or match address, Empty RAM, Empty Location Calculating (ELC) block, Expiration Time RAM and Expiration Time Calculating (ETC) block. All Components and control signals for the ELTC stage are shown in Figure 7.4.



ELC – Empty Location Calculating block
ETC – Expiration Time Calculating block

Figure 7.4. Components of ELTC (Second) Stage of Pipelined ESS

Dependent on the tag match from the first stage, the address for the whole of ESS is chosen from the multiplexer. The operation of the ELC and ETC are the same as described in Chapter 4. If it is a 'GET' operation, and if there is a match from the first stage, the lifetime for the (tag, value) binding is checked by the ETC block by comparing the current clock value and value being read from the expiration time RAM at the match address location. 'Get Failed (GF)' is generated either from the first stage if there is no match or from the second stage if lifetime of the binding has expired. On success of 'GET', the 'mux addr' is given to the third stage for retrieving the value. If it is a 'PUT' operation and if there is a match from the first stage, the second stage checks for the expiration time to decide whether to update it in the third stage or not. On failure of a match from the first stage, empty ram is checked for an empty location to place this new (tag, value) binding in ESS. 'Put Failed (PF)' is generated in this stage if there is no match and no empty location. Writing to Empty RAM and Expiration RAM takes place in the third stage when needed, to update on a 'PUT' operation for a new (tag, value) pair and on the expiration of lifetime for an existing (tag, value) pair respectively.

7.1.3 ESS Update (EUD) Stage

Figure 7.5 shows the main components of this third stage – ESS Update (EUD) stage.

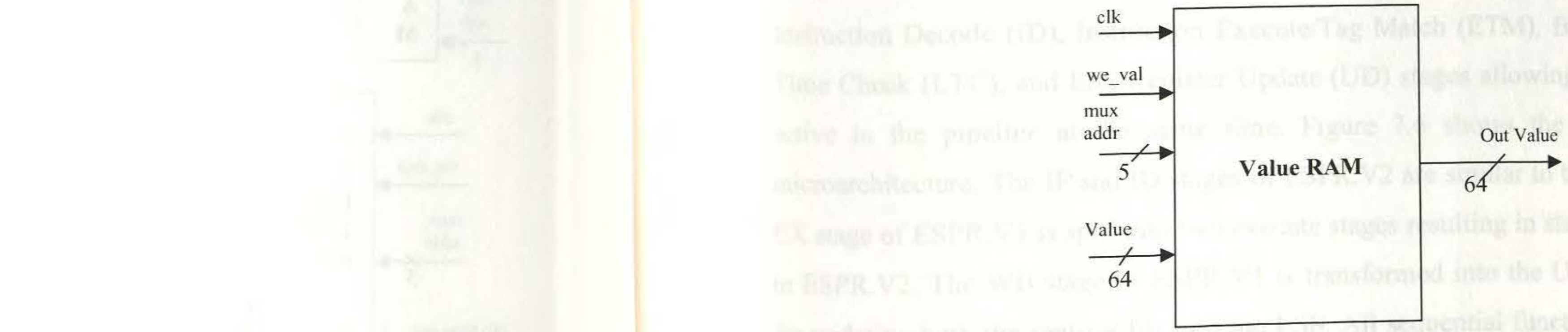


Figure 7.5. Main Component of EUD (Third) Stage of Pipelined ESS

The EUD stage contains the Value RAM for retrieving the value if there is a successful 'GET' operation and for updating (writing) the value if there is a 'PUT' operation. Other components of the ESS such as CAM, Expiration Time RAM and Empty RAM are also updated here in this third stage if it is a 'PUT' operation for a new (tag, value) binding (see Figure 7.2). On account of lifetime expiry for an existing (tag, value) pair on a 'PUT' operation', only the Expiration Time RAM gets updated. No operation is performed in the third stage if 'GET' or 'PUT' fails – 'GF' or 'PF'.

Operations formally performed in five clock cycles in the original ESS organization/architecture have been transformed into a three stage pipelined ESS. The next section deals with how this three staged ESS is incorporated into the existing Four-Stage pipelined ESPR.V1 resulting in a Five-Stage pipelined ESPR.V2 architecture.

7.2. Five-Stage Pipelined ESPR.V2 Architecture

To improve the performance of the ESPR architecture further, the ESS is pipelined as described above, and the ESPR.V1 architecture is further pipelined into ESPR.V2 architecture by including the pipelined ESS and some necessary modifications to the existing ESPR.V1 architecture. Basic operations performed by the ESPR, its functionality, and the macro and micro instructions of the already defined ISA of ESPR.V1 will remain the same for ESPR.V2.

ESPR.V2 is a Five-Stage pipelined architecture with Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute/Tag Match (ETM), Branch Detection/Life Time Check (LTC), and ESS/Register Update (UD) stages allowing 5 instructions to be active in the pipeline at the same time. Figure 7.6 shows the ESPR.V2 pipelined microarchitecture. The IF and ID stages of ESPR.V2 are similar to that of ESPR.V1. The EX stage of ESPR.V1 is split into two execute stages resulting in stages – ETM and LTC in ESPR.V2. The WB stage of ESPR.V1 is transformed into the UD stage of ESPR.V2 for updating both the register file and the ESS. All sequential functional units, including the ESS components in each stage, operate at a Master Clock (MC – clk_pipe) frequency and the Packet RAM operates at twice the MC frequency to enable proper packet processing. The architecture contains full hazard detection and elimination capability in addition to exception handling capability similar to that of ESPR.V1.

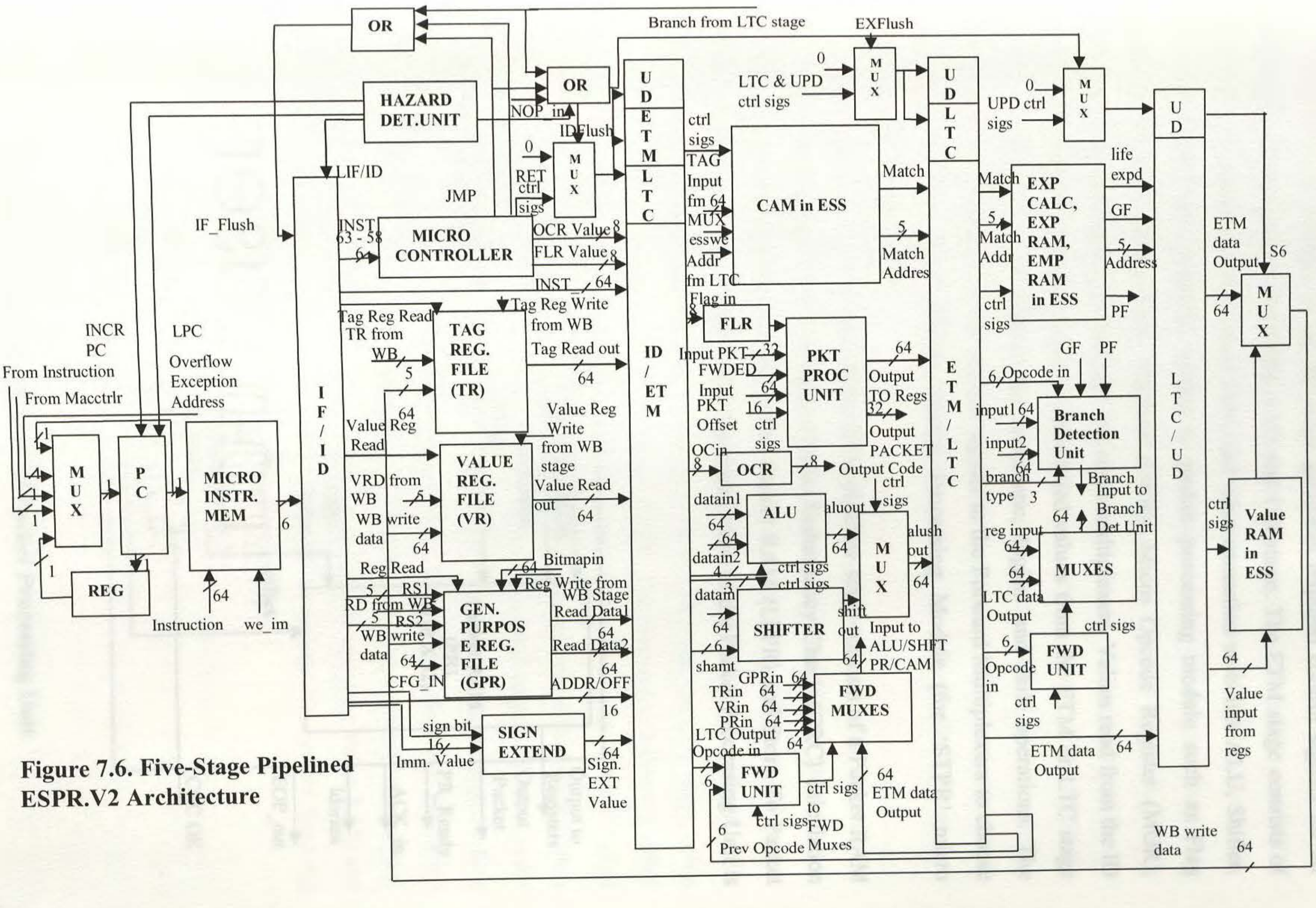


Figure 7.6. Five-Stage Pipelined ESPR.V2 Architecture

The Micro Controller in the ID stage generates required control signals for all remaining functional units in the ETM, LTC and UD stage. The ETM stage consists of the first stage of pipelined ESS – the CAM, and other execution units like ALU, Shifter, Packet Processing unit, registers related to packet processing module such as Flag Register (FLR) and Output Code Register (OCR), Micro Opcode Register (MOR), Forwarding Unit to eliminate the hazards and some multiplexers. Values read from the ID stage register files, sign extend value or forwarded values from the ETM or LTC stage are given to the ALU and Shifter for arithmetic, logical and shift operations. The Forwarding unit is used to provide control signals to the Forward Multiplexers to choose input values for the ALU, Shifter, Packet Processing Module (for 'STPR' micro instruction) and for the CAM.

The Packet Processing Unit of the ETM pipeline stage consists of a Packet RAM (PR), Packet Processing Unit Controller, Cyclic Redundancy Check (CRC) calculation unit and processing modules for Load From Packet RAM (LFPR) and Store To Packet RAM (STPR) instructions. A high-level functional view of the Packet Processing Unit is shown in the following Figure 7.7.

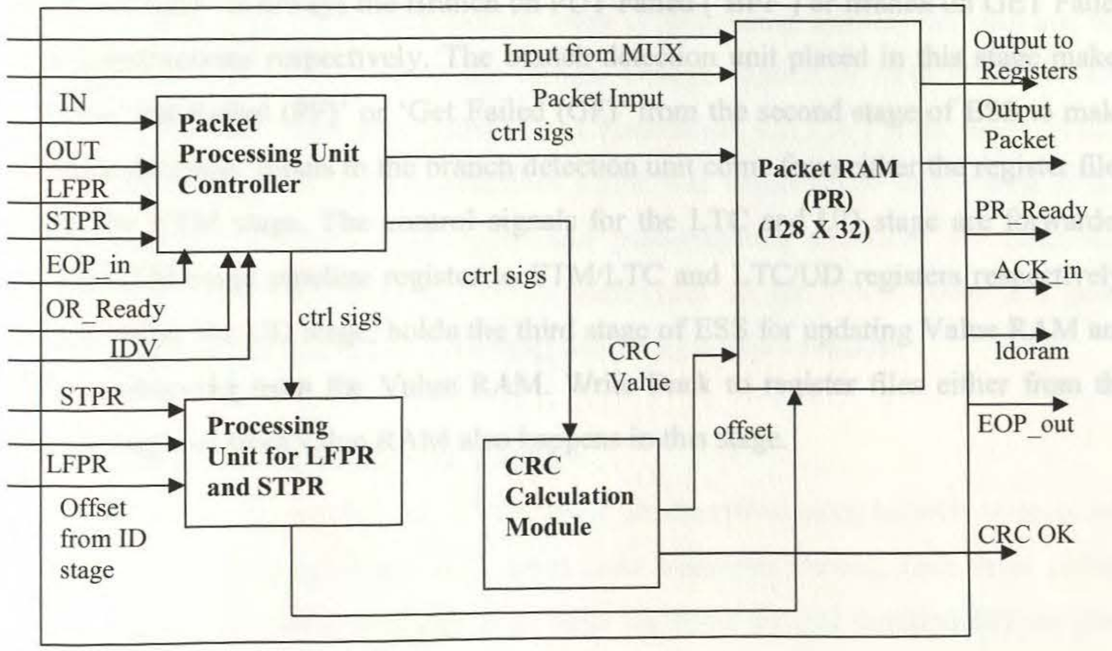


Figure 7.7. High-Level View of Packet Processing Unit

The Controller in the Packet Processing unit generates the necessary control signals for the PR, CRC module and for the processing module for LFPR and STPR instructions. The PR is 32 bits wide to hold the incoming packets in 32-bit blocks in each clock cycle and 128 bits deep to hold the maximum packet size, and can be extended to any size deeper without any change in the existing design. As PR is 32 bits wide, it takes 2 clock cycles for both LFPR and STPR instructions to handle 64 bit data, and so the whole of the packet processing module operates at twice the frequency of the ESPR.V2 pipeline frequency (clk_pipe). The CRC calculation module checks the CRC of incoming packets to precede the further operation of ESP macro instructions. It also calculates CRC for the outgoing ESP packets and places this at the end of the packet before giving it to the output port.

Depending on the micro instructions, the Micro Controller in the ID stage of Figure 7.6 generates the Flag Register code to be placed in FLR and PR and Output Code to be placed in the OCR. The control signals needed for the ETM, LTC and UD stages and necessary inputs to the ETM stage are placed in the ID/ETM pipeline register for further operations of the current micro instruction. The Fourth stage, the LTC stage, holds the Branch Detection Unit and the second stage of ESS. The instruction following either a 'PUT' or 'GET' is always the Branch on PUT Failed ('BPF') or Branch on GET Failed ('BGF') instructions respectively. The branch detection unit placed in this stage makes use of the 'Put Failed (PF)' or 'Get Failed (GF)' from the second stage of ESS to make the branch decision. Inputs to the branch detection unit come from either the register files or from the ETM stage. The control signals for the LTC and UD stage are forwarded from the ETM stage pipeline register to ETM/LTC and LTC/UD registers respectively. The final stage, the UD stage, holds the third stage of ESS for updating Value RAM and CAM or retrieving from the Value RAM. Write Back to register files either from the previous stages or from value RAM also happens in this stage.

Chapter Eight

VHDL Design Capture of ESPR Architectures

Use of a Hardware Description Language (HDL) is one of the best ways to describe a system to make the design vendor-independent, reusable and retargetable. And downloading the HDL design of a system to a FPGA chip makes it more convenient for systems that require reconfigurability. There are various ways of coding using HDLs including Behavioral Coding Style, Register Transfer Level (RTL) coding style and Gate-Level Coding Style. Behavioral level Coding Style used to describe a system is the easiest method and is also easy to understand. But the synthesizing and implementing CAD tool may not synthesize and implement this design to operate as needed. Necessary modifications can be made in the existing behavioral design or coding styles can be combined to make the CAD tool implement the design in a silicon chip efficiently. After the architectural design of the ESPR has been developed, most time is spent in design description using VHDL for the functional modules, ways to ameliorate them and the application of constraints on the existing design using the CAD tool for improvement in functional and timing performance. This Chapter discusses the HDL design approach used in describing the ESPR architecture, ways of initializing memory on chip and the constraints that can be applied to the design. Design capture of both ESPR.V1 and ESPR.V2 architectures was done using Xilinx Foundation 4.2i CAD tools, using VHDL as a description language and the described ESPR was implemented to a Virtex FPGA chip. The design was then synthesized and post-synthesis simulated for functional validation and then implemented (virtual prototype) to the FPGA chip and post-implementation simulation tested for timing and performance validation.

8.1 Design Partitioning and Design Capture

Most of the functional units of the ESPR are described using behavioral level and a combination of behavioral and RTL level code whenever needed. Gate level coding style is also used for some modules to achieve the exact desired functionality on chip. The whole of ESPR.V1 and ESPR.V2 is designed based on a bottom-up, hierarchical and modular approach. Since the design of the pipelined architectures involved many functional units, it was necessary to design and test the individual lower level modules before using them to design a whole processor. And so the whole design of ESPR is partitioned into separate stages, and it became easy to separate them on the basis of their pipeline stages. Bottom-up and hierarchical level coding is needed in such a design of interfacing separate functional modules, and it has to be made sure that each of the low level modules function correctly. The modular approach also helps to separate out individual modules and to reuse them if they have identical functionality. Figure 8.1 and Figure 8.2 illustrate how the code was laid out at a high level for ESPR.V1 and ESPR.V2 and the organization of functional units in the individual pipeline stages are shown in the following figures.

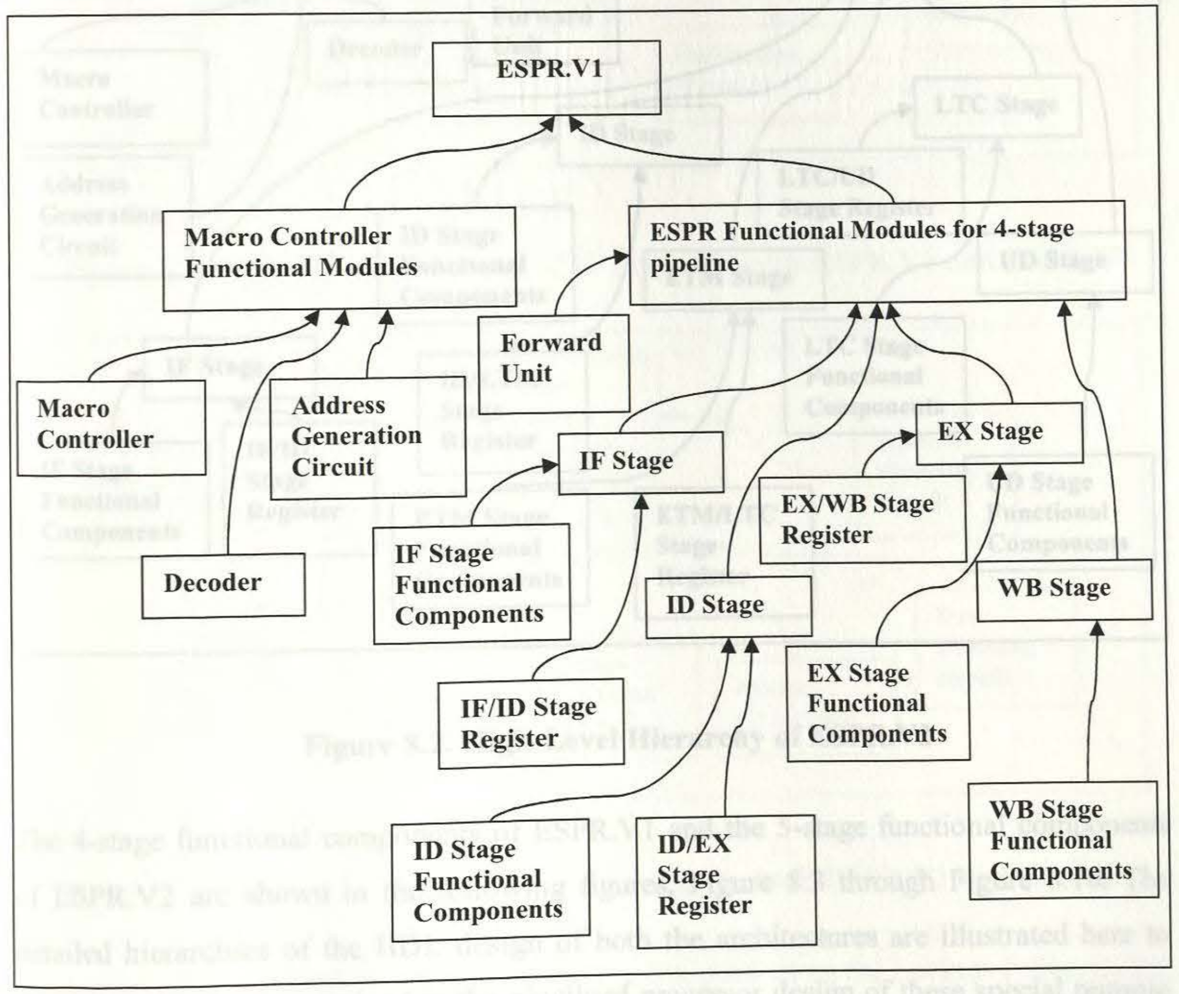

ESPR.V1
Macro Controller Functional Modules
ESPR Functional Modules for 4-stage pipeline
Forward Unit
Macro Controller
Address Generation Circuit
IF Stage
EX Stage
EX/WB Stage Register
Decoder
IF Stage Functional Components
ID Stage
WB Stage
IF/ID Stage Register
EX Stage Functional Components
ID Stage Functional Components
ID/EX Stage Register
WB Stage Functional Components

Figure 8.1. High-Level Hierarchy of ESPR.V1

ESPR.V2 has the same functional hierarchy as that of ESPR.V1 except the EX stage is split into Execute/Tag Match (ETM) stage and Branch Detection and Lifetime Check (LTC) stage, and the WB stage is transformed into the updating stage for ESS and Register files – ESS/Reg. Update (UD) stage.

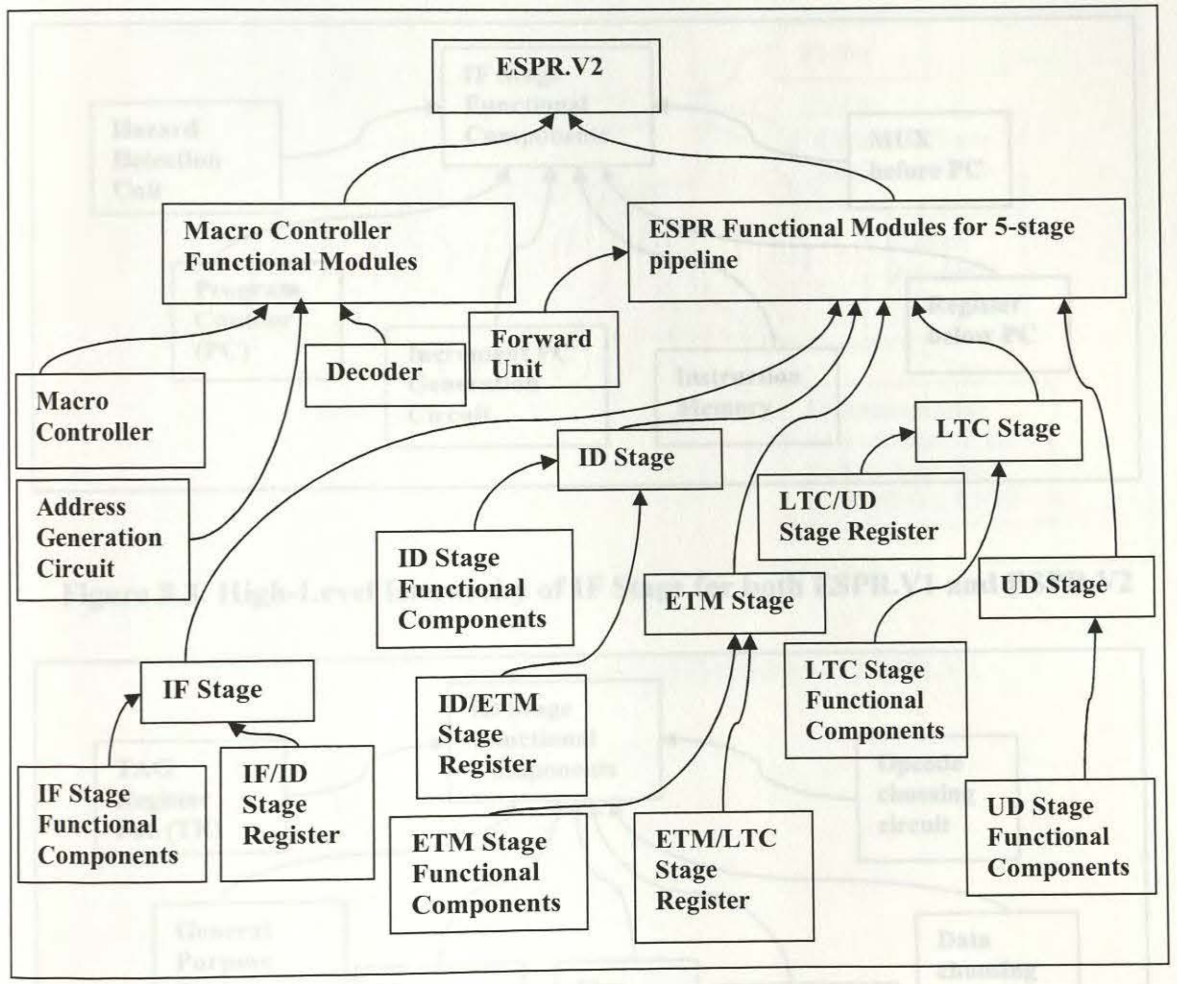

ESPR.V2
Macro Controller Functional Modules
ESPR Functional Modules for 5-stage pipeline
Forward Unit
Decoder
Macro Controller
Address Generation Circuit
ID Stage
LTC Stage
LTC/UD Stage Register
ID Stage Functional Components
ETM Stage
UD Stage
IF Stage
ID/ETM Stage Register
LTC Stage Functional Components
IF Stage Functional Components
IF/ID Stage Register
ETM Stage Functional Components
ETM/LTC Stage Register
UD Stage Functional Components

Figure 8.2. High-Level Hierarchy of ESPR.V2

The 4-stage functional components of ESPR.V1 and the 5-stage functional components of ESPR.V2 are shown in the following figures, Figure 8.3 through Figure 8.10. The detailed hierarchies of the HDL design of both the architectures are illustrated here to show the complexity involved in the pipelined processor design of these special purpose architectures. At most, care is taken in the HDL description of individual functional modules and they are optimized for speed on-chip rather than the area of the chip. After successful synthesis, simulation and implementation, the performance and the area occupied by the design in the chip are compared. Figure 8.3 and Figure 8.4 shows the hierarchy of the IF and ID stage functional components respectively that are utilized by both ESPR.V1 and ESPR.V2.

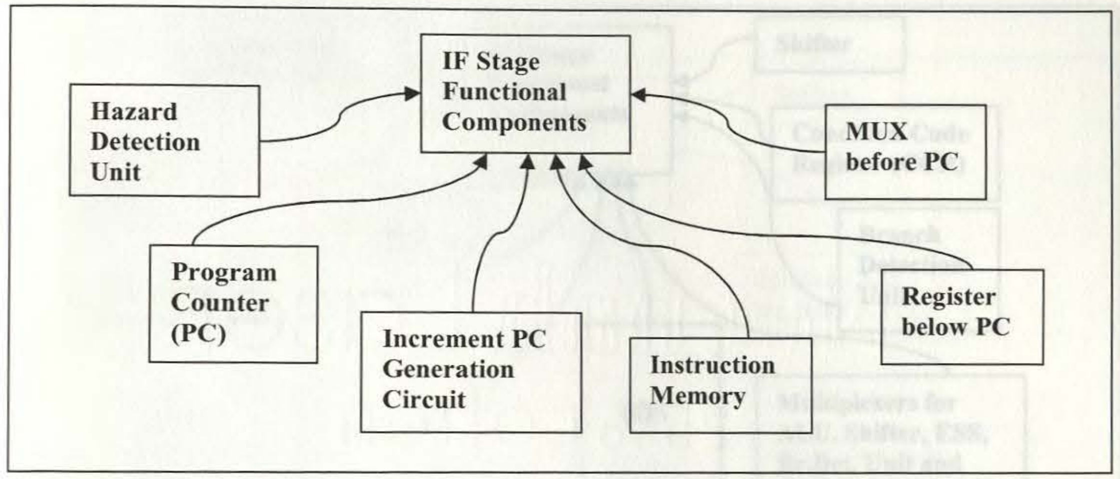


Figure 8.3. High-Level Hierarchy of IF Stage for both ESPR.V1 and ESPR.V2

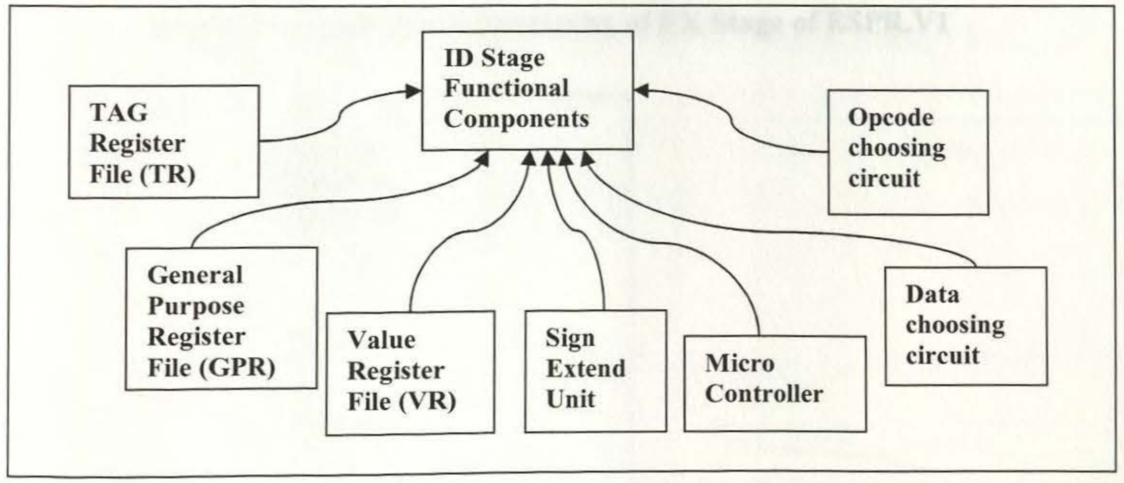


Figure 8.4. High-Level Hierarchy of ID Stage for both ESPR.V1 and ESPR.V2

Instruction memory in the IF stage was initially designed using the Lookup Tables (LUT) of the Virtex FPGA in ESPR.V1 design and later modified to use the core block RAM available on chip to give a significant performance improvement in memory design.

Various options for designing the register files GPR, TR and VR of the ID stage were studied, coded in VHDL and tested and an optimized final design is used in the ESPR.V1 and ESPR.V2 architecture. The following Figure 8.5 and Figure 8.6 shows the high level hierarchy of EX and WB stages of ESPR.V1 respectively.

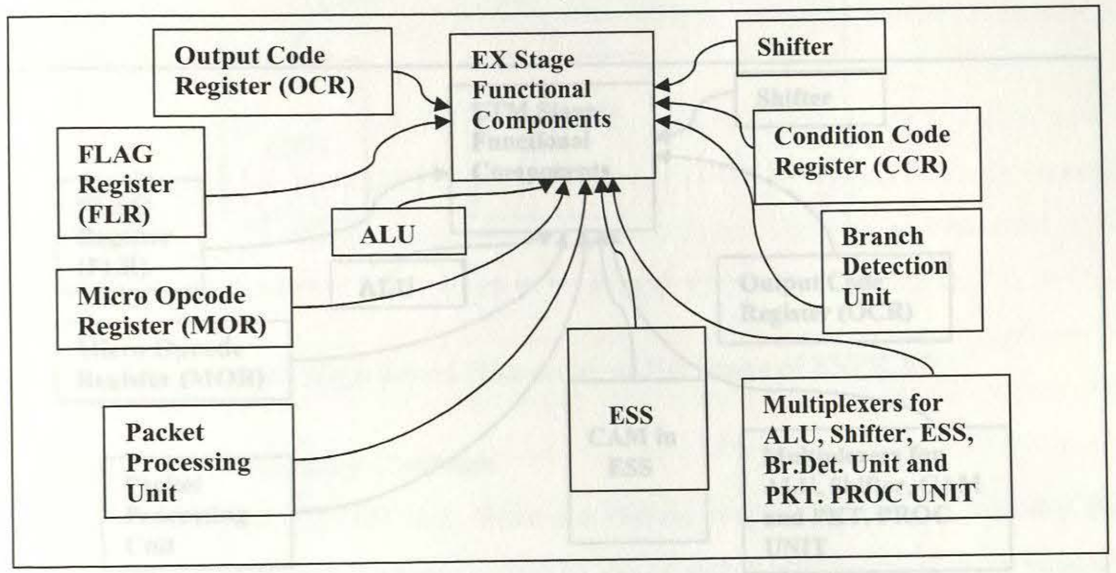

Output Code Register (OCR)
EX Stage Functional Components
Shifter
Condition Code Register (CCR)
FLAG Register (FLR)
ALU
Branch Detection Unit
Micro Opcode Register (MOR)
ESS
Multiplexers for ALU, Shifter, ESS, Br.Det. Unit and PKT. PROC UNIT
Packet Processing Unit

Figure 8.5. High-Level Hierarchy of EX Stage of ESPR.V1

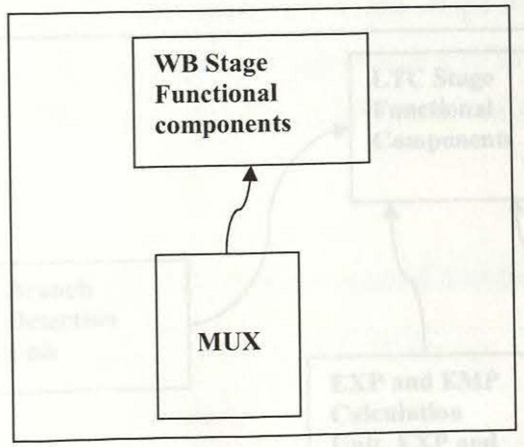


Figure 8.6. High-Level Hierarchy of WB Stage of ESPR.V1

The Packet Processing unit and ESS of the EX stage have their own internal functional components that can be seen from the previous chapters. They are not shown here. The packet processing unit is the same for both architectures, ESPR.V1 and ESPR.V2. The whole of ESS placed in the EX stage of ESPR.V1 is split into three stages in ESPR.V2 as can be seen from the high-level hierarchy of the ETM, LTC and UD stages of ESPR.V2. Figures 8.7, 8.8 and 8.9 illustrate the ETM, LTC and UD stages of ESPR.V2 respectively.

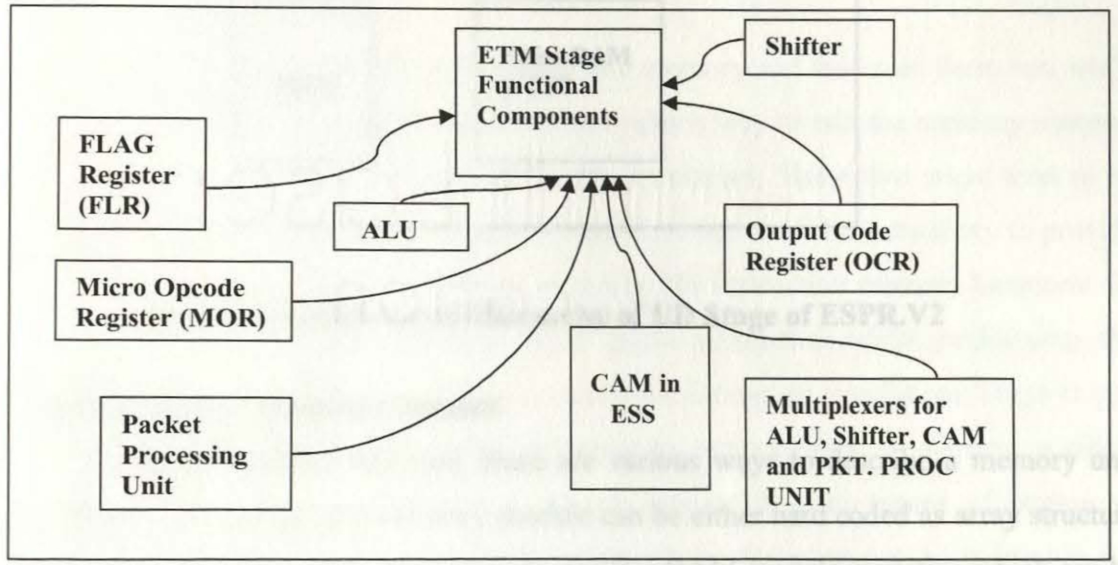

ETM Stage Functional Components
Shifter
FLAG Register (FLR)
ALU
Output Code Register (OCR)
Micro Opcode Register (MOR)
CAM in ESS
Multiplexers for ALU, Shifter, CAM and PKT. PROC UNIT
Packet Processing Unit

Figure 8.7. High-Level Hierarchy of ETM Stage of ESPR.V2

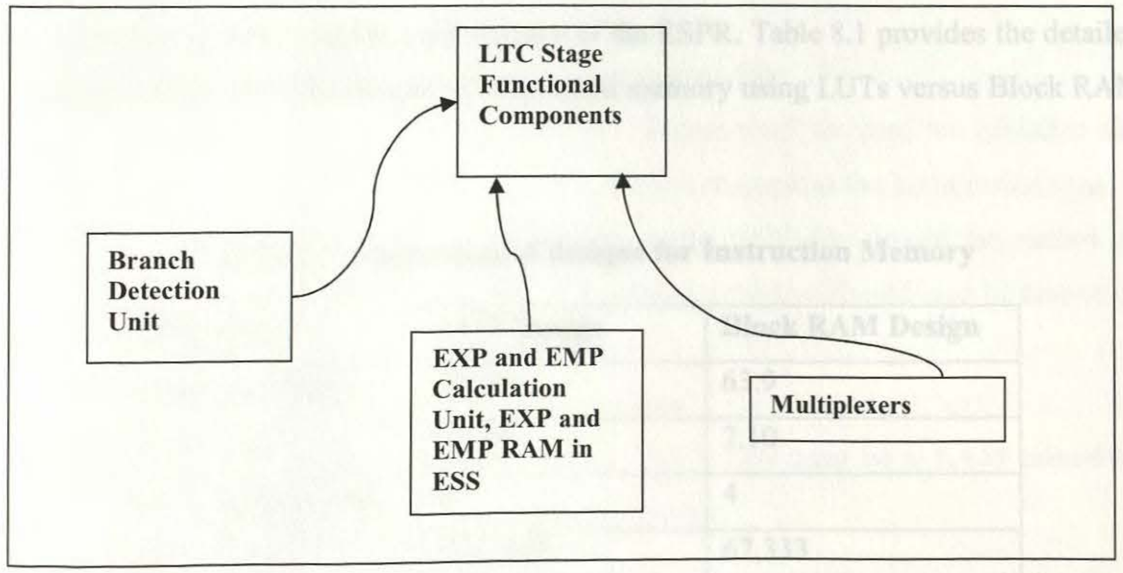

LTC Stage Functional Components
Branch Detection Unit
EXP and EMP Calculation Unit, EXP and EMP RAM in ESS
Multiplexers

Figure 8.8. High-Level Hierarchy of LTC Stage of ESPR.V2

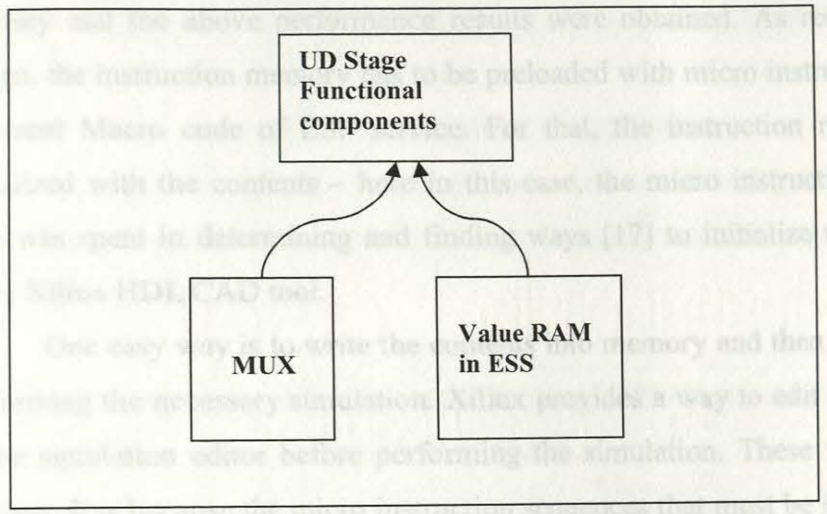


Figure 8.9. High-Level Hierarchy of UD Stage of ESPR.V2

8.2 Initializing the Memory Contents

Using the Xilinx CAD tool, there are various ways to describe a memory unit using HDLs. The design of a memory module can be either hard coded as array structure storage, or by using the stack of an already existing RAM module primitive which can be implemented as LUTs on chip, or by using the block core RAM memory available. Out of the three ways described above, the usage of core RAM turned out to be the most efficient and resulted in higher performance of the ESPR. Table 8.1 provides the detailed comparison chart for both designs of instruction memory using LUTs versus Block RAM design.

Table 8.1. Comparison of designs for Instruction Memory

Parameters	LUT Design	Block RAM Design
Frequency (MHz)	27.59	63.9
Delay (ns)	24.37	7.10
Block RAMs Used	0	4
Gate Count	136, 553	67,333
Number of Slices	730/19,200 (3%)	87/19,200 (1%)

LUT and Block RAM design were used in testing the design of the instruction memory and the above performance results were obtained. As required by the ESPR design, the instruction memory has to be preloaded with micro instruction sequences that represent Macro code of ESP service. For that, the instruction memory needs to be initialized with the contents – here in this case, the micro instruction sequences. More time was spent in determining and finding ways [17] to initialize the memory contents using Xilinx HDL CAD tool.

One easy way is to write the contents into memory and then read them out, while performing the necessary simulation. Xilinx provides a way to edit the memory contents in the simulation editor before performing the simulation. These two ways tend to be fruitless. It is because the micro instruction sequences that must be in memory to provide any ESP service is huge and occupies up to nearly 256 instruction memory locations. So it is troublesome to write each and every micro instruction while performing the simulation, and also difficult to edit the contents each time on simulation. There is one other way in which the memory can be initialized by writing using the constraints editor [17] provided by Xilinx. The same method can also be done by means of writing an external constraints file prior to synthesis of the whole design or can be written in the VHDL design file for the instruction memory. The following description shows these two ways of initializing the instruction memory.

8.2.1. Initializing a RAM Primitive via a Constraints File

A 'NCF' (Netlist Constraints File) – 'filename.ncf' is used to initialize the memory contents. The NCF file must have the same root name as the input netlist (e.g., if the input netlist name was 'inst_mem.edf' then the NCF file should be named as 'inst_mem.ncf', and the instance name(s) of the RAM primitive should also be known. It should be written in the NCF file as follows,

INST instname INIT = Value

where 'instname' is the instance name of the RAM. This must be a RAM primitive, enclosed in quotes and 'Value' is a hexadecimal number.

For example, if the instance name of 'RAM32x1s' primitive is 'RAM1' then the contents of 'RAM1' could be set in the NCF file by placing the following line in a NCF file.

INST "RAM1" INIT = ABCD0000;

The following example gives a clear picture of how initializing the instruction memory can be done using the way described above. Consider the following instruction sequence to be initialized into memory.

MOV R3, R1 – 1C61000001000000 (Eq. HEX Value for instruction)
ADD R4, R3, R3 – 2483180001000000
ADD R5, R4, R3 – 24A4180001000000

The hexadecimal numbers on the right is the equivalent value for the micro instructions on the left, and the 64-bit values are laid out in order of (63 down to 0). Figure 8.10 shows the contents of the NCF file for the above sequence. The instance name "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R320/R321" describes the level of hierarchy with the top level module 'esprcomp' in the left and the lowest level module 'R321' at the end.

8.2.2. Initializing a Block RAM in VHDL

The block RAM structures can be initialized in VHDL for synthesis and simulation. The VHDL code uses a 'generic' to pass the initialization. The generic types are not supported by the present day Synopsys FPGA compiler, and a built-in dc_script (e.g., translate_off) is used to attach the attributes to the RAM. The following Table 8.2 illustrates the RAM initialization properties to be used along with the generics in VHDL. Figure 8.11 shows the example instruction sequence starting from address location zero for the VHDL code described below. Figure 8.12 shows an example VHDL code for initializing the block RAM for the instruction memory.

```
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R320/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R321/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R322/R321" INIT=00000000;
                              .
                              .
                              .
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3223/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3224/R321" INIT=00000007;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3225/R321" INIT=00000000;
                              .
                              .
                              .
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3243/R321" INIT=00000006;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3244/R321" INIT=00000006;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3245/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3246/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3247/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3248/R321" INIT=00000003;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3249/R321" INIT=00000002;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3250/R321" INIT=00000004;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3251/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3252/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3253/R321" INIT=00000005;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3254/R321" INIT=00000001;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3255/R321" INIT=00000006;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3256/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3257/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3258/R321" INIT=00000007;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3259/R321" INIT=00000001;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3260/R321" INIT=00000001;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3261/R321" INIT=00000006;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3262/R321" INIT=00000000;
INST "esprcomp/IFFULL/ifpipecomp/instrmemnew/IMEM/R3263/R321" INIT=00000000;
```

Figure 8.10. NCF file for Initializing Instruction Memory

Table 8.2. Block RAM Initialization Properties

Property	**Memory Cells**
INIT_00	255 – 0
INIT_01	511 - 256
…………	…………
…………	…………
INIT_0F	4095 - 3840

```
IN PKT
NOP
MOVI R4, 1
ADD R5, R4, R3
MOV TR1, R5
MOV VR1, R5
PUT TR1, VR1
BPF ADDR1 (0X41)
NOP
NOP
LFPR  <O 3>  TR1
GET TR1, VR1
BGF ADDR2 (0X1B)
NOP
NOP
INCR R4, VR1
```

Figure 8.11. Example Micro Instruction Sequence

```
-- Instruction Memory Design using Block RAM

library IEEE;
use IEEE.std_logic_1164.all;

--synopsys translate_off;
library unisim;
use unisim.vcomponents.all;
--synopsys translate_on;

entity INSTMEM is
port(clk, we, en, rst: in std_logic;
     addr: in std_logic_vector(7 downto 0);
     inst_in: in std_logic_vector(63 downto 0);
     inst_out: out std_logic_vector(63 downto 0));
end entity INSTMEM;

architecture behavioural of INSTMEM is

component RAMB4_S16 is
port(ADDR: in std_logic_vector(7 downto 0);
     CLK: in std_logic;
     DI: in std_logic_vector(15 downto 0);
     DO: out std_logic_vector(15 downto 0);
     EN, RST, WE: in std_logic);
end component RAMB4_S16;
```

Figure 8.12. VHDL Code for Instruction Memory using Block RAM

```
attribute INIT_00: string;
attribute INIT_01: string;
attribute INIT_02: string;
attribute INIT_03: string;
attribute INIT_04: string;
attribute INIT_05: string;
attribute INIT_06: string;
attribute INIT_07: string;
attribute INIT_08: string;
attribute INIT_09: string;
attribute INIT_0A: string;
attribute INIT_0B: string;
attribute INIT_0C: string;
attribute INIT_0D: string;
attribute INIT_0E: string;
attribute INIT_0F: string;

attribute INIT_00 of Instram0 : label is
"00000000000006C0000000C00000000010400000000000000000004000000000";
attribute INIT_00 of Instram1 : label is
"0100000000000000800000000000000000000000810001000100000000000000";
attribute INIT_00 of Instram2 : label is
"0001000000000000004100600000000000000041000100601800000000000000";
attribute INIT_00 of Instram3 : label is
"2C80000000008000780054000000000084007C001C051C0524A4208000000400";

begin
Instram0: RAMB4_S16
--synopsys translate_off
GENERIC MAP ( INIT_00 =>
X"00000000000006C0000000C00000000010400000000000000000004000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(15 downto 0), DO=>inst_out(15
downto 0), EN=>en, RST=>rst, WE=>we);

Instram1: RAMB4_S16
--synopsys translate_off
GENERIC MAP ( INIT_00 =>
X"0100000000000000800000000000000000000000810001000100000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(31 downto 16), DO=>inst_out(31
downto 16), EN=>en, RST=>rst, WE=>we);

Instram2: RAMB4_S16
--synopsys translate_off
GENERIC MAP ( INIT_00 =>
X"0001000000000000004100600000000000000041000100601800000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(47 downto 32), DO=>inst_out(47
downto 32), EN=>en, RST=>rst, WE=>we);

Instram3: RAMB4_S16
--synopsys translate_off
GENERIC MAP ( INIT_00 =>
X"2C80000000008000780054000000000084007C001C051C0524A4208000000400")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(63 downto 48), DO=>inst_out(63
downto 48), EN=>en, RST=>rst, WE=>we);
end architecture behavioural;
```

Figure 8.12. VHDL Code for Instruction Memory using Block RAM (continued)

8.3 Timing Constraints

The Xilinx 4.2i Foundation CAD tool provides a means (constraints editor) for specifying constraints for timing, placement, mapping, routing etc., on the specific design to provide some performance improvements in terms of area and/or speed of the design. It is up to a designer to consult the Xilinx Constraints Guide [17] and apply their own needed constraints to the design. The constraints can be externally specified using a 'UCF' (User Constraints File) or can be described using the constraints editor. In the ESPR architecture design, only timing constraints of specific extent were applied to ESPR.V1 and ESPR.V2 to determine the performance. And it is believed that, more optimum performance of any design can be obtained by applying more tight constraints at the expense of longer synthesis and implementation time.

Chapter Nine

Post-Implementation Simulation Validation of ESPR.V2 Architecture

Following the design layout and design description of the ESPR.V2, the next step is to synthesize and simulate the design. After HDL post-synthesis simulation to validate functional correctness, and prior to implementing and prototyping the design on a FPGA prototype board, the ESPR.V2 has to be validated for both functional and timing (performance) correctness via Post-Implementation simulation. This process is referred to as virtual hardware prototyping as it involves timing validation of the system. This section presents the Post-Implementation HDL simulation validation of the ESPR.V2 architecture. Simulation results are presented in a step-by-step fashion. The ESPR.V2 was first simulated executing single micro instructions to validate their correct functional and timing operations. The ESPR.V2 architecture was then simulated executing short sequences of micro instructions. Lastly, it was validated that the ESPR.V2 architecture correctly executes all macro instructions for which it was developed to execute. Post Implementation simulation validation of the architectural design was performed on a PC (Personal Computer) system – Pentium III 550 MHz Processor, with Windows 2000 platform and 640 MB (Megabytes) of RAM memory. The utilized HDL simulator is contained within the Xilinx Foundation 4.2i CAD tool set utilized during this research project. The logic resources utilized in the Xilinx Virtex2 – 4000 FPGA chip to implement the described ESPR.V2 architecture is given in the following Table 9.1.

Table 9.1 Logic Resources Utilization for ESPR.V2 Architecture

Resources	**Utilization**
4 Input LUTs	5,902
Flip flops	916
Block RAMs	33
Equivalent System Gates	2,256,291

9.1 Validation of Correct Execution of Single Micro Instructions

All the micro instructions described for the ESPR.V2 architecture were tested separately for their functional and timing correctness. The individual pipeline stages for each instruction were also tested for proper generation of control and data signals. This section presents two micro instructions flowing through the five pipeline stages to show correct execution in all stages. The first micro instruction to be presented is the Shift Right (SHR) micro instruction in the following Figures 9.1a and 9.1b. The micro instruction with its equivalent Hex. Format is given as,

SHR R7, R4, 1 – 48E4000001000001 (Hex. Format)

From Figure 9.1a, after the Instruction Fetch (IF) stage at approximately 10 μs, the SHR instruction is read out from 'instchk' value. Prior to that, a value of '0xA' is written into register R4 using the 'MOVI' micro instruction to be used by the 'SHR' instruction to result in a value of '0x5' in register R7. The opcode for the SHR instruction (0x12) is decoded and read from the 'op2o' variable at the end of the Instruction Decode (ID) stage at approximately 12 μs. The Shifter is placed in the ETM stage of ESPR.V2 and the value of '0xA' from register R4 is shifted right by one position as specified by the instruction and a value of '0x5' is read out from the ETM stage.

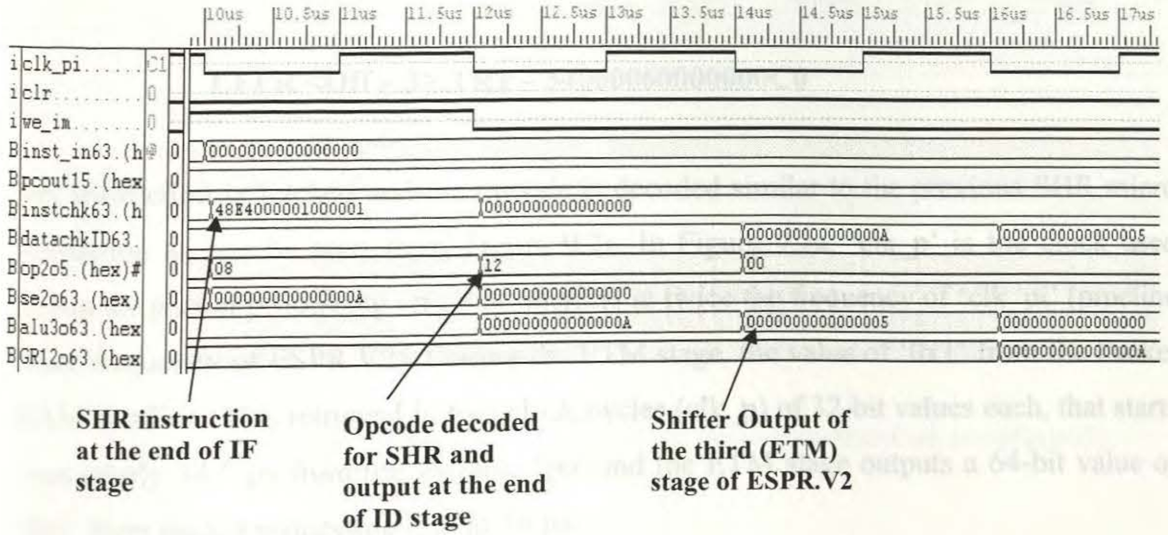
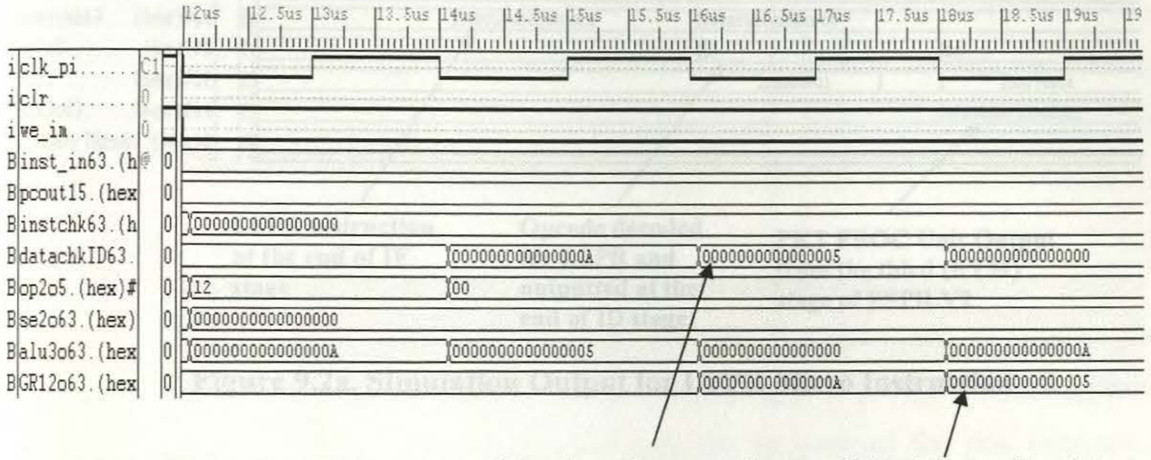


Figure 9.1a. Simulation Output for SHR Micro Instruction

Figure 9.1b shows the continuation of the simulation output of the SHR micro instruction. The LTC stage passes the data value of '0x5' and the value is written back to register R7 during the UD stage. This can be seen from the value also being read out by the variable 'GPR12o' of Figure 9.1b at approximately 18 μs.


12us
12.5us
13us
13.5us
14us
14.5us
15us
15.5us
16us
16.5us
17us
17.5us
18us
18.5us
19us
19
i clk_pi...... C1
i clr......... 0
i we_im....... 0
B inst_in63.(h 0
B pcout15.(hex 0
B instchk63.(h 0 0000000000000000
B datachkID63. 0 000000000000000A 0000000000000005 0000000000000000
B op2o5.(hex)# 0 12 00
B se2o63.(hex) 0 0000000000000000
B alu3o63.(hex 0 000000000000000A 0000000000000005 0000000000000000 000000000000000A
B GR12o63.(hex 0 000000000000000A 0000000000000005

Pass through output of the fourth (LTC) stage

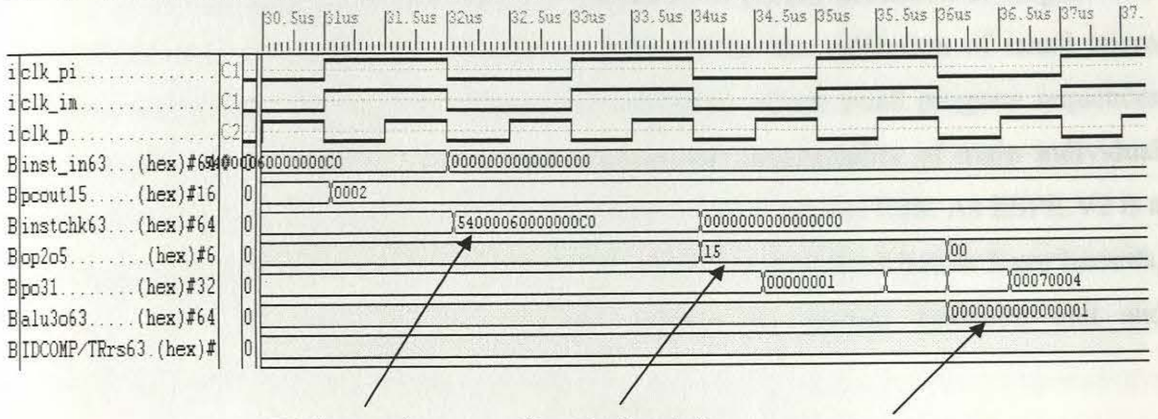
Value being written into register R7 during UD stage

Figure 9.1b. Simulation Output for SHR Micro Instruction (continued)

Figures 9.2a and 9.2b show another micro instruction – 'Load From Packet RAM (LFPR)'. This instruction utilizes the packet processing unit. LFPR with its hexadecimal code is given as,

LFPR <Off – 3> TR1 – 54000060000000C0

The instruction is fetched and the opcode is decoded similar to the previous SHR micro instruction as can be seen from Figure 9.2a. In Figure 9.2a, 'clk_p' is the clock used within the packet processing unit that operates at twice the frequency of 'clk_pi' (pipeline clock frequency of ESPR.V2). During the ETM stage, the value of '0x1' from the Packet RAM at offset '3' is retrieved in two clock cycles (clk_p) of 32-bit values each, that starts from nearly 34.5 μs from the variable 'po' and the ETM stage outputs a 64-bit value of '0x1' from packet processing unit at 36 μs.



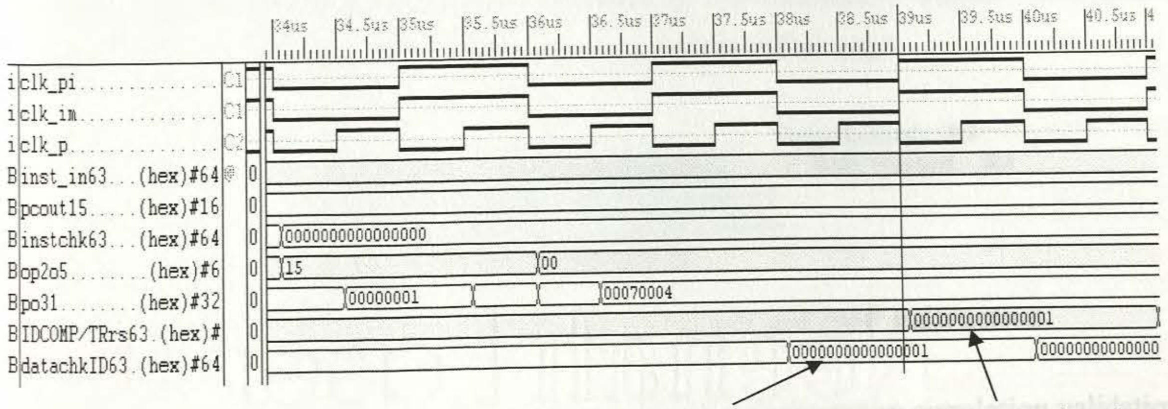
LFPR instruction at the end of IF stage

Opcode decoded for LFPR and outputted at the end of ID stage

PKT PROC Unit Output from the third (ETM) stage of ESPR.V2

Figure 9.2a. Simulation Output for LFPR Micro Instruction

Figure 9.2b shows the continuation of the simulation output for the LFPR instruction. At 38 μs, the output value for the LFPR instruction is passed through the LTC stage and during the UD stage the value of ‘0x1’ is written to the tag register TR1.



Pass through output of the fourth (LTC) stage

Value being written into tag register TR1 during UD stage

Figure 9.2b. Simulation Output for LFPR Micro Instruction (continued)

9.2 Small Micro Program and Individual Functional Unit Testing of ESPR.V2

In the testing process of ESPR.V2, validation of proper execution of single micro instructions was first achieved and will be followed by the validation of small micro program sequences. This section presents the validation of two small program sequences and a couple of instructions which also validates the functionality of main individual functional units such as the ALU Unit, Packet Processing Unit and ESS. As ESPR.V2 is a five-stage pipelined processor, micro program sequences explained below have hazards, and the following simulation results also validate the hazard detection unit and forwarding unit that eliminates the hazards.

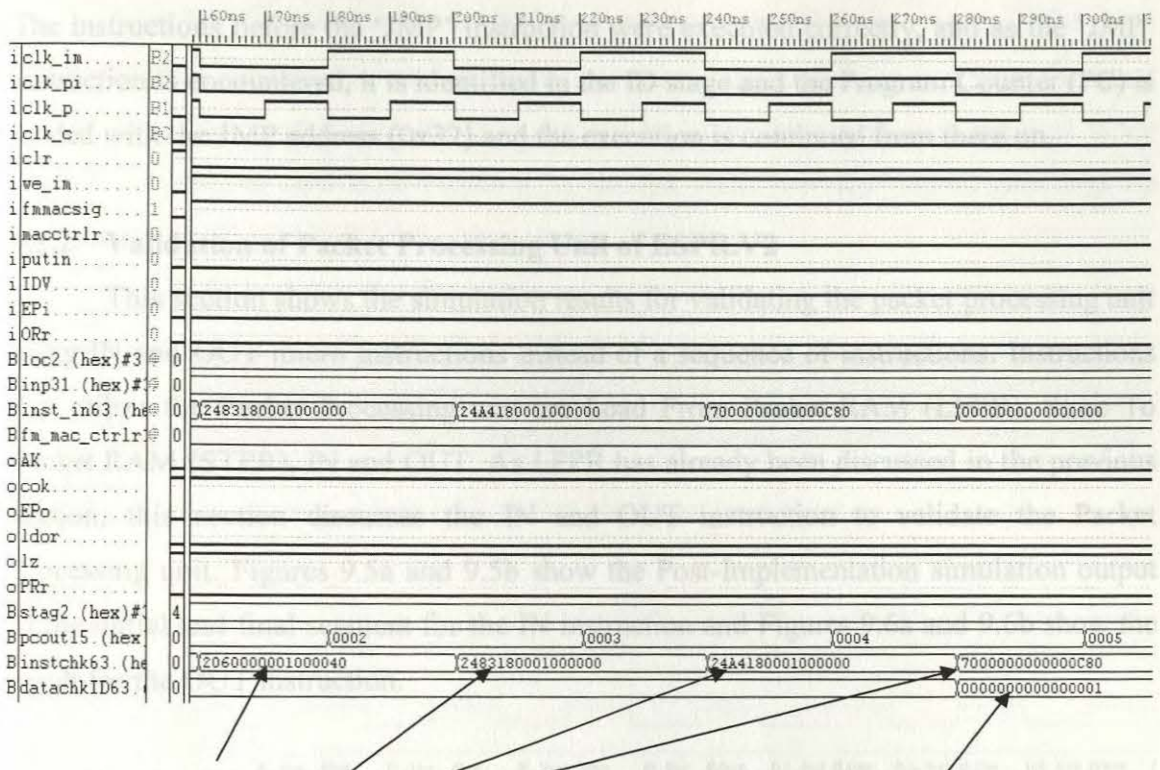
9.2.1 Validation of ALU Unit and JMP Instruction of ESPR.V2

Figure 9.3 shows the micro instruction program sequence used to validate the ALU unit and JMP instruction and also provides an example to test the forwarding unit. The micro instruction memory is preloaded with the bit patterns for this instruction sequence. As can be seen from the program sequence, a data hazard arises when the next instruction in sequence in the ID stage wants to read the new value before the data is written into the same hazard prone register in the UD stage. The forwarding unit of ESPR.V2 takes care of the hazardous situation by forwarding the needed value either from the ETM stage or the LTC stage to the input of the ETM stage.

```
                                            Data Hazard – R3
0. MOVI R3, 1 – 2060000001000040              Data Hazard – R4
1. ADD R4, R3, R3 – 2483180001000000
2. ADD R5, R4, R3 – 24A4180001000000
3. JMP 32h – 7000000000000C80
```

Figure 9.3. Program for Validating ALU Unit and JMP Instruction

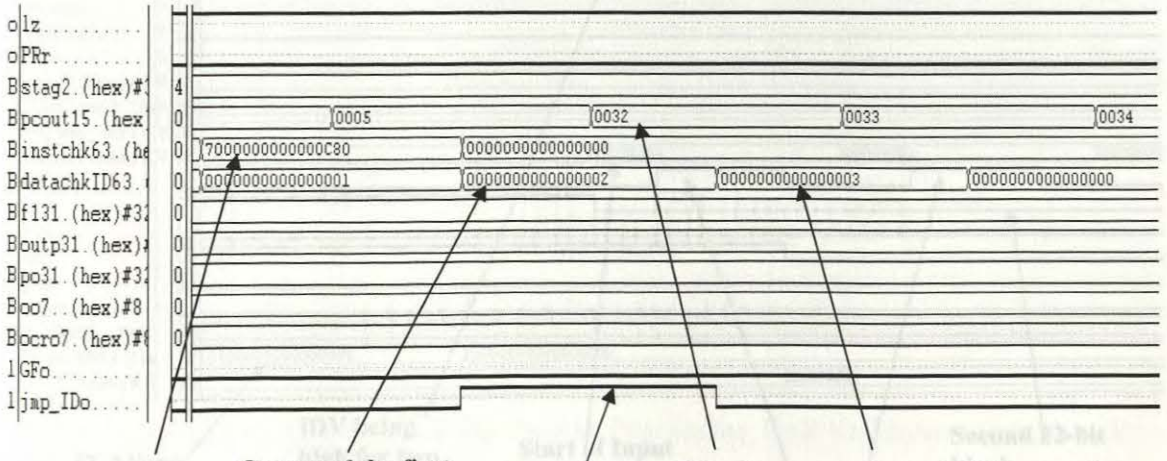
The following Figures 9.4a and 9.4b show the Post-Implementation simulation validation for the above program sequence.



Fetching of instructions of the micro program sequence from instruction memory

Output of the first MOVI instruction at the UD stage (5th stage)

Figure 9.4a. Simulation Output for ALU Unit and JMP Instruction Validation



Jump Micro Instruction

Output of the first ADD instruction at the UD stage (5th stage)

JUMP signal

PC value changed to the JUMP address

Output of the second ADD instruction at the UD stage (5th stage)

Figure 9.4b. Simulation Output for ALU Unit and JMP Instruction Validation (continued)

The instructions before the 'JMP' instruction were executed correctly, and as the 'JMP' instruction is encountered, it is identified in the ID stage and the Program Counter (PC) is loaded with the JMP address (0x32) and the execution is continued from there on.

9.2.2 Validation of Packet Processing Unit of ESPR.V2

This section shows the simulation results for validating the packet processing unit using IN and OUT micro instructions instead of a sequence of instructions. Instructions that utilize the Packet Processing unit are Load From Packet RAM (LFPR), Store To Packet RAM (STPR), IN and OUT. As LFPR has already been discussed in the previous section, this section discusses the IN and OUT instruction to validate the Packet Processing unit. Figures 9.5a and 9.5b show the Post-Implementation simulation output of the initial and final segment for the IN instruction and Figures 9.6a and 9.6b show the result for the OUT instruction.

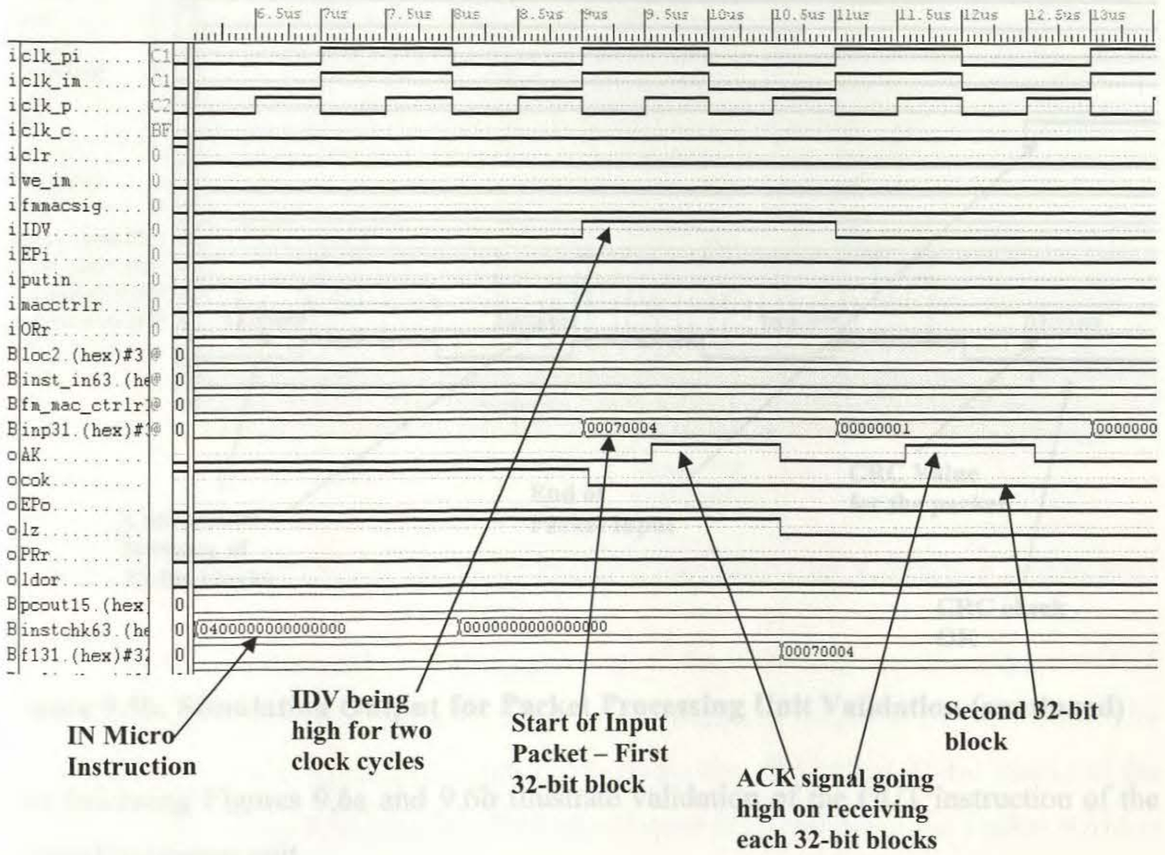


Figure 9.5a. Simulation Output for Packet Processing Unit (IN) Validation

After the IN instruction is fetched from memory, the IDV signal in the ETM stage has to be high for two Packet Processing unit clock cycles (clk_p) and then the input packet from the Input Packet RAM is fetched in 32-bit blocks in each pipeline clock (clk_pi). As can be seen from Figure 9.5a, the AK (Acknowledge) signal goes high on receiving each 32-bit block of the input packet.

Figure 9.5b shows the final segment of the IN instruction. The end of the input packet is determined by the EPi (End of Packet Input) signal going high, when receiving the Cyclic Redundancy Check (CRC) for the packet. On receiving the CRC for the input packet, the CRC calculation unit performs the CRC check for the entire packet and the execution for the corresponding ESP packet continues from there on.

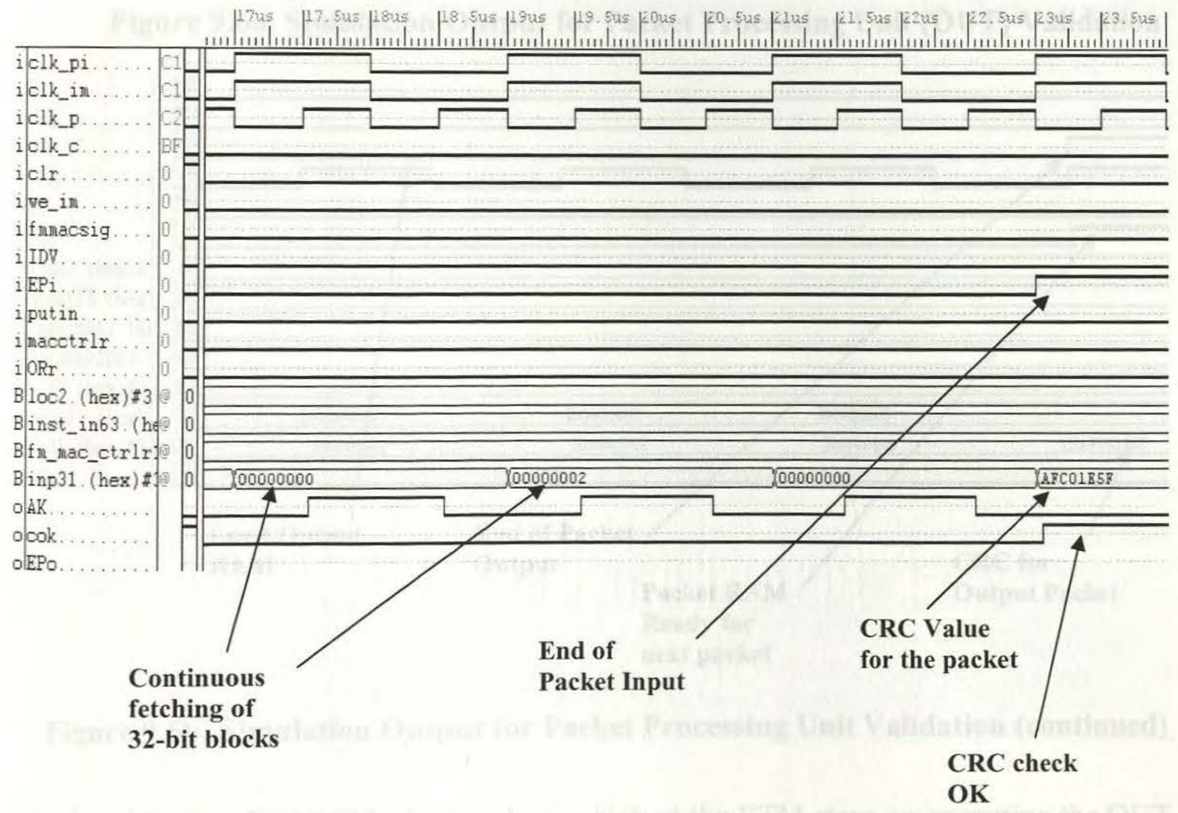


Figure 9.5b. Simulation Output for Packet Processing Unit Validation (continued)

The following Figures 9.6a and 9.6b illustrate validation of the OUT instruction of the Packet Processing unit.

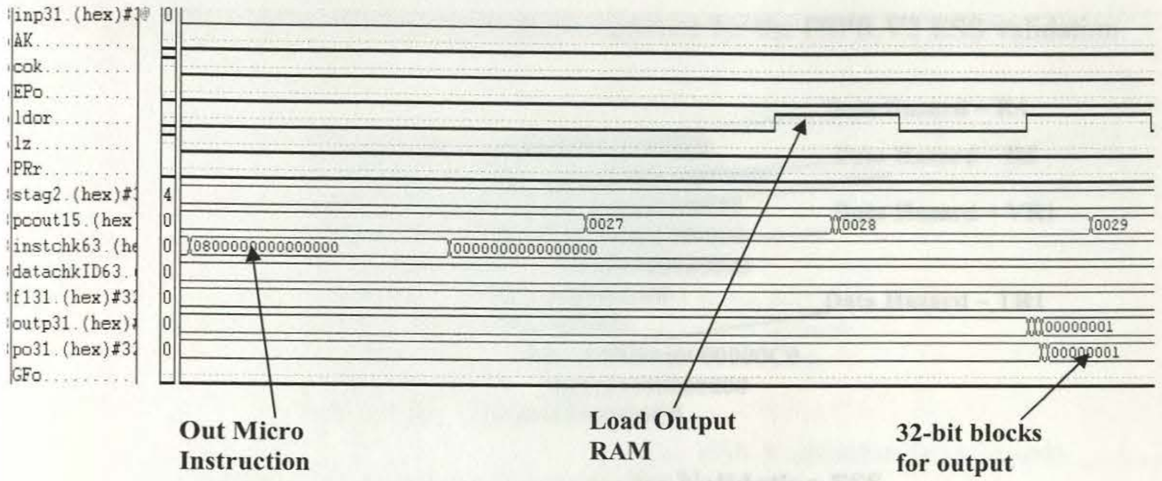


Figure 9.6a. Simulation Output for Packet Processing Unit (OUT) Validation

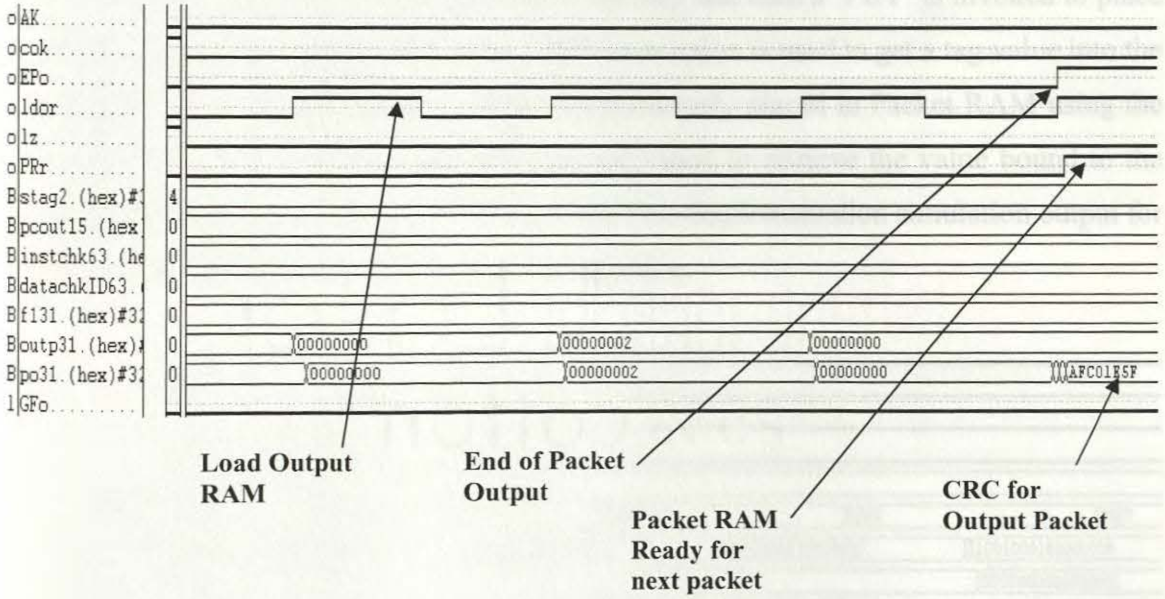
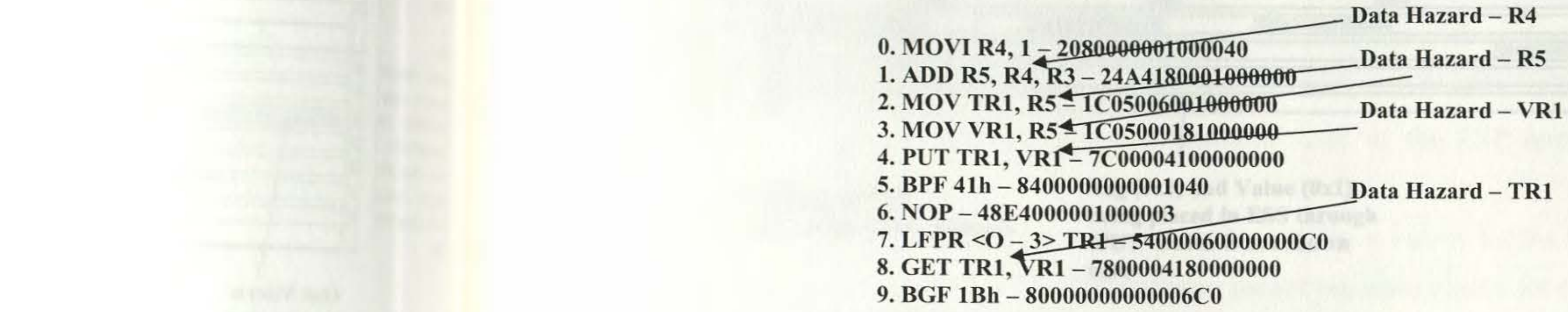


Figure 9.6b. Simulation Output for Packet Processing Unit Validation (continued)

The Load Output RAM ('ldor') signal goes high at the ETM stage on executing the OUT instruction and it goes high for each block as the packet is output in 32-bit blocks. The End of Output Packet signal goes high on receiving the CRC (final 32-bit block) of the packet and the Packet RAM Ready (PRr) signal goes high indicating the Packet RAM is ready to receive input packets.

9.2.3 Validation of ESS of ESPR.V2

Figure 9.7 shows the micro program sequence for the ESPR.V2 ESS validation.



```
0. MOVI R4, 1 – 2080000001000040
1. ADD R5, R4, R3 – 24A4180001000000
2. MOV TR1, R5 – 1C05006001000000
3. MOV VR1, R5 – 1C05000181000000
4. PUT TR1, VR1 – 7C00004100000000
5. BPF 41h – 8400000000001040
6. NOP – 48E4000001000003
7. LFPR <O – 3> TR1 – 54000060000000C0
8. GET TR1, VR1 – 7800004180000000
9. BGF 1Bh – 80000000000006C0
```

Figure 9.7. Program Sequence for Validating ESS

To get a value into the tag and value registers for performing the ‘PUT’ operation, a series of ALU operations were performed initially and then a ‘PUT’ is invoked to place a specific (tag, value) pair in ESS. The LFPR instruction is used to get a tag value into the tag register TR1 from the packet which was previously placed in Packet RAM using the IN instruction. Later a ‘GET’ operation is performed to retrieve the value bound to the tag. Figures 9.8a, 9.8b, 9.8c and 9.8d show the Post-Implementation simulation output for the ESS Validation via the above program sequence.

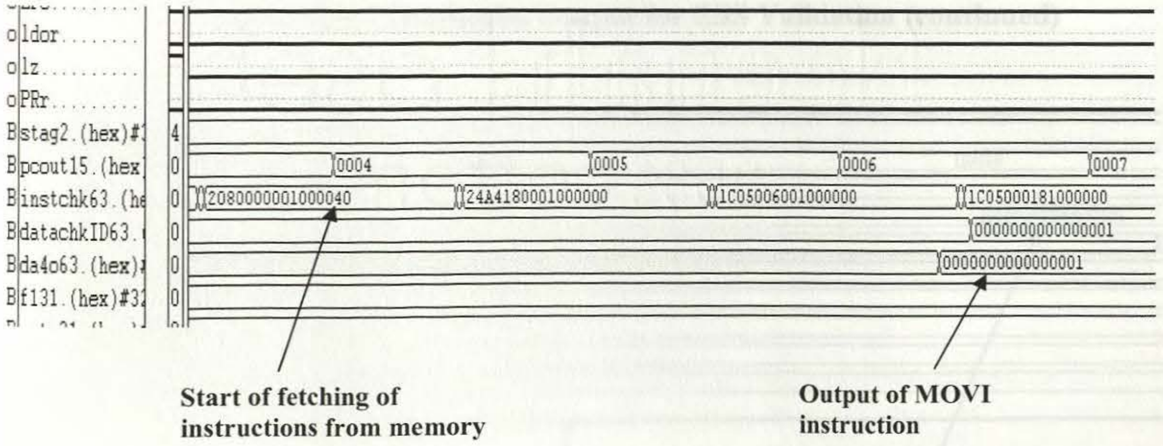
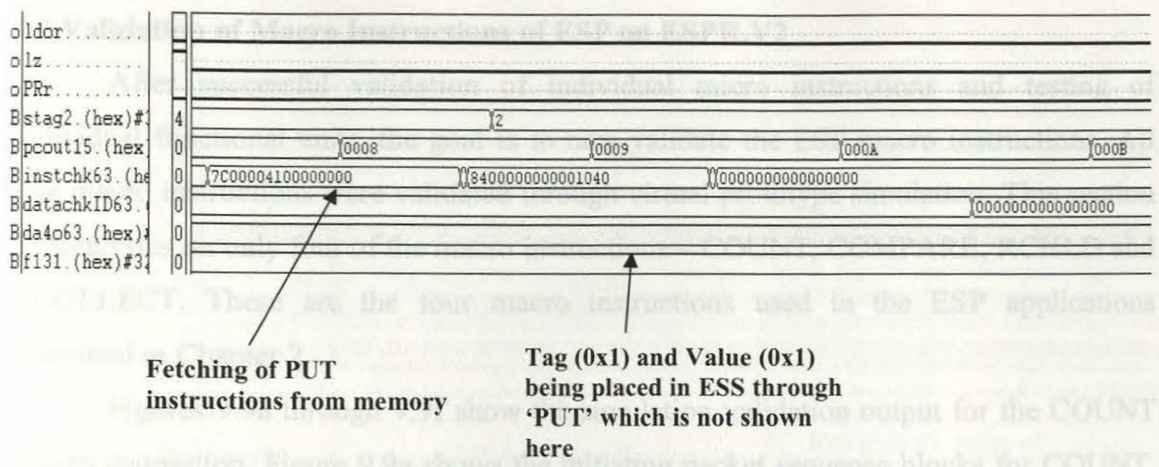


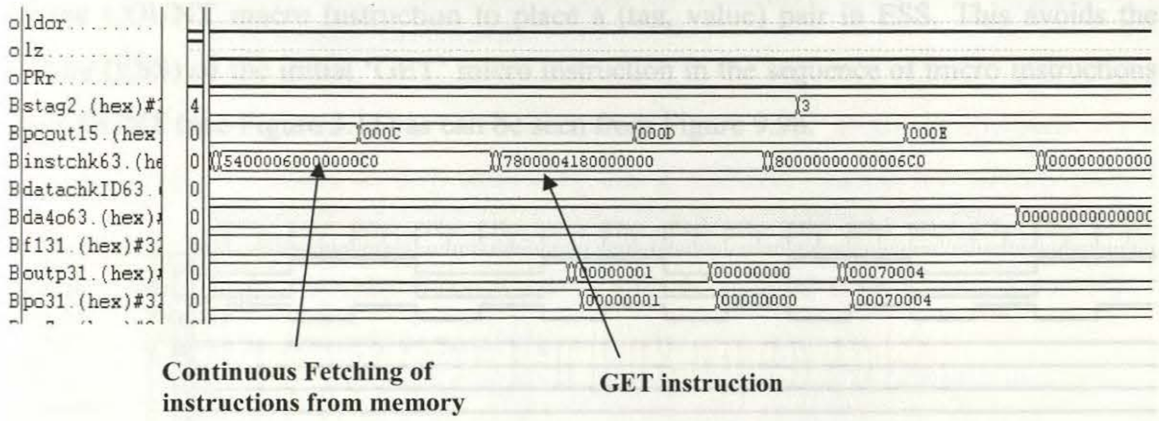
Figure 9.8a. Simulation Output for ESS Validation



Fetching of PUT instructions from memory

Tag (0x1) and Value (0x1) being placed in ESS through 'PUT' which is not shown here

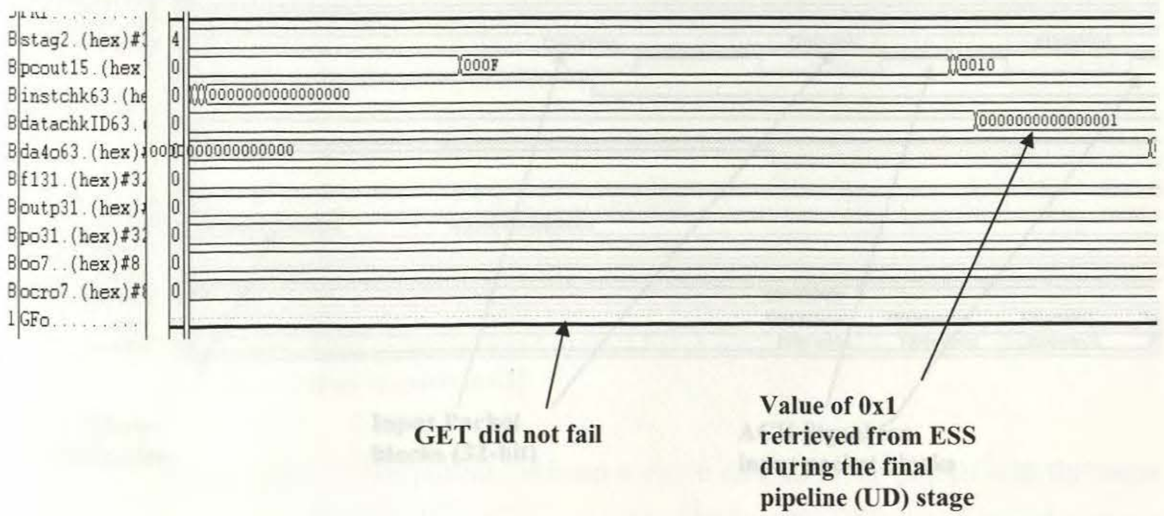
Figure 9.8b. Simulation Output for ESS Validation (continued)



Continuous Fetching of instructions from memory

GET instruction

Figure 9.8c. Simulation Output for ESS Validation (continued)



GET did not fail

Value of 0x1 retrieved from ESS during the final pipeline (UD) stage

Figure 9.8d. Simulation Output for ESS Validation (continued)

9.3 Validation of Macro Instructions of ESP on ESPR.V2

After successful validation of individual micro instructions and testing of individual functional units, the goal is to now validate the ESP macro instructions. All five macro instructions were validated through virtual prototype simulation. This section concentrates on only four of the macro instructions – COUNT, COMPARE, RCHLD and RCOLLECT. These are the four macro instructions used in the ESP applications described in Chapter 2.

Figures 9.9a through 9.9f show the simulation validation output for the COUNT Macro instruction. Figure 9.9a shows the initiating packet sequence blocks for COUNT. A different sequence of micro instructions (not shown) is executed before the execution of the COUNT macro instruction to place a (tag, value) pair in ESS. This avoids the failure (ESS) of the initial 'GET' micro instruction in the sequence of micro instructions for COUNT (see Figure 3.11) as can be seen from Figure 9.9b.

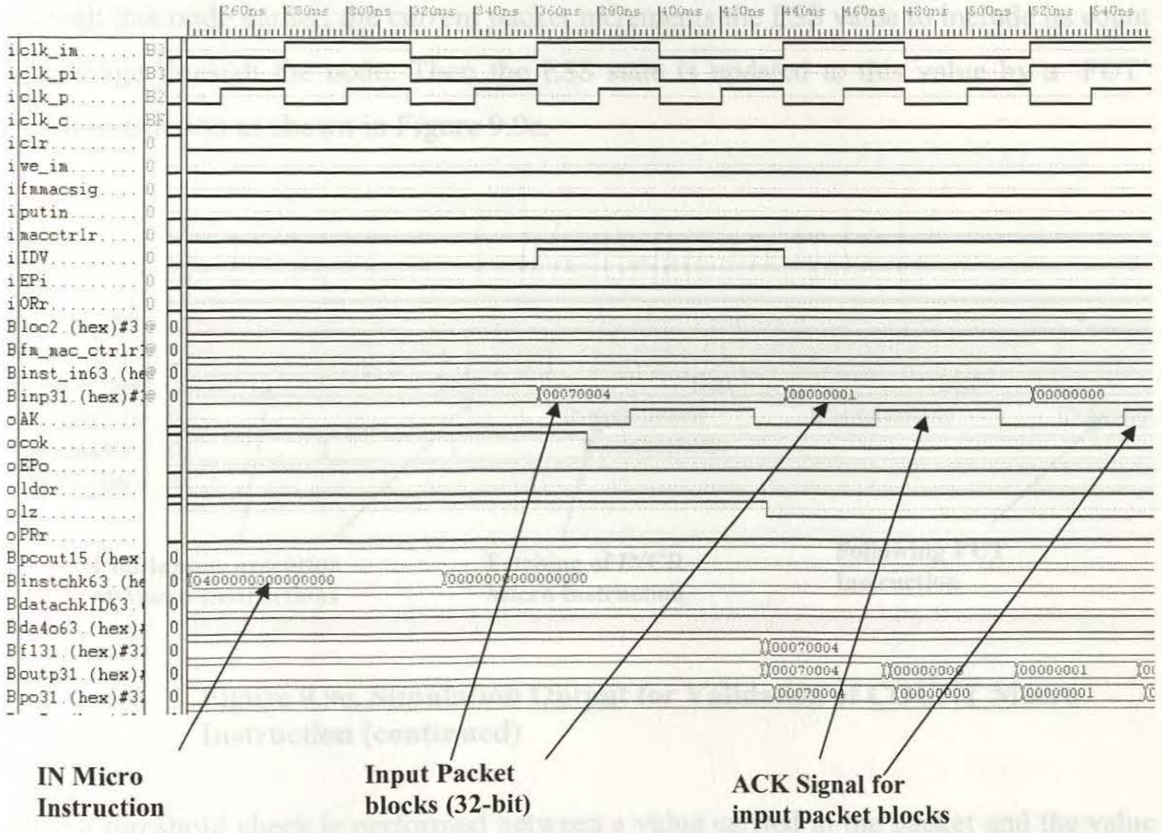


Figure 9.9a. Simulation Output for Validation of COUNT Macro Instruction

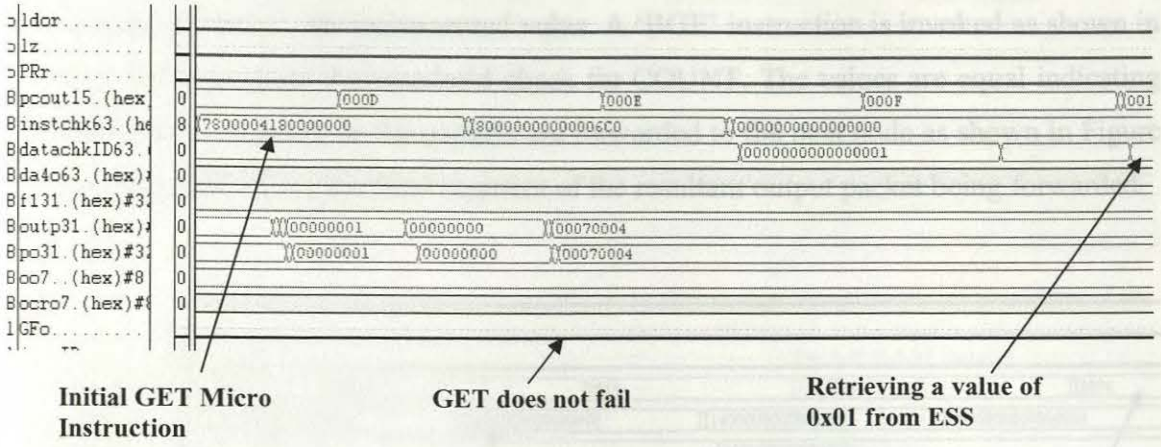


Figure 9.9b. Simulation Output for Validation of COUNT Macro Instruction (continued)

The execution continues followed by the 'INCR' and 'PUT' micro instructions. As a binding is already placed in ESS indicating that a 'COUNT' packet has already passed through this node earlier, the current packet increments the ESS value to include its count of passage through the node. Then the ESS state is updated to this value by a 'PUT' micro instruction as shown in Figure 9.9c.

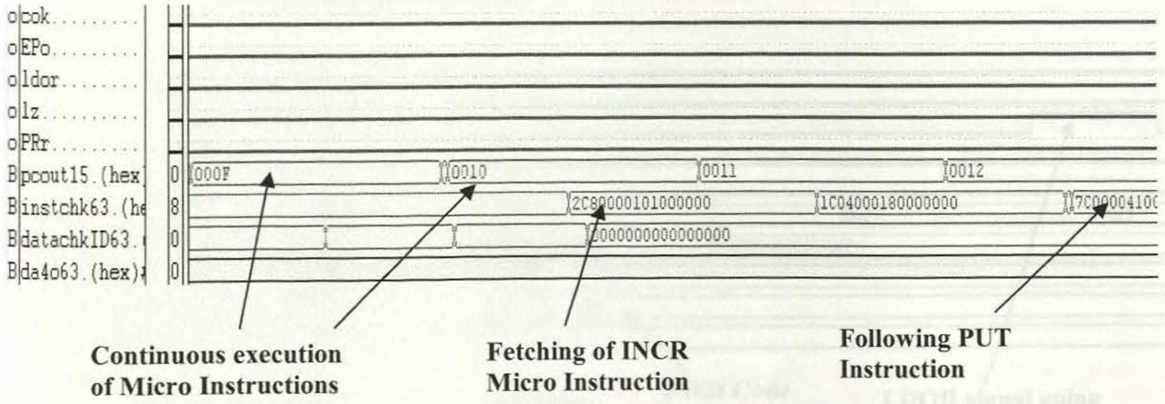


Figure 9.9c. Simulation Output for Validation of COUNT Macro Instruction (continued)

Then a threshold check is performed between a value carried in the packet and the value in the ESS. The value carried in the packet, '0x02' at offset '4' is retrieved using a

'LFPR' instruction as shown in Figure 9.9d. The current binding in the ESS for tag 'TR1' has a value of '0x02', the incremented value. A 'BGE' instruction is invoked as shown in Figure 9.9d to perform the threshold check for COUNT. The values are equal indicating the threshold is reached, so the packets are forwarded to the next node as shown in Figure 9.9e. Figure 9.9f shows the final segment of the resultant output packet being forwarded.

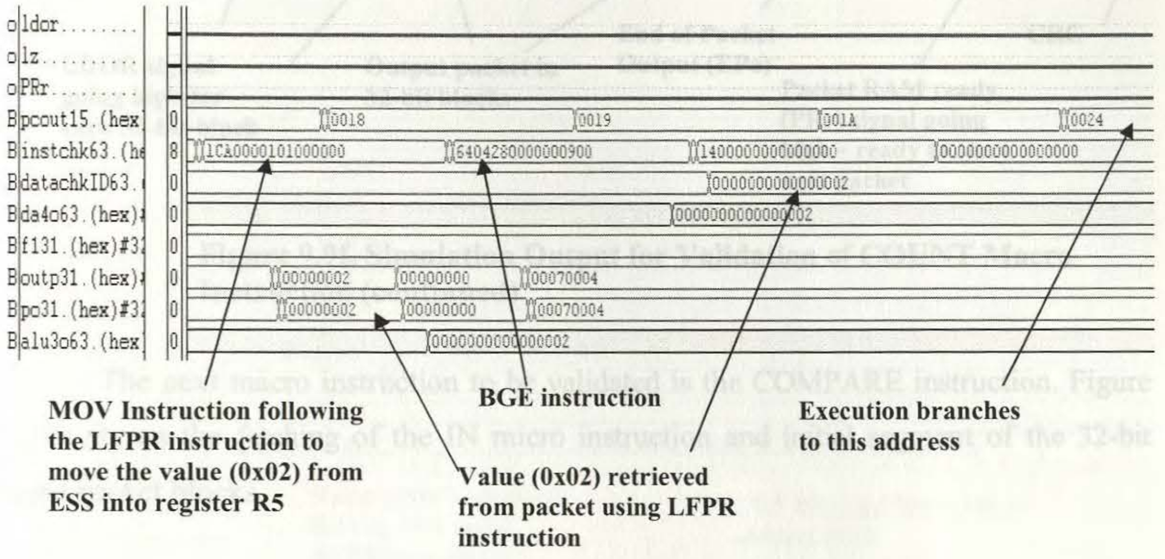


Figure 9.9d. Simulation Output for Validation of COUNT Macro Instruction (continued)

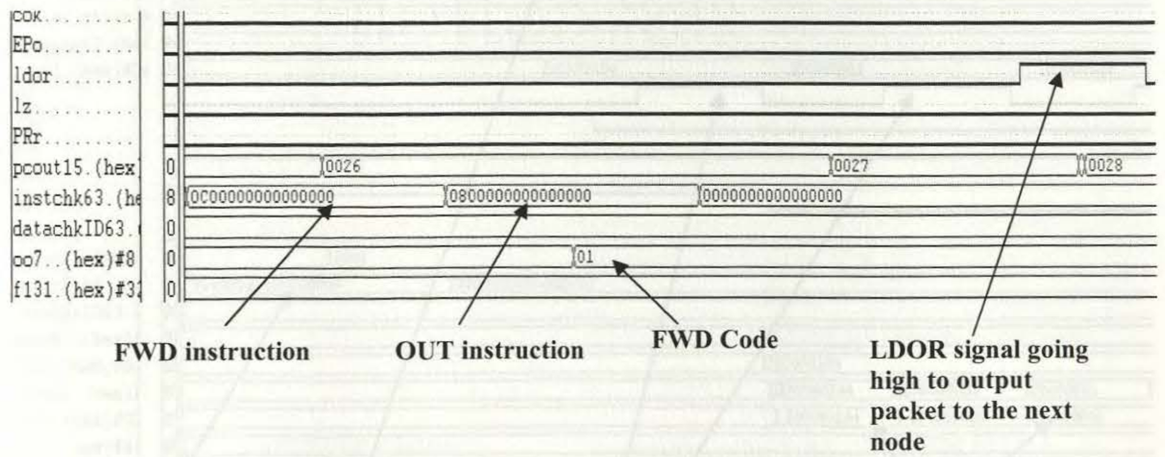


Figure 9.9e. Simulation Output for Validation of COUNT Macro Instruction (continued)

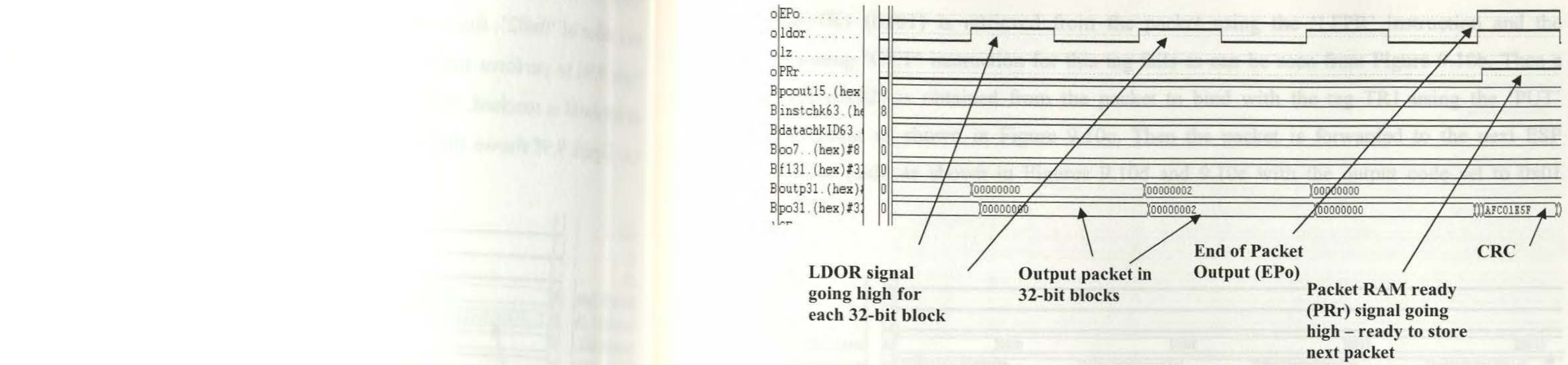


Figure 9.9f. Simulation Output for Validation of COUNT Macro Instruction (continued)

The next macro instruction to be validated is the COMPARE instruction. Figure 9.10a shows the fetching of the IN micro instruction and initial segment of the 32-bit input packet blocks.

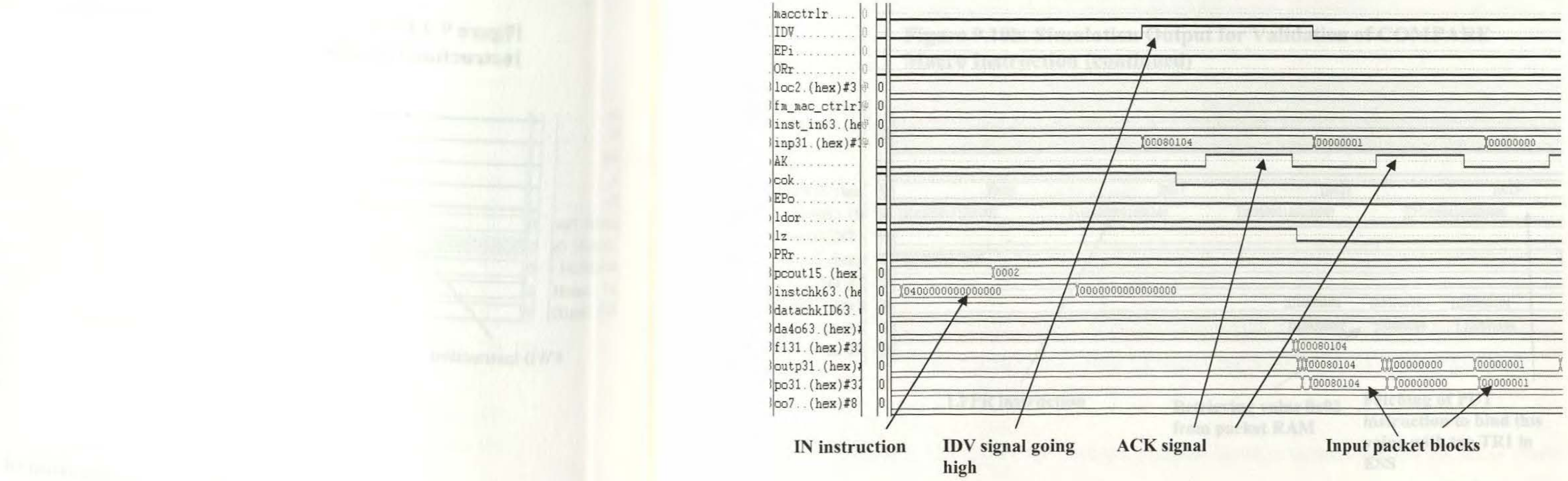
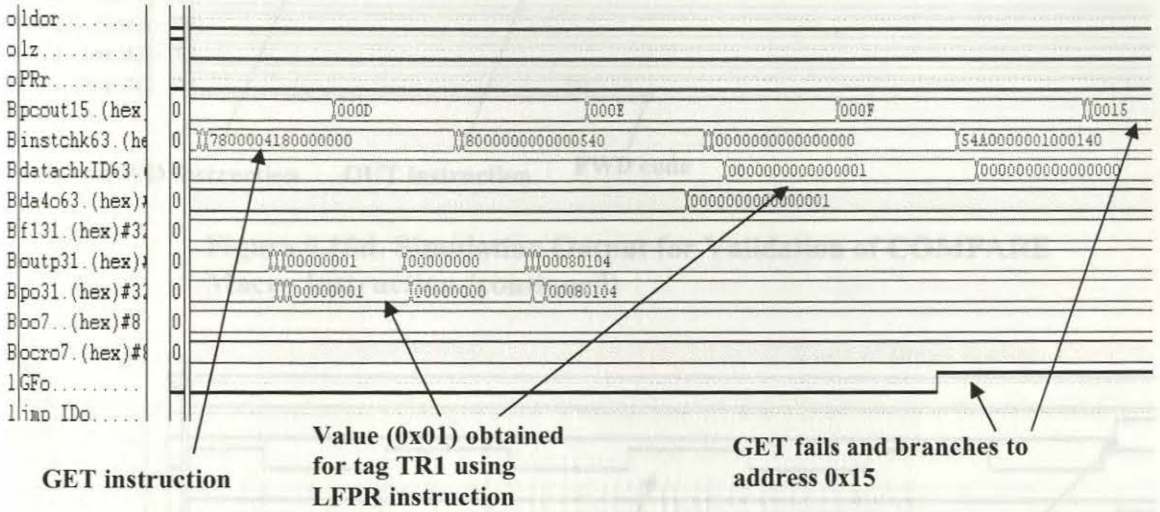


Figure 9.10a. Simulation Output for Validation of COMPARE Macro Instruction

Tag TR1 (0x01) is retrieved from the packet using the 'LFPR' instruction and the following 'GET' instruction for this tag fails as can be seen from Figure 9.10b. Then a value (0x02) is obtained from the packet to bind with the tag TR1 using the 'PUT' instruction as shown in Figure 9.10c. Then the packet is forwarded to the next ESP capable node as shown in Figures 9.10d and 9.10e with the output code set to 0x01 (FWD).

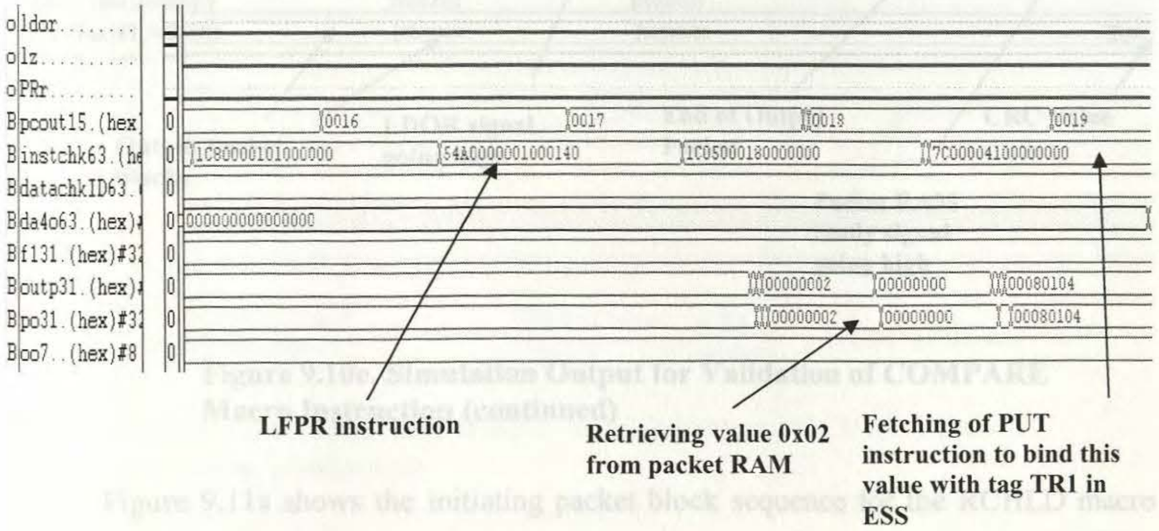


GET instruction

Value (0x01) obtained for tag TR1 using LFPR instruction

GET fails and branches to address 0x15

Figure 9.10b. Simulation Output for Validation of COMPARE Macro Instruction (continued)



LFPR instruction

Retrieving value 0x02 from packet RAM

Fetching of PUT instruction to bind this value with tag TR1 in ESS

Figure 9.10c. Simulation Output for Validation of COMPARE Macro Instruction (continued)

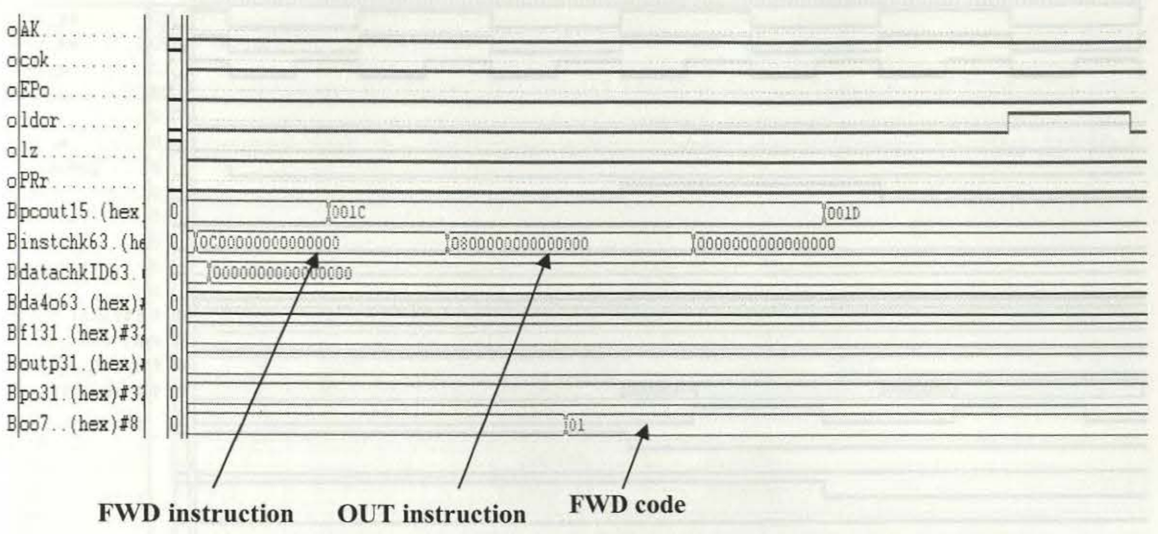


Figure 9.10d. Simulation Output for Validation of COMPARE Macro Instruction (continued)

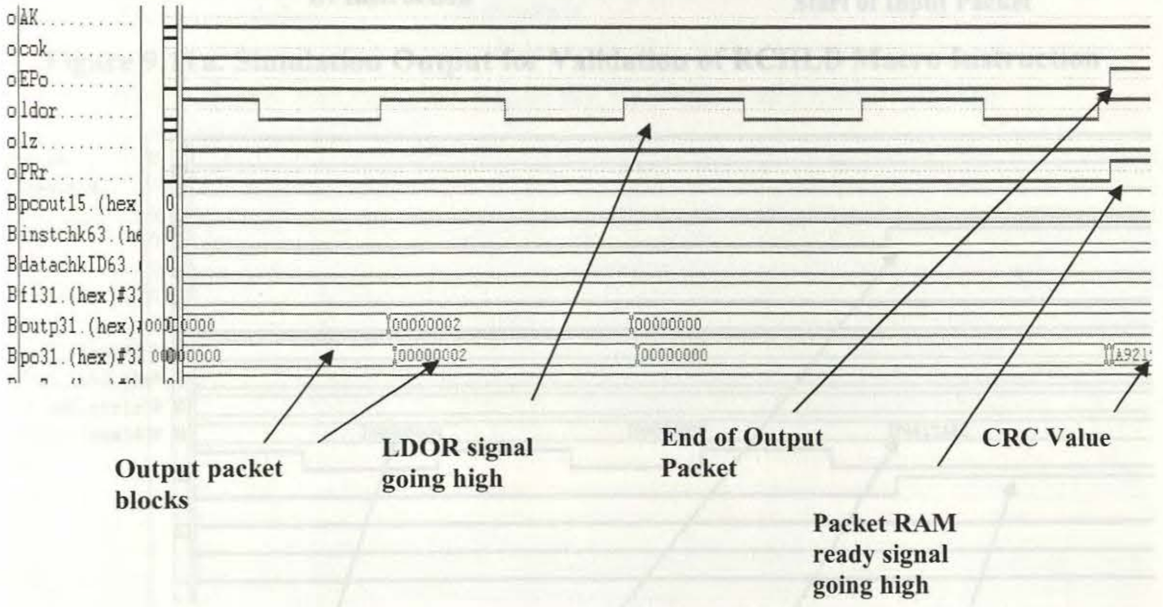


Figure 9.10e. Simulation Output for Validation of COMPARE Macro Instruction (continued)

Figure 9.11a shows the initiating packet block sequence for the RCHLD macro instruction on execution of the IN instruction and Figure 9.11b shows the ending sequence of the input packet block with the CRC.

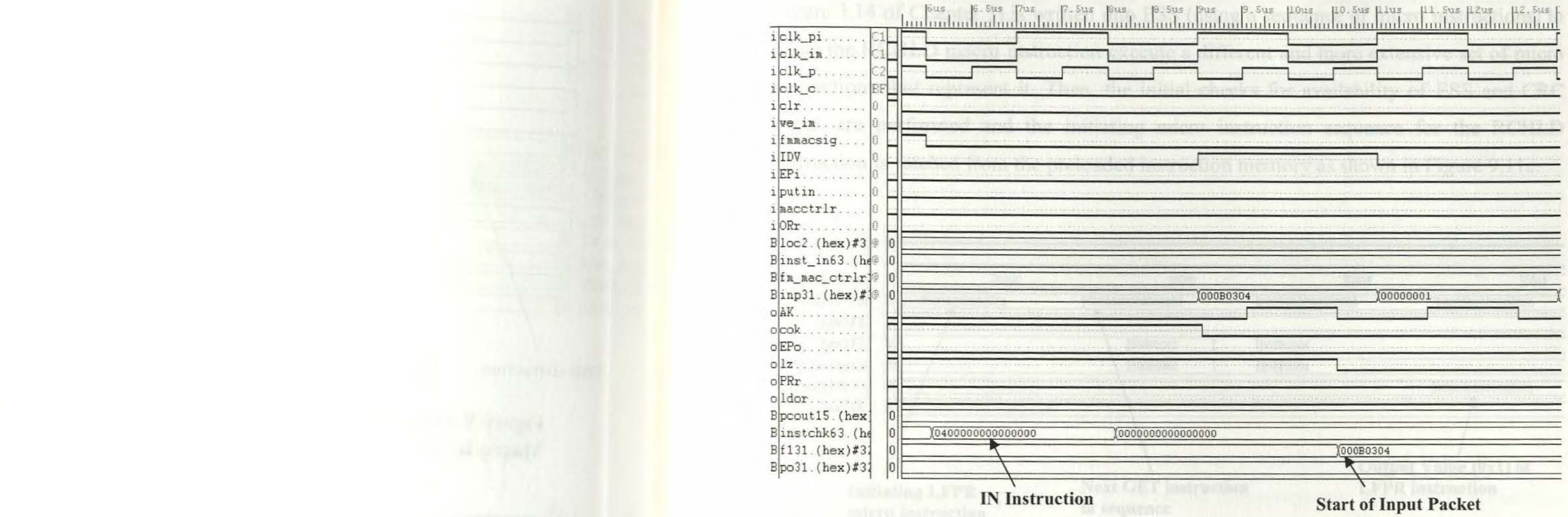


Figure 9.11a. Simulation Output for Validation of RCHLD Macro Instruction

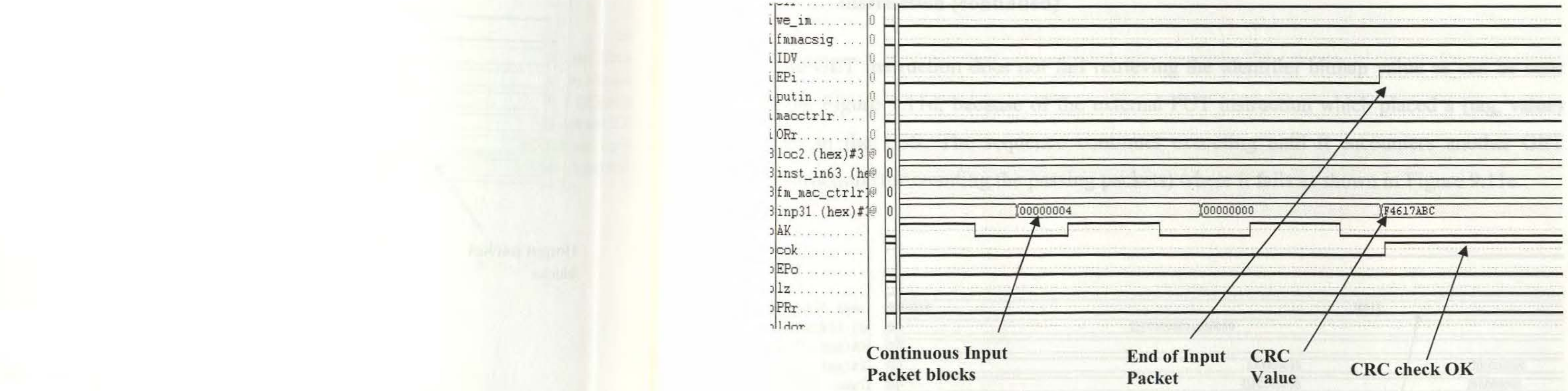


Figure 9.11b. Simulation Output for Validation of RCHLD Macro Instruction (continued)

To avoid the initial failure of the 'GET' instruction in the ESS, a value for the tag (TR2) (can be seen from the micro instruction sequence representation for 'RCHLD' from

Figure 3.14 of Chapter 3) is written into ESS (using a sequence of micro instructions) to make the RCHLD macro instruction execute a different and more extensive set of micro instructions that represent it. Then, the initial checks for availability of ESS and CRC check are performed and the initiating micro instruction sequence for the RCHLD instruction is fetched from the preloaded instruction memory as shown in Figure 9.11c.

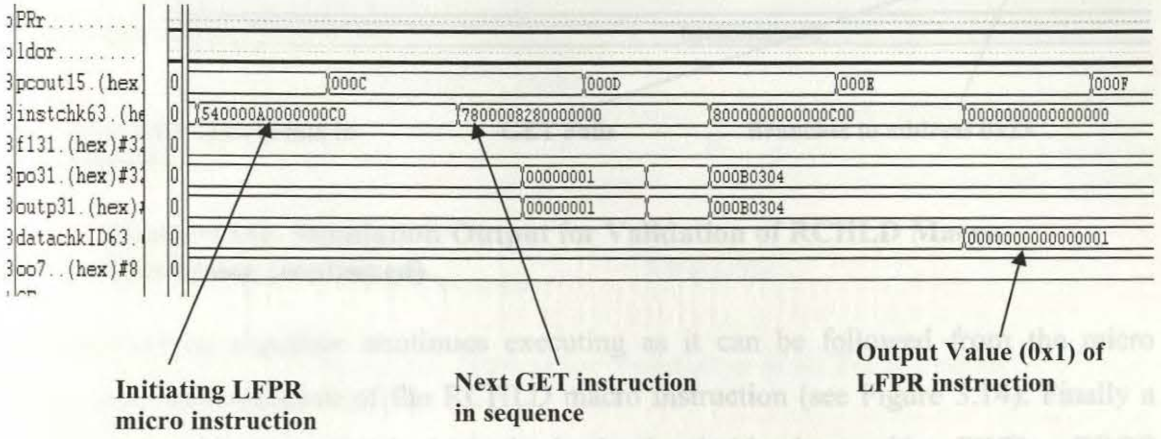


Figure 9.11c. Simulation Output for Validation of RCHLD Macro Instruction (continued)

The GET instruction does not fail retrieving the identifier bitmap value as can be seen from Figure 9.11d, because of the external PUT instruction which placed a (tag, value) pair in the ESS. The sequence continues executing until it encounters another GET instruction (for counting the passing packets) where it fails as shown in Figure 9.11e.

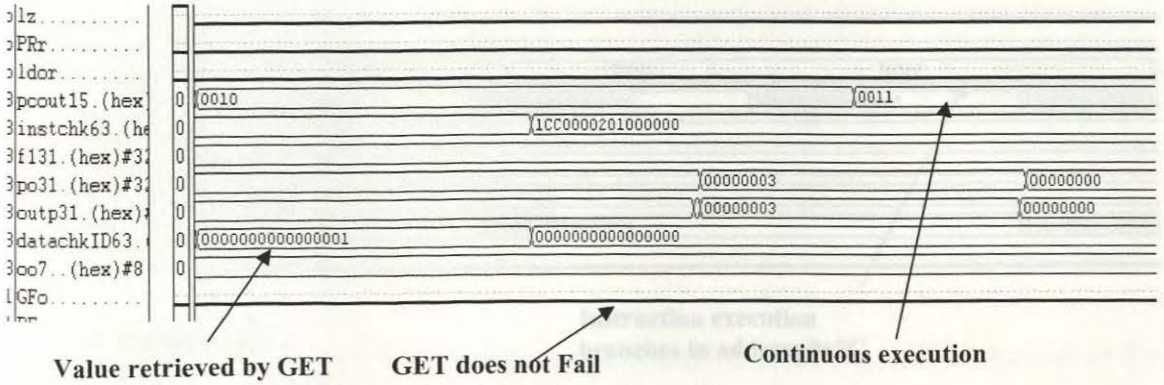
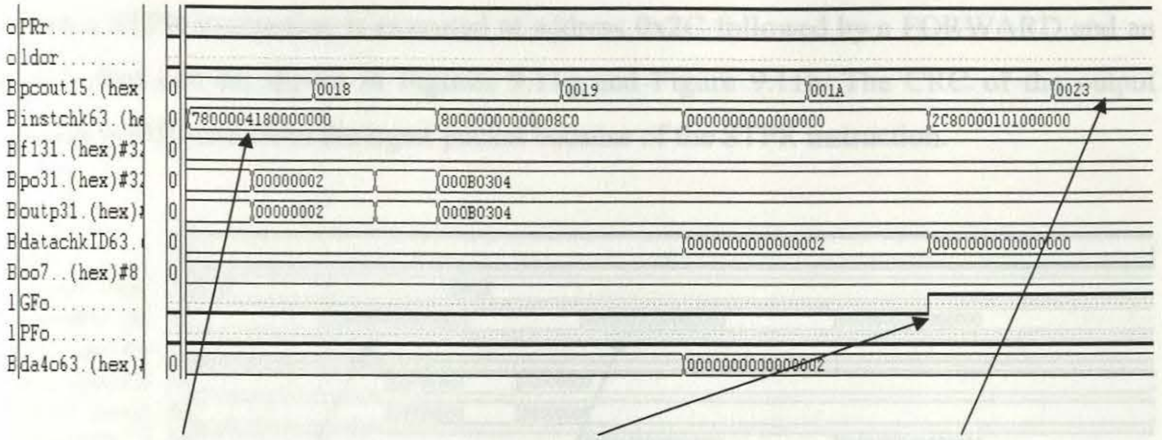


Figure 9.11d. Simulation Output for Validation of RCHLD Macro Instruction (continued)



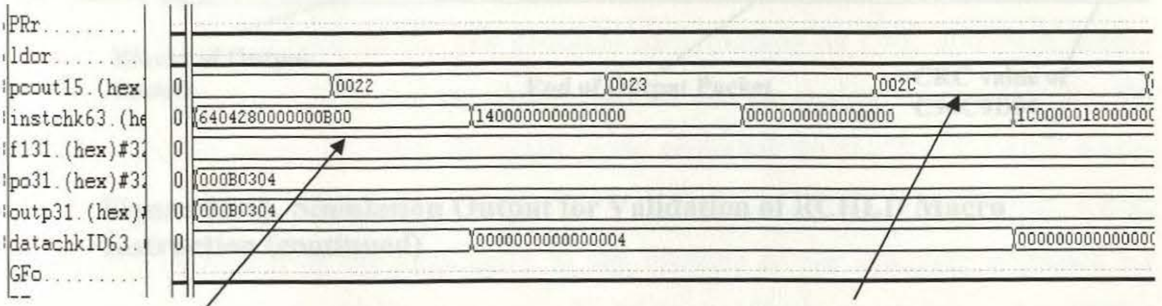
Next GET Instruction in sequence

GET Fails

Branches to address 0x23

Figure 9.11e. Simulation Output for Validation of RCHLD Macro Instruction (continued)

The instruction sequence continues executing as it can be followed from the micro instruction representation of the RCHLD macro instruction (see Figure 3.14). Finally a 'BGE' instruction is executed which checks the threshold value to either FWD or DROP the packet. The value of the input packet block at offset 0x9 is 0x4 (threshold). This value is placed in register R4 using the LFPR instruction which is not shown here. The value from register VR1 (0x1) is moved into register R5. When a 'BGE R4, R5 2Ch' instruction is executed, the value of R4 is greater than R5 indicating the threshold is not reached and the packet has to be forwarded. The instruction execution branches to address 0x2C as can be seen from Figure 9.11f.



BGE Instruction

Instruction execution branches to address 0x2C

Figure 9.11f. Simulation Output for Validation of RCHLD Macro Instruction (continued)

Then a STPR instruction is executed at address 0x2C followed by a FORWARD and an OUT, that can be shown in Figures 9.11g and Figure 9.11h. The CRC of the output packet is different from the input packet because of the STPR instruction.

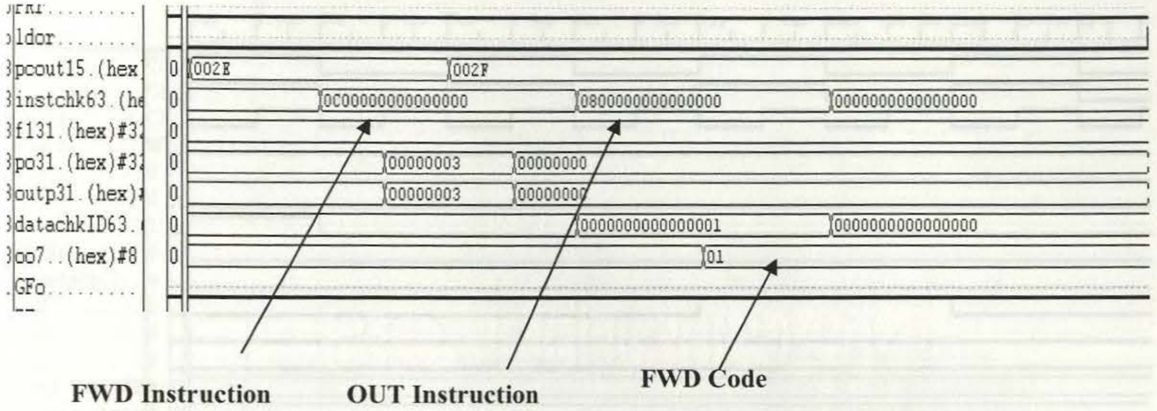


Figure 9.11g. Simulation Output for Validation of RCHLD Macro Instruction (continued)

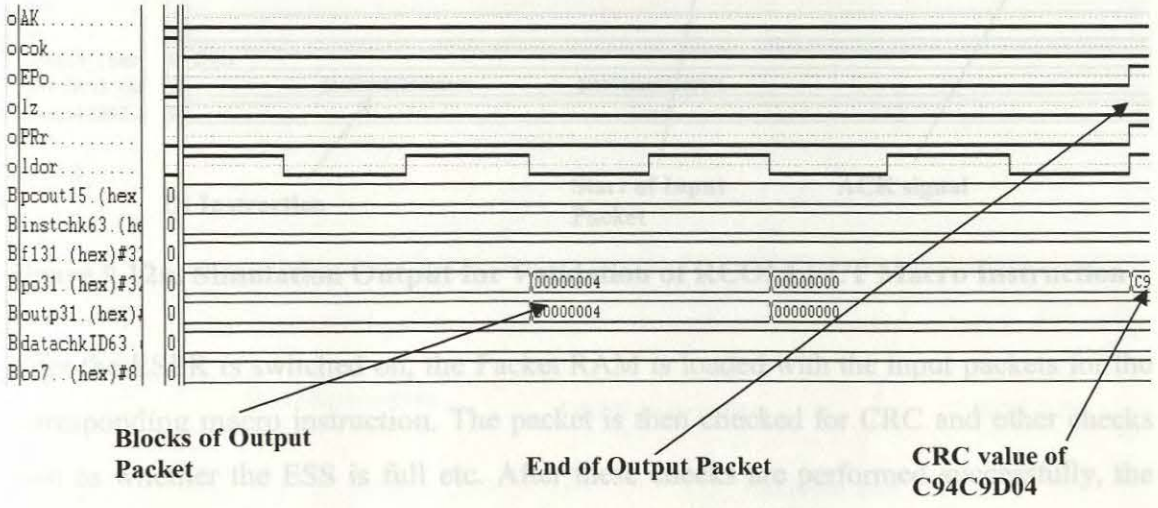


Figure 9.11h. Simulation Output for Validation of RCHLD Macro Instruction (continued)

RCOLLECT is the macro instruction which requires execution of most of the micro instructions of ESPR.V2. The following description briefly explains the Post-Implementation validation of the RCOLLECT macro instruction of ESP. Figure 9.12a shows the initial input packet for the RCOLLECT macro instruction. Figure 9.12b shows the initiating sequence of micro instructions to implement the functionality of RCOLLECT macro instruction.

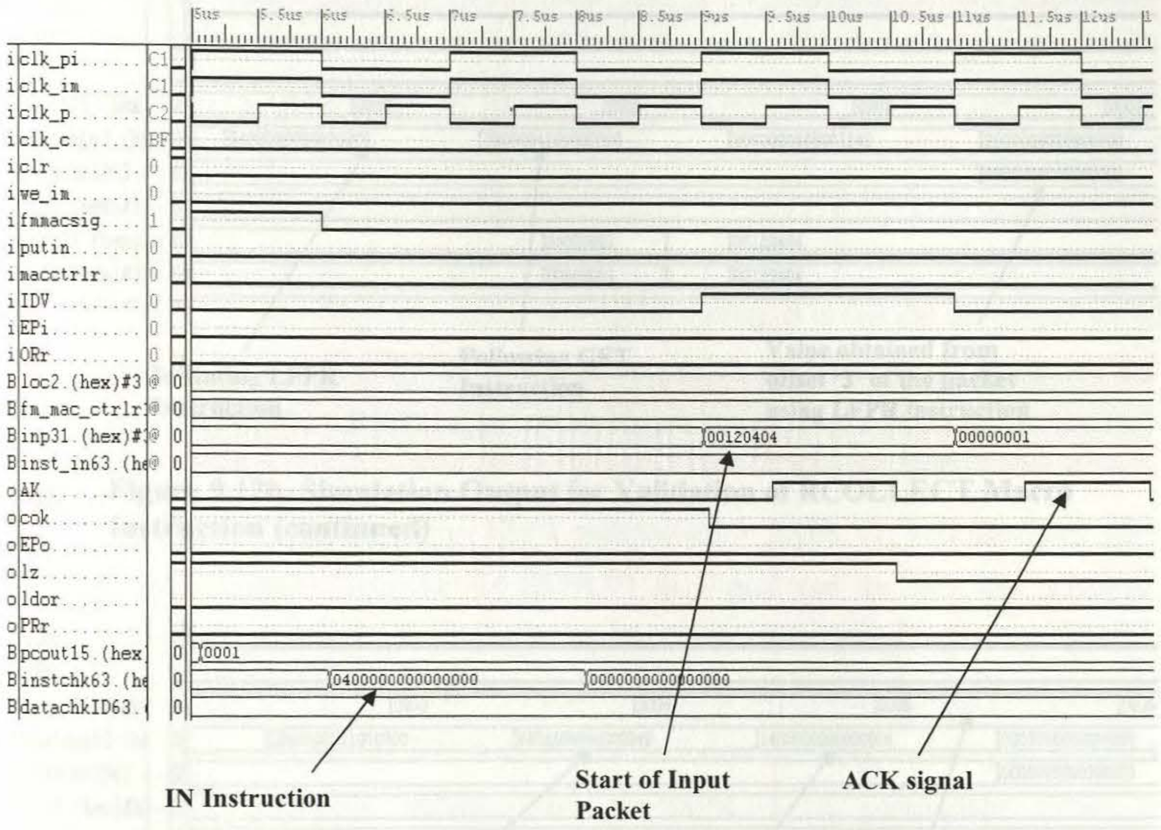


Figure 9.12a. Simulation Output for Validation of RCOLLECT Macro Instruction

After the ESPR is switched on, the Packet RAM is loaded with the input packets for the corresponding macro instruction. The packet is then checked for CRC and other checks such as whether the ESS is full etc. After these checks are performed successfully, the program counter starts fetching the micro code sequence for the RCOLLECT macro instruction as shown in Figure 9.12b. Similar to the previous RCHLD instruction, a (tag, value) pair is placed in the ESS prior to the fetching of the initiating sequence for RCOLLECT, and so the GET instruction in Figure 9.12b does not fail and continues execution from there on. The second GET fails and it executes till JMP instruction in the ADDR2 (0x26) block because R4 has a value of zero. Then it fails in the GET instruction in ADDR3 (0x1B) block and branches to ADDR5 (0x2B) block. In the ADDR5 block, the execution of 'BEQ R10, R11, ADDR7' fails because R10 has a value of 0x1 from VR1 and R11 has a value of 0x0 from VR2 and so the packet gets dropped as can be seen from Figure 9.12c.

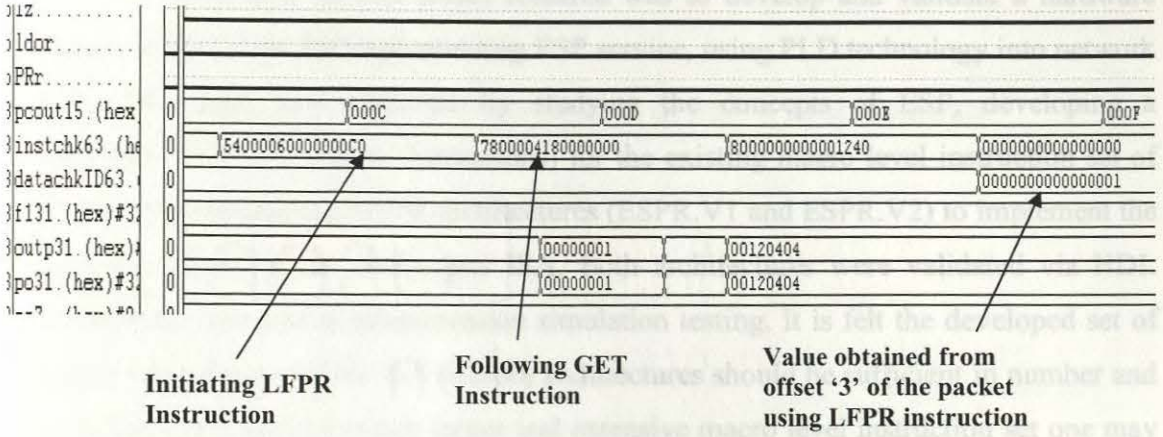


Figure 9.12b. Simulation Output for Validation of RCOLLECT Macro Instruction (continued)

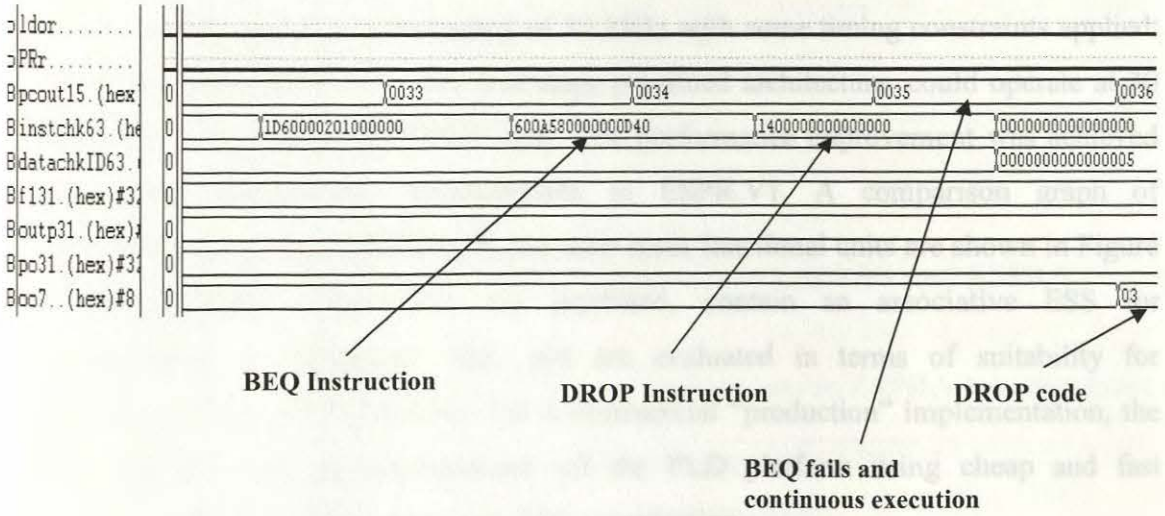


Figure 9.12c. Simulation Output for Validation of RCOLLECT Macro Instruction (continued)

Chapter Ten

Conclusions and Future Research

The main goal of this thesis research was to develop and validate a hardware processor architecture for implementing ESP service, using PLD technology into network routers. The goal was achieved by studying the concepts of ESP, developing a "lightweight ISA" (37 micro instructions) for the existing macro level instruction set of ESP, and then developing ESPR architectures (ESPR.V1 and ESPR.V2) to implement the micro-instructions of the developed ISA. Both architectures were validated via HDL post-synthesis and post-implementation simulation testing. It is felt the developed set of 37 micro-instructions of the ISA of both architectures should be sufficient in number and functionality to support a much larger and extensive macro level instruction set one may use to support ESP.

The second version of the ESPR architecture – ESPR.V2, was designed with increasing performance over that of ESPR.V1 as a goal and the aim was achieved. ESPR.V1 could operate at a frequency of 20 MHz with some timing constraints applied. On the other hand ESPR.V2 – the five-stage pipelined architecture, could operate at 30 MHz in the same technology FPGA chip. The performance improvement was achieved strictly from architectural enhancements to ESPR.V1. A comparison graph of performance of both the architectures and their main functional units are shown in Figure 10.1. Both ESPR architectures are pipelined, contain an associative ESS for storage/retrieval of ephemeral data, and are evaluated in terms of suitability for implementation to a PLD platform. For a commercial "production" implementation, the ESS probably would be implemented off the PLD platform using cheap and fast commodity memory implementing the ESS organization.

Table 10.1 gives the approximate throughput measured in packets per second (pps) obtained using the ESPR.V2 architecture through virtual prototype simulation. Since each macro instruction executes a different set of micro instructions according to the previous state in the ESS, and also, since it is not experimentally tested, the throughput results using post implementation simulation are considered to be an approximate but reliable estimate. It should also be noted that the post-implementation simulation results of Table 10.1 were achieved after implementation of the ESPR.V2 architecture to a moderate speed and older FPGA chip. The Kpps rates shown in Table 10.1 could and would be significantly increased via implementation of the ESPR.V2 architecture to a more modern and higher speed FPGA chip.

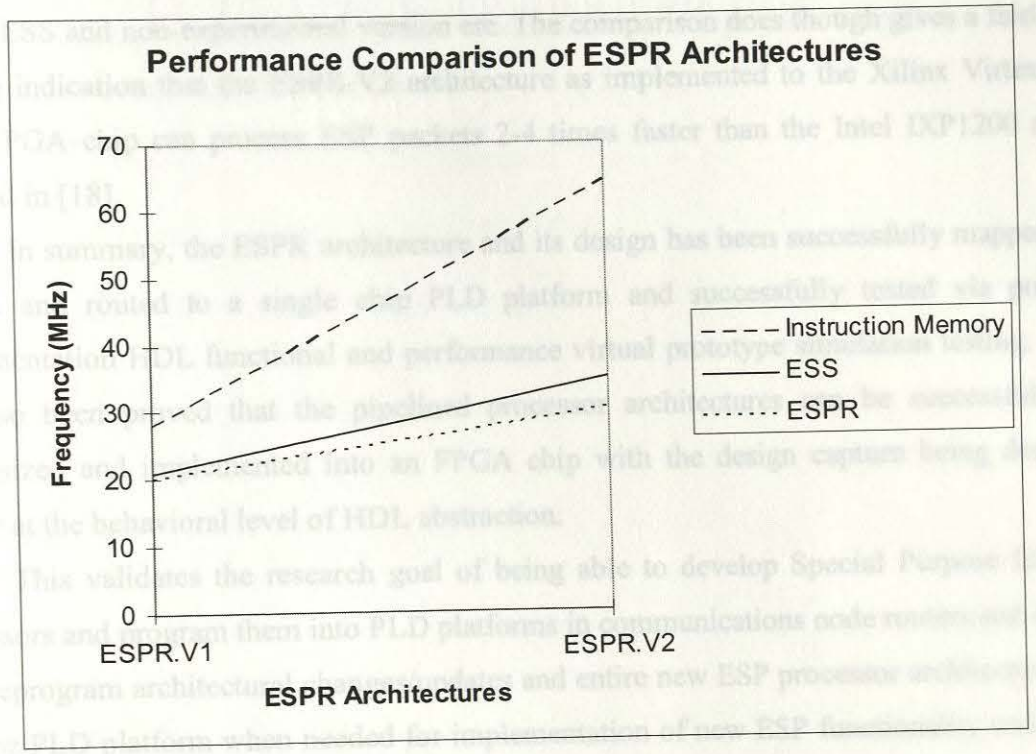


Figure 10.1. Performance Comparison of ESPR.V1 and ESPR.V2

Table 10.1. Throughput of ESP Macro Instructions in ESPR.V2 Architecture

Macro Operations	**Throughput in ESPR.V2 (Kpps) (approx.)**
COUNT ( )	810
COMPARE ( )	857
COLLECT ( )	833
RCHLD ( )	500
RCOLLECT ( )	517

The experimental results obtained using an Intel IXP1200 [18] router as stated in [8] produces an estimate of 340 Kpps and 232 Kpps for the COUNT () and COMPARE () macro instructions respectively using an SRAM implementation of ESS. The HDL simulation results obtained through post implementation simulation of ESPR.V2 cannot be directly compared to the experimental results of [8] as such, because of the issues of size of ESS and non-experimental version etc. The comparison does though gives a fairly reliable indication that the ESPR.V2 architecture as implemented to the Xilinx Virtex2 4000 FPGA chip can process ESP packets 2-4 times faster than the Intel IXP1200 as reported in [18].

In summary, the ESPR architecture and its design has been successfully mapped, placed, and routed to a single chip PLD platform and successfully tested via post implementation HDL functional and performance virtual prototype simulation testing. It has also been proved that the pipelined processor architectures can be successfully synthesized and implemented into an FPGA chip with the design capture being done mostly at the behavioral level of HDL abstraction.

This validates the research goal of being able to develop Special Purpose ESP processors and program them into PLD platforms in communications node routers and in-field reprogram architectural changes/updates and entire new ESP processor architectures into the PLD platform when needed for implementation of new ESP functionality and/or increased performance as communications line speeds increase.

Future Research can address issues such as: Experimental testing of ESP and ESPR architectures at the network level and improving the performance of ESPR architectures via deeper pipelining, using a multiple-issue superscalar or VLIW architectural concepts and via considering a single-chip packet-driven multiprocessor approach to ESP. Use of commercially available simple-pipeline-architecture GP processors can also be evaluated and compared on a cost/performance/adaptability basis to the ESP implementation approach addressed within this thesis.

Static and dynamically reconfigurable processor architectures are currently an active research area [20,21,22,23]. Unfortunately, none of these past reconfigurable architectures can directly and immediately meet our application requirements. Our current ESPR architecture could obtain a future performance boost via deeper pipelining, inclusion of one additional pipeline within a single ESPR resulting in a dual-issue ESPR architecture, and through use of the ESPR as a basic processor module in an envisioned dynamically reconfigurable single-chip multiprocessor ESPR system. This system could possibly be based upon some of the framework presented in [23,24,25,26]. It is felt some of the architectural framework of [23,24,25,26] could potentially be used to meet network node processing performance needs imposed by expected extremely high communications line speeds of the future.

Appendices

Appendix A – Presents the Micro Instruction Set Architecture and Definition for the ESPR Architectures.

Appendix B – Presents the Macro System Flowchart for ESPR.

Appendix C – Shows the Micro System Flowchart for ESPR.V1.

Appendix D – Shows the Micro System Flowchart for ESPR.V2.

Appendix E – Presents the VHDL Code for ESPR.V2.

VHDL Code for ESPR.V1 can be obtained from [28].

Appendix A

Micro Instruction Set Architecture and Definition

0. NOP (OTHER Type Instruction) – No Operation

63 58	57 0
000000	

1. IN (OTHER Type Instruction) – Input Packet to Packet Register

63 58	57 0
000001	

```
If (IDV == 1) then {
        PR ←——        Input Packet
        ACK_in ←——    1 }
} Else  wait.
```

2. OUT (OTHER Type Instruction) – Outputs the Packet to Output port and also sends Output Code Register as Output

63 58	57 0
000010	

```
If (OPRAMready == 1) then {
        Output port ←——    Packet Register
        Output Code ←——    Output Code Register
} Else   wait.
```

3. FWD (OTHER Type Instruction) – Sets Forward Code in Output Code Register to Forward the packet.

63 58	57 0
000011	

Output Code Register ←—— 1 (FWD Code)

4. ABORT1 (OTHER Type Instruction) – Sets the LOC bits to zero in packet by loading Flag Register to Flag field of Packet and the packet is forwarded.

63 58	57 0
000100	

FLR ← "00000000"
Output Code Register ← 2 (ABORT1 Code)
Flag field of PR ← Flag Register

5. DROP (OTHER Type Instruction) – Drops the packet and is indicated by setting Drop code in Output Code Register

63 58	57 0
000101	

Output Code Register ← 3 (DROP code)
Output Code ← Output Code Register

6. CLR - Clears the register RD by moving R0, which contains 0 to RD

63 58	57 53	52 48	47	24	0
000110	RD	R0		1	

RD ← R0

7. MOVE RD, RS – Move value in RS to RD

63 58	57 53	52 48	47	24	0
000111	RD	RS		1	

RD ← RS

8. MOVI RD, Imm. Val (I Type Instruction) – Move Sign Extended Immediate value to RD

63 58	57 53	52	24	23 22	22	21 6	5 0
001000	RD		1		S	16 bit Imm Val	

RD ← Sign Extended Imm.val

9.ADD RD, RS1, RS2 (ALU Type Instruction) – Adds RS1 and RS2 and places the result in RD

63 58	57 53	52 48	47 43	42	24	0
001001	RD	RS1	RS2		1	

RD ← RS1 + RS2

10.SUB RD, RS1, RS2 (ALU Type Instruction) – Subtracts RS2 from RS1 and places the result in RD

63 58	57 53	52 48	47 43	42	24	0
001010	RD	RS1	RS2		1	

RD ← RS1 - RS2

11. INCR RS (ALU Type Instruction) – Increments RS by adding it with R1, which contains 1 and places the result in RD

63 58	57 53	52 48	47 43	42	24	0
001011	RS	RS	R1		1	

RS ← RS + R1

12. DECR RS (ALU Type Instruction) – Decrements RS by subtracting R1 from RS and places the result in RD

63 58	57 53	52 48	47 43	42	24	0
001100	RS	RS	R1		1	

RS ← RS - R1

13. OR RD, RS1, RS2 (ALU Type Instruction) – Logical OR of RS1 and RS2 and places result in RD

63 58	57 53	52 48	47 43	42	24	0
001101	RD	RS1	RS2		1	

RD ← RS1 (OR) RS2

14. AND RD, RS1, RS2 (ALU Type Instruction) – Logical AND of RS1 and RS2 and places result in RD

63 58	57 53	52 48	47 43	42	24	0
001110	RD	RS1	RS2		1	

RD ← RS1 (AND) RS2

15. EXOR RD, RS1, RS2 (ALU Type Instruction) – Logical EXOR of RS1 and RS2 and places result in RD

63 58	57 53	52 48	47 43	42	24	0
001111	RD	RS1	RS2		1	

RD ← RS1 (EXOR) RS2

16. COMP RD, RS (ALU Type Instruction) – Logical NOT of RS and place result in RD

63	58 57	53 52	48 47	24	0
010000	RD	RS		1	

RD ← (NOT) RS

17. SHL RD, RS, SHAMT (SHIFT Type Instruction) – Logical shift left of RS by SHAMT and result is placed in RD

63	58 57	53 52	48 47	24	0
010001	RD	RS		1	SHAMT

RD ← RS << SHAMT (Default shift by 1)

18. SHR RD, RS, SHAMT (SHIFT Type Instruction) – Logical shift right of RS by SHAMT and result is placed in RD

63	58 57	53 52	48 47	24	0
010010	RD	RS		1	SHAMT

RD ← RS >> SHAMT (Default shift by 1)

19. ROL RD, RS, SHAMT (SHIFT Type Instruction) – Logical rotate left of RS by SHAMT and result is placed in RD

63	58 57	53 52	48 47	24	0
010011	RD	RS		1	SHAMT

RD ← $\overleftarrow{\text{RS << SHAMT}}$

20. ROR RD, RS, SHAMT (SHIFT Type Instruction) – Logical rotate right of RS by SHAMT and result is placed in RD

63	58 57	53 52	48 47	24	0
010100	RD	RS		1	SHAMT

RD ← $\overrightarrow{\text{RS >> SHAMT}}$

21. LFPR <Offset> RD (LFPR / STPR Type Instruction) – Loads 64 bit value at a given offset from Packet Register (PR) to RD

63	58 57	53 52	24	22 21	6 5	0
010101	RD		1		16 bit Offset	

RD ← PR[Offset] to PR[Offset + 63]

22. STPR <Offset> RS (LFPR / STPR Type Instruction) – Stores 64 bit value at a given offset in Packet Register (PR) from RS

63 58	57 53	52 48	47 22	21 6	5 0
010110		RS		16 bit Offset	

PR[Offset] to PR[Offset + 63] ← RS

23. BRNE RS1, RS2, Addr (JUMP / BRANCH Type Instruction) – Checks if RS1 not equal to RS2; if yes, execution branches to sequence of instructions starting at Br. Addr by placing Br. Addr in PC, else PC is incremented and resumes execution of normal sequence of instructions.

63 58	57 53	52 48	47 43	42 22	21 6	5 0
010111		RS1	RS2		16 bit Br. Addr	

```
IF      RS1 != RS2     then
PC      ←              Br. Addr
ELSE    PC    ←        PC + 1
```

24. BREQ RS1, RS2, Addr (JUMP / BRANCH Type Instruction) – Checks if RS1 equal to RS2; if yes, execution branches to sequence of instructions starting at Br. Addr by placing Br. Addr in PC, else PC is incremented and resumes execution of normal sequence of instructions.

63 58	57 53	52 48	47 43	42 22	21 6	5 0
011000		RS1	RS2		16 bit Br. Addr	

```
IF      RS1 == RS2     then
PC      ←              Br. Addr
ELSE    PC    ←        PC + 1
```

25. BRGE RS1, RS2, Addr (JUMP / BRANCH Type Instruction) – Checks if RS1 greater than or equal to RS2; if yes, execution branches to sequence of instructions starting at Br. Addr by placing Br. Addr in PC, else PC is incremented and resumes execution of normal sequence of instructions.

63 58	57 53	52 48	47 43	42 22	21 6	5 0
011001		RS1	RS2		16 bit Br. Addr	

```
IF      RS1 >= RS2     then
PC      ←       Br. Addr
ELSE    PC    ←        PC + 1
```

26. BNEZ RS, Addr (JUMP / BRANCH Type Instruction) – Checks if RS1 not equal to R0 (0); if yes, execution branches to sequence of instructions starting at Br. Addr by placing Br. Addr in PC, else PC is incremented and resumes execution of normal sequence of instructions.

63 58	57 53	52 48	47 43	42 22	21 6	5 0
011010		RS	R0		16 bit Br. Addr	

```
IF      RS != R0      then
PC      ←            Br. Addr
ELSE    PC     ←      PC + 1
```

27. BEQZ RS, Addr (JUMP / BRANCH Type Instruction) – Checks if RS1 equal to R0 (0); if yes, execution branches to sequence of instructions starting at Br. Addr by placing Br. Addr in PC, else PC is incremented and resumes execution of normal sequence of instructions.

63 58	57 53	52 48	47 43	42 22	21 6	5 0
011011		RS	R0		16 bit Br. Addr	

```
IF      RS == R0      then
PC      ←            Br. Addr
ELSE    PC     ←      PC + 1
```

28. JMP Addr (JUMP / BRANCH Type Instruction) – Jumps to a location specified by Br. Addr by placing Br. Addr in PC

63 58	57 22	21 6	5 0
011100		16 bit Br. Addr	

```
PC      ←       Br. Addr
```

29. RET (JUMP / BRANCH Type Instruction) – Returns from execution of a subroutine to normal sequence execution by placing Reg in PC.

63 58	57 0
011101	

```
PC      ←       Reg
```

30. GET VR, TR (GET / PUT TYPE INSTRUCTION) – Gets Value in VR Corresponding to Tag TR and Sets CCR as GF = 1, for Failure of GET operation.

011110		TR	VR	

```
Tag and Value given to ESS
If match found: then,
        If Lifetime not expired then,
                VR  ←       Value
                GF  ←       0
        Else
                GF  ←       1, VR ←  0
                Clean that location and sets Empty (E) bit to 1
Else
                GF  ←       1, VR ←  0
```

31. PUT TR, VR (GET / PUT TYPE INSTRUCTION) – Puts Tag and Value (creates a tag, value binding) in ESS by placing tag from TR and value from VR into ESS. Sets CCR as PF = 1, for failure of PUT operation

011111		TR	VR	

```
Tag and Value given to ESS
If match found: then,
        If Lifetime not expired then,
                Value  ←  VR
        Else
                Tag    ←  TR
                Value  ←  VR
                Reset Expiration Time
Else
        If Empty Location then,
                Tag    ←  TR
                Value  ←  VR
                Store Expiration Time
                Empty bit  ←  0
        Else
                PF     ←  1
```

32. BGF Addr (GET / PUT TYPE INSTRUCTION) – Checks the Condition Code Register (CCR) for failure of GET operation. If GF is 1 indicating failure of GET, execution branches to sequence of instructions starting at Br. Addr by placing Br. Addr in PC, else PC is incremented and resumes execution of normal sequence of instructions.

100000		16 bit Br. Addr	

```
If GF == 1 then PC  ←  Br. Addr
Else    PC  ←  PC + 1
```

33. BPF Addr (GET / PUT TYPE INSTRUCTION) – Checks the Condition Code Register (CCR) for failure of PUT operation. If PF is 1 indicating failure of PUT, execution branches to sequence of instructions starting at Br. Addr by placing Br. Addr in PC, else PC is incremented and resumes execution of normal sequence of instructions.

100001		16 bit Br. Addr	

```
If PF == 1 then PC  ←  Br. Addr
Else    PC  ←  PC + 1
```

34. ABORT2 (OTHER Type Instruction) – Sets the LOC bits to zero and E bit to ‘1’ in packet by loading Flag Register to Flag field of Packet and the packet is forwarded.

63 58	57 0
100010	

FLR ← “00000001”
Output Code Register ← 4 (ABORT2 Code)
Flag field of PR ← Flag Register

35. BLT RS1, RS2, Addr (JUMP / BRANCH Type Instruction) – Checks if RS1 is less than RS2; if yes, execution branches to sequence of instructions starting at Br. Addr by placing Br. Addr in PC, else PC is incremented and resumes execution of normal sequence of instructions.

63 58	57 53	52 48	47 43	42 22	21 6	5 0
100011		RS1	RS2		16 bit Br. Addr	

```
IF      RS1 < RS2      then
PC      ←       Br. Addr
ELSE    PC      ←       PC + 1
```

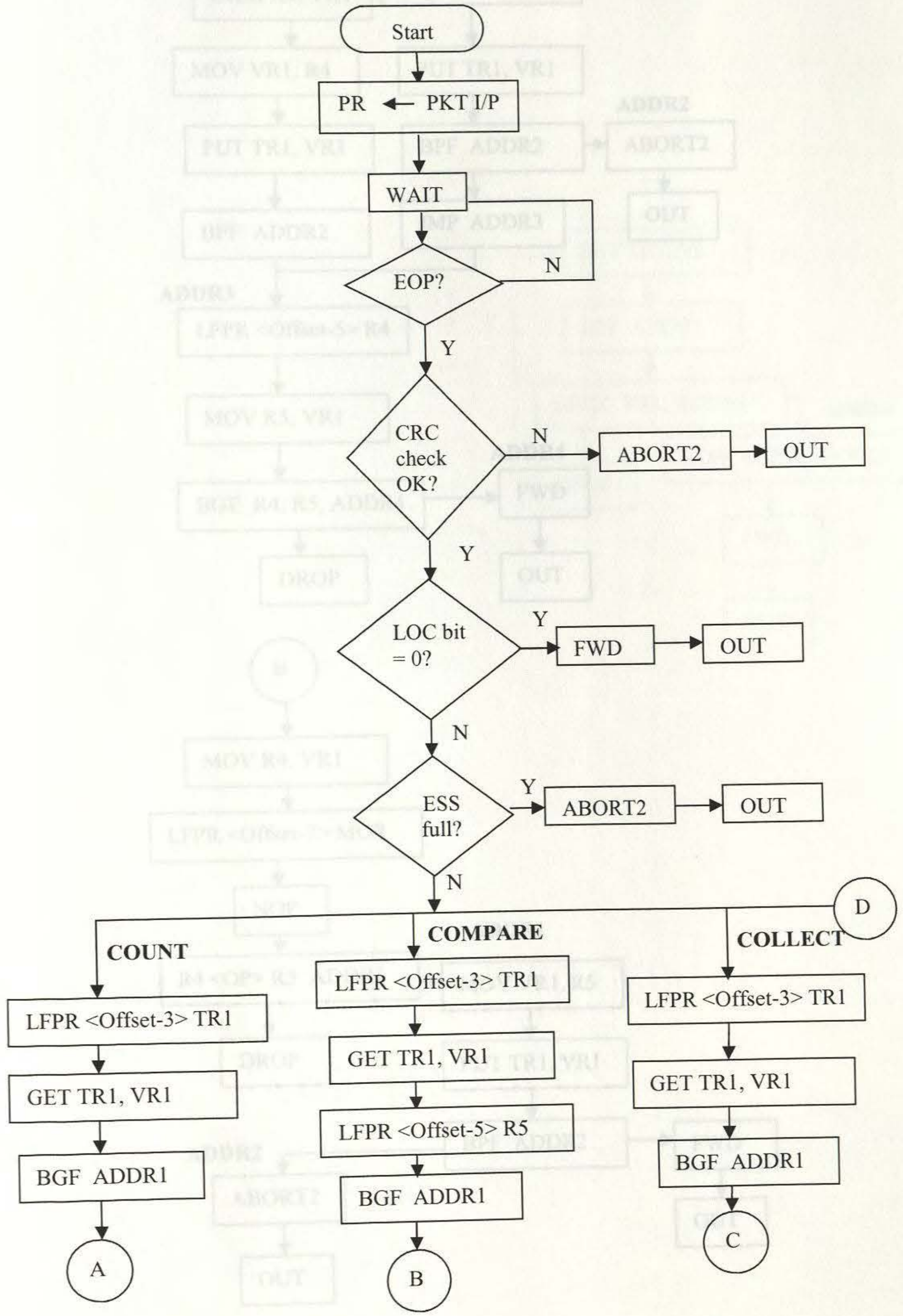
36. SETLOC (OTHER Type Instruction) – Sets the LOC bits in packet to a specified given value.

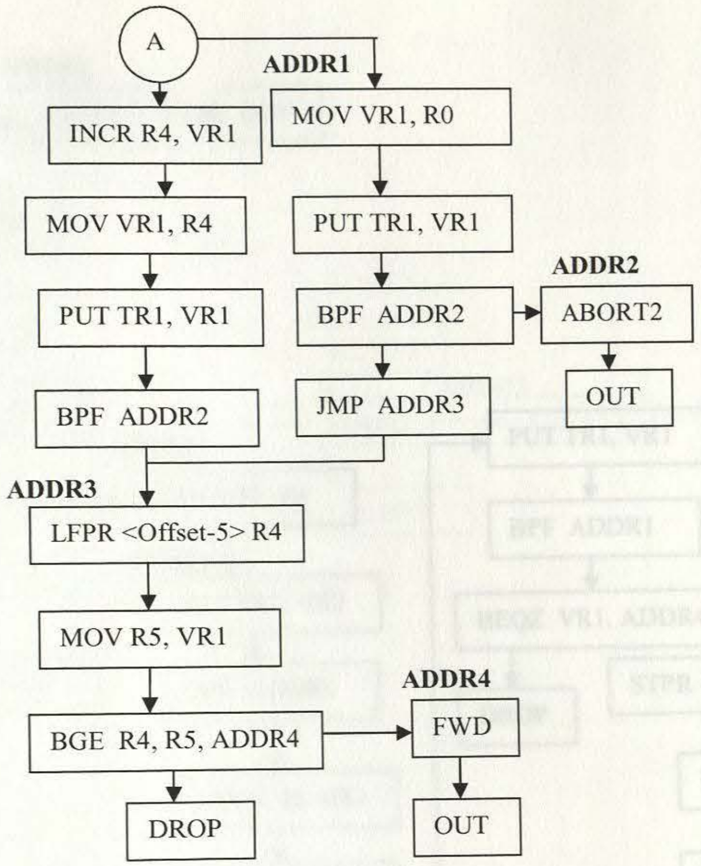
63 58	57 0
100100	

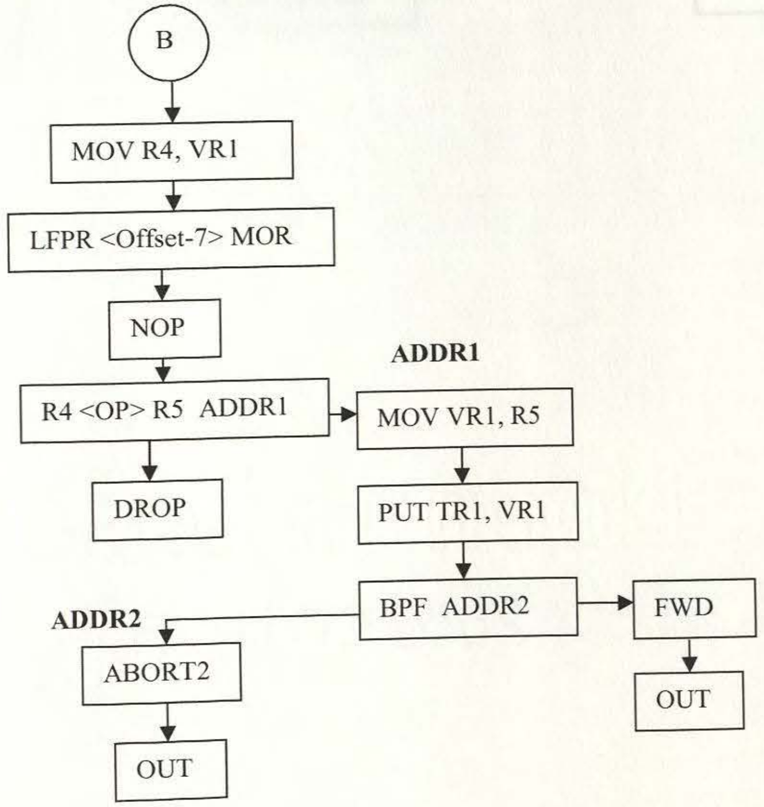
FLR (7 downto 5) ← Given LOC Value (3 bits)
Flag field of PR ← Flag Register

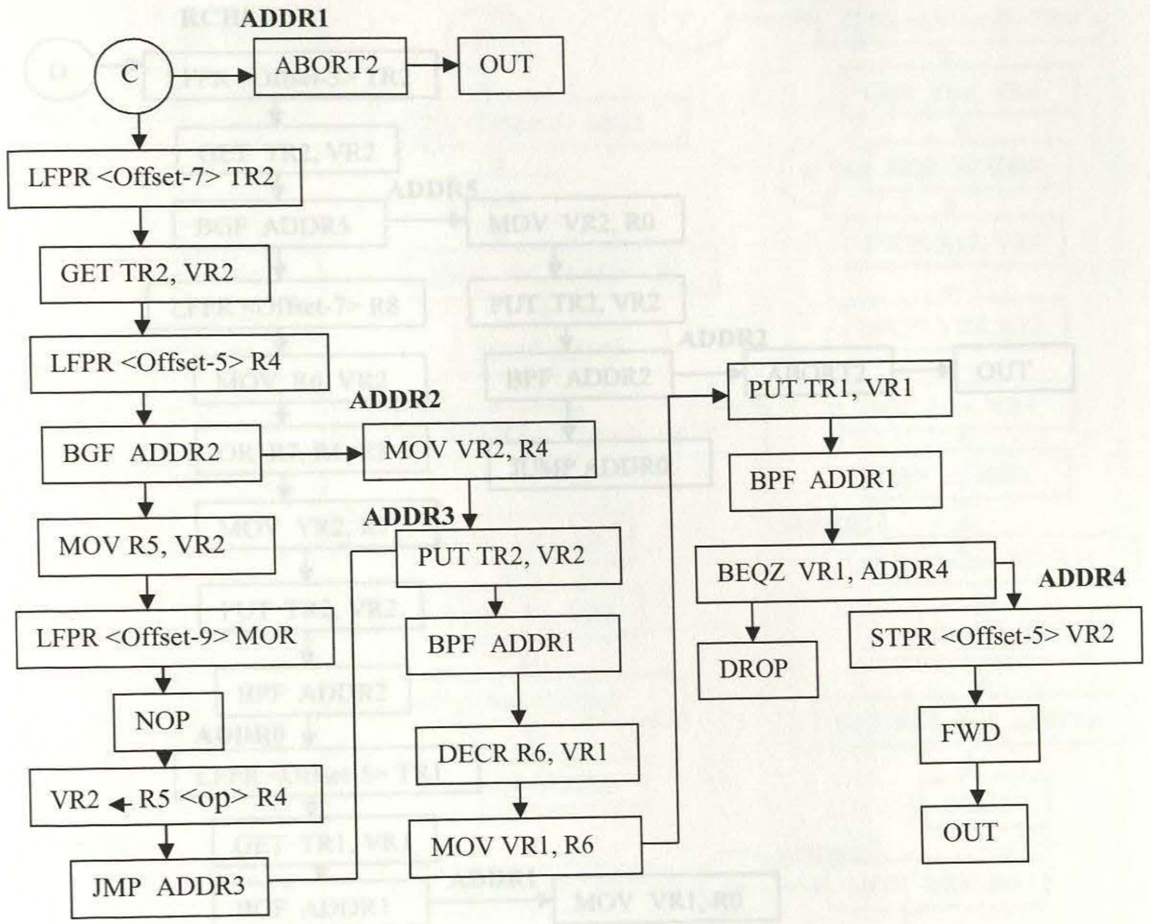
APPENDIX B

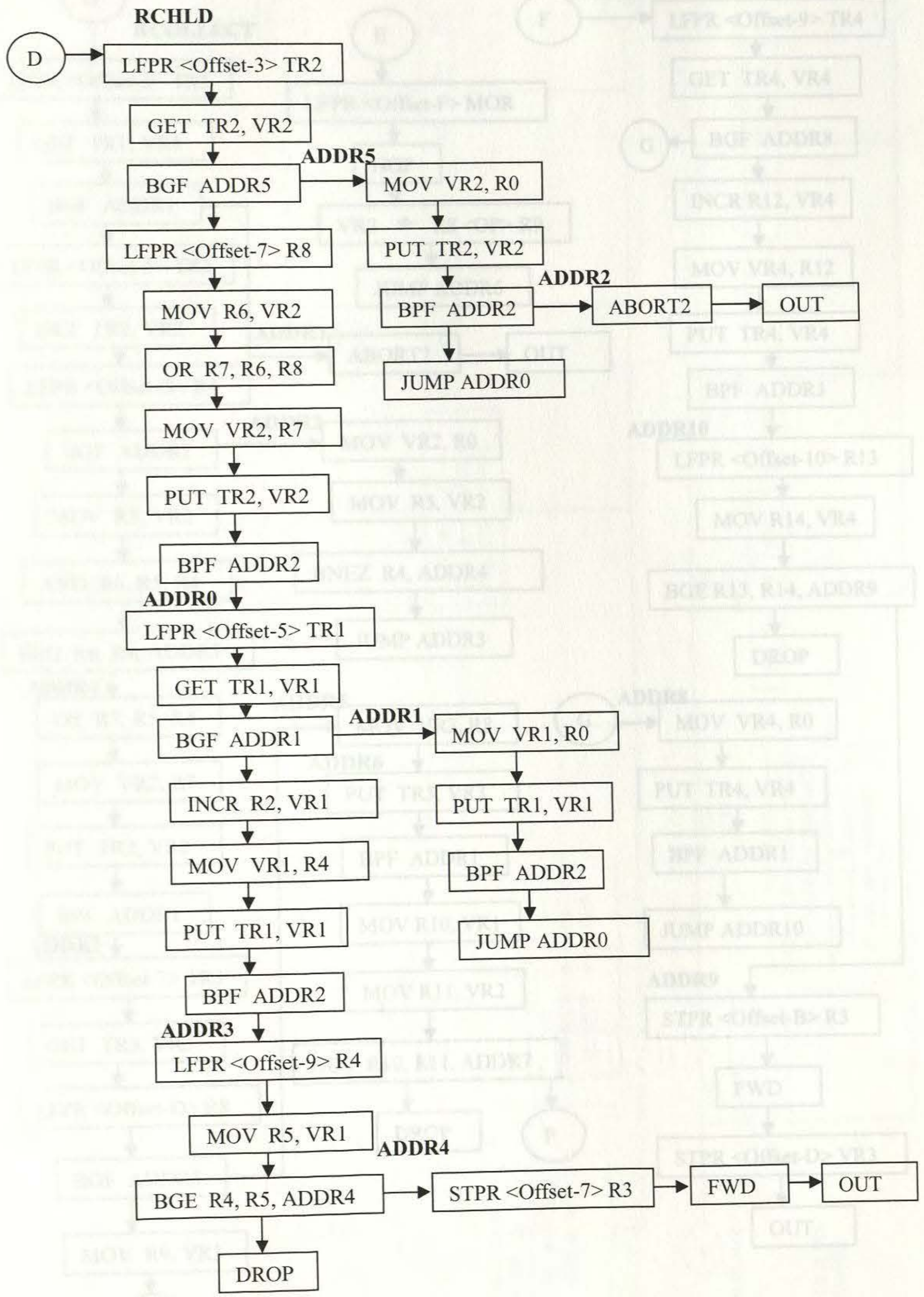
MACRO LEVEL SYSTEM FLOW CHART

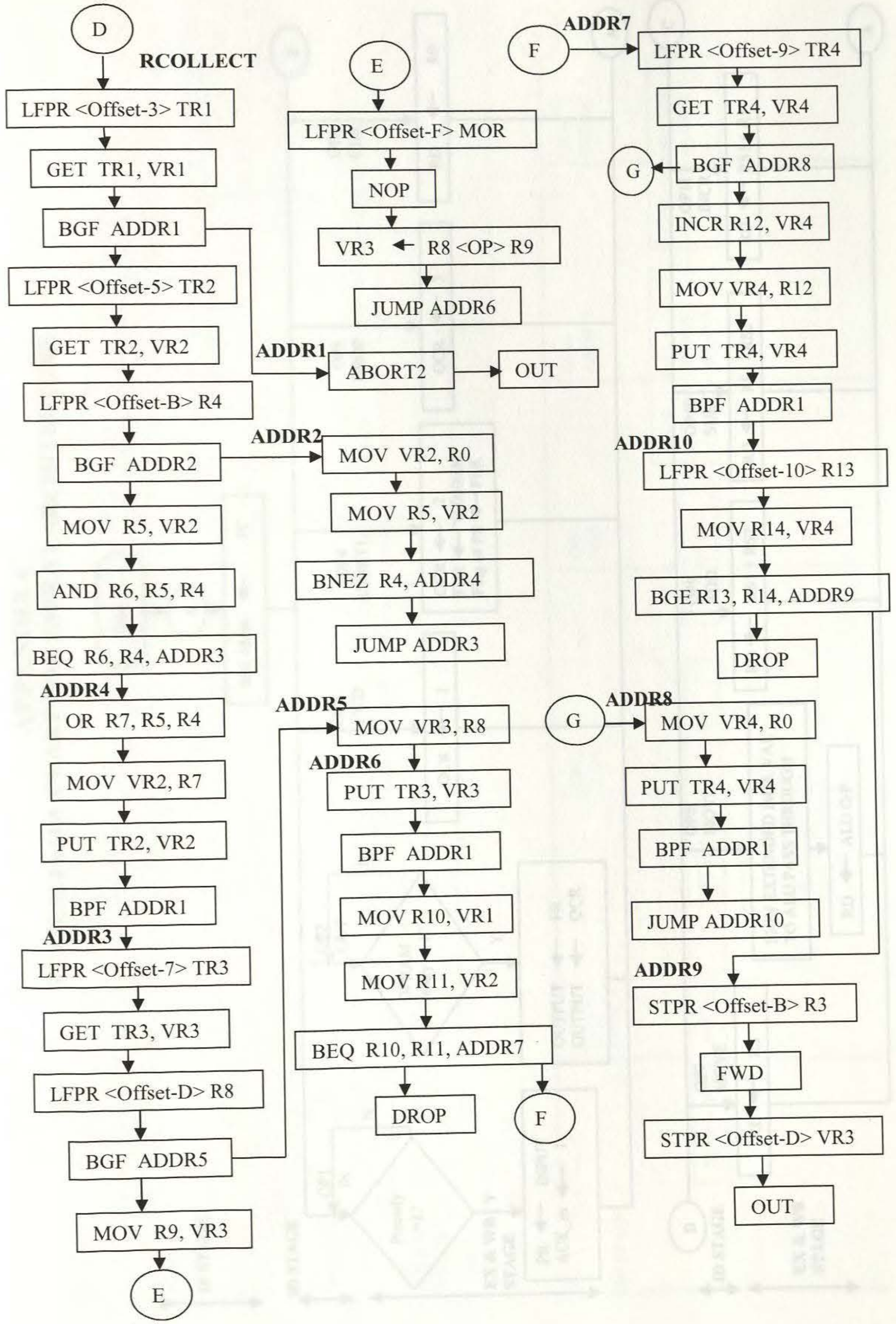



A
ADDR1
INCR R4, VR1
MOV VR1, R0
MOV VR1, R4
PUT TR1, VR1
PUT TR1, VR1
ADDR2
BPF ADDR2
ABORT2
BPF ADDR2
JMP ADDR3
OUT
ADDR3
LFPR <Offset-5> R4
MOV R5, VR1
ADDR4
BGE R4, R5, ADDR4
FWD
DROP
OUT

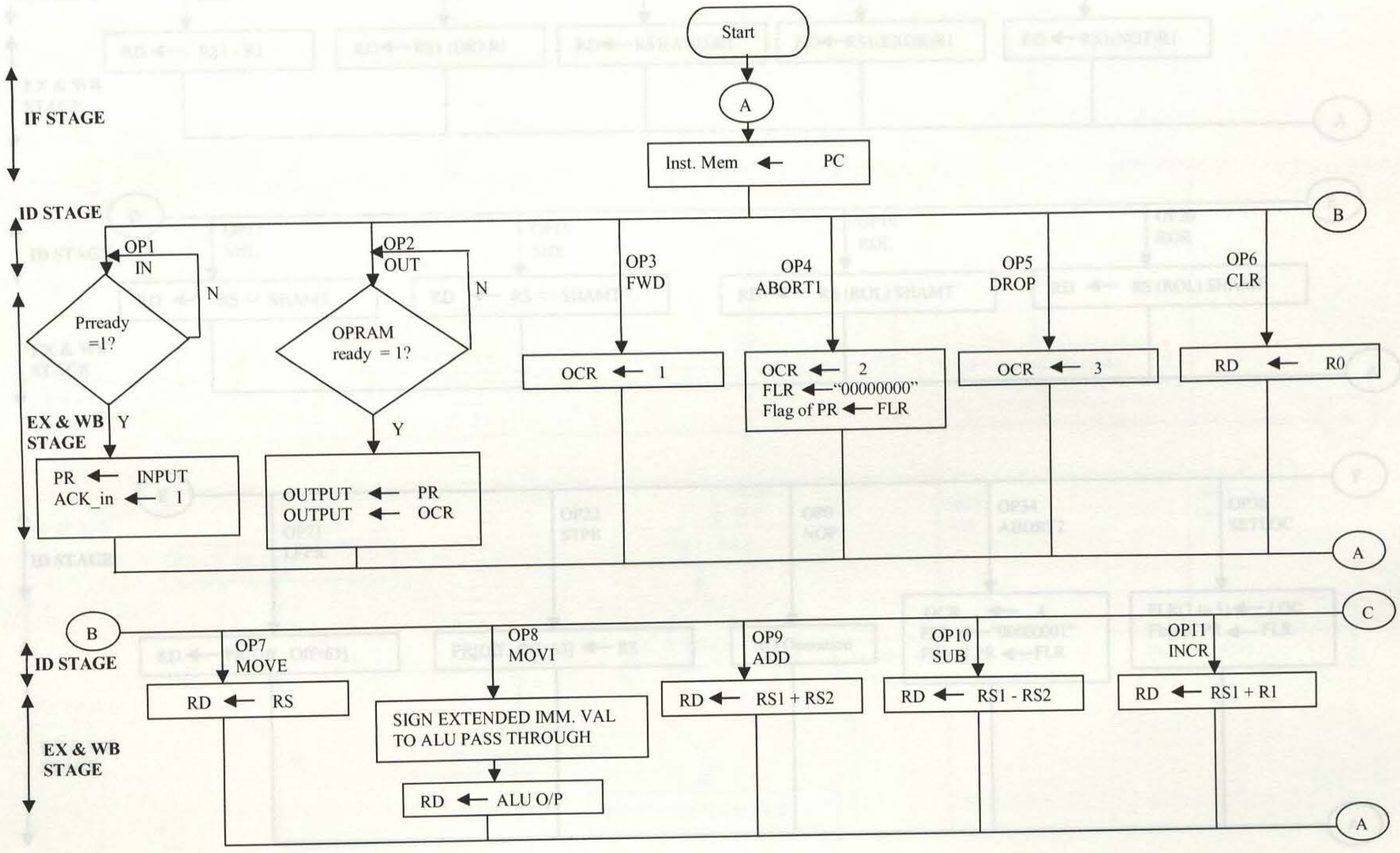

B
MOV R4, VR1
LFPR <Offset-7> MOR
NOP
ADDR1
R4 <OP> R5 ADDR1
MOV VR1, R5
DROP
PUT TR1, VR1
ADDR2
BPF ADDR2
FWD
ABORT2
OUT
OUT

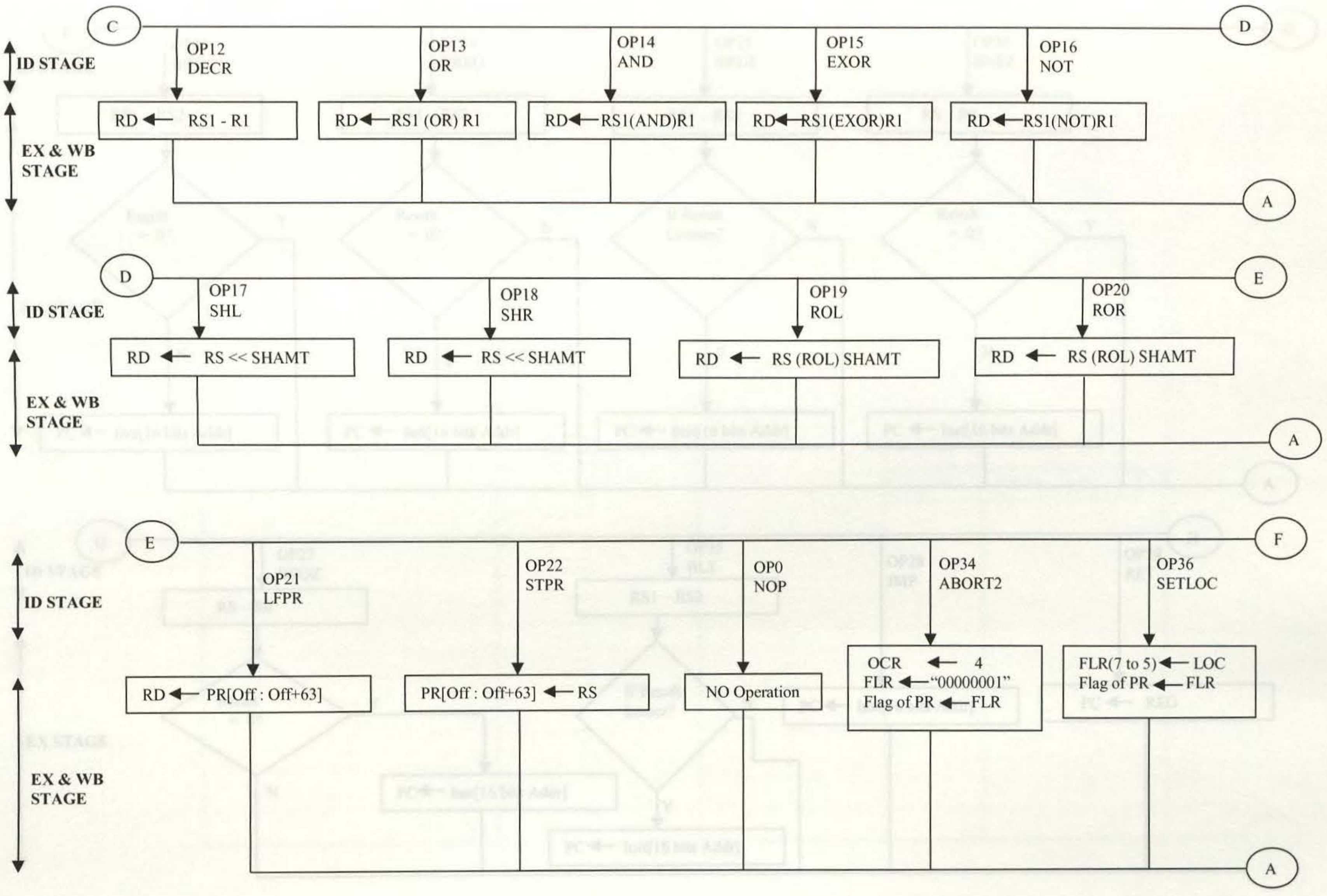

ADDR1
C
ABORT2
OUT
LFPR <Offset-7> TR2
GET TR2, VR2
LFPR <Offset-5> R4
BGF ADDR2
ADDR2
MOV VR2, R4
ADDR3
MOV R5, VR2
PUT TR2, VR2
LFPR <Offset-9> MOR
BPF ADDR1
NOP
DECR R6, VR1
VR2 ← R5 <op> R4
MOV VR1, R6
JMP ADDR3
PUT TR1, VR1
BPF ADDR1
BEQZ VR1, ADDR4
ADDR4
DROP
STPR <Offset-5> VR2
FWD
OUT

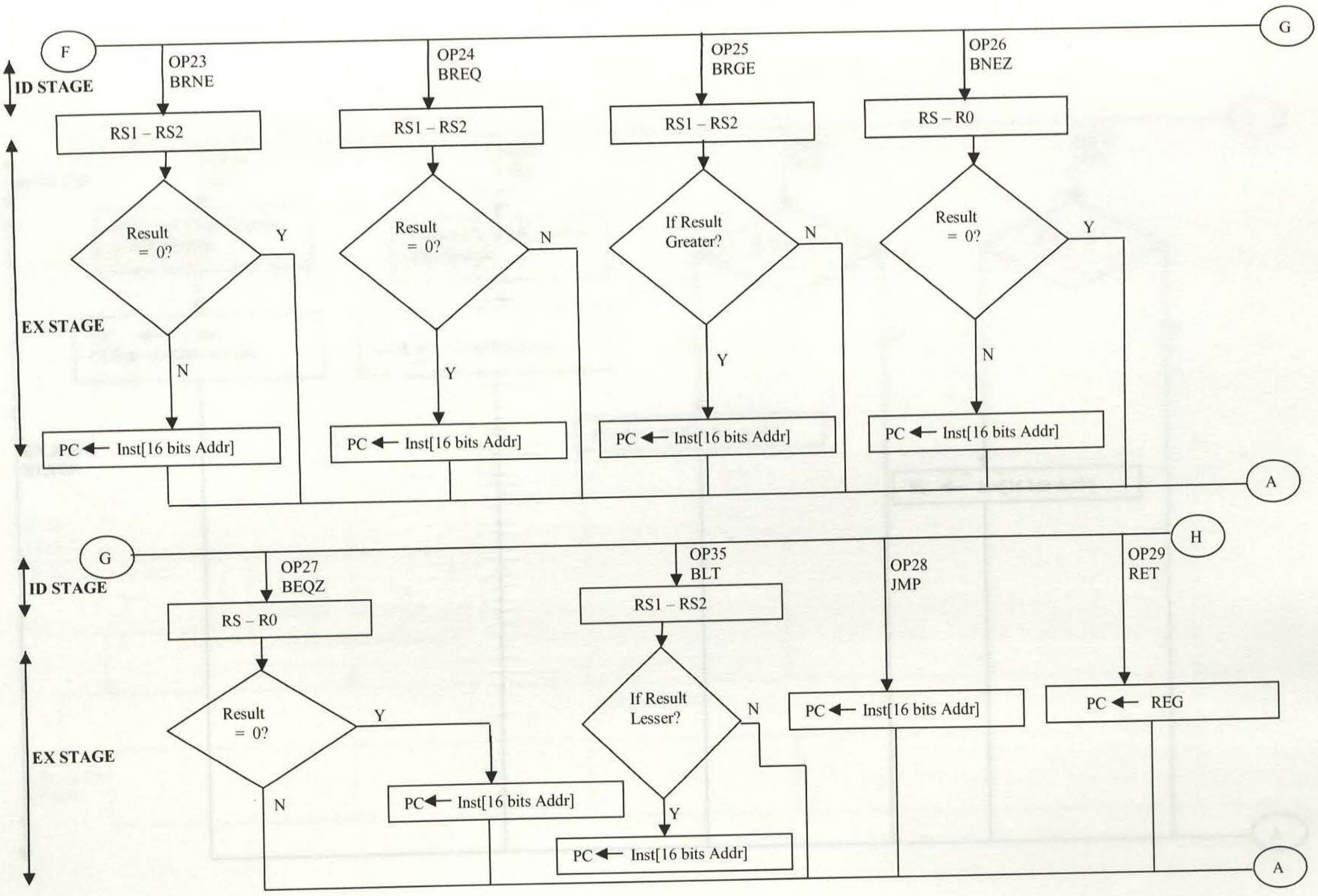

RCHLD
D
LFPR <Offset-3> TR2
GET TR2, VR2
BGF ADDR5
ADDR5
MOV VR2, R0
PUT TR2, VR2
ADDR2
BPF ADDR2
ABORT2
OUT
JUMP ADDR0
LFPR <Offset-7> R8
MOV R6, VR2
OR R7, R6, R8
MOV VR2, R7
PUT TR2, VR2
BPF ADDR2
ADDR0
LFPR <Offset-5> TR1
GET TR1, VR1
BGF ADDR1
ADDR1
MOV VR1, R0
PUT TR1, VR1
BPF ADDR2
JUMP ADDR0
INCR R2, VR1
MOV VR1, R4
PUT TR1, VR1
BPF ADDR2
ADDR3
LFPR <Offset-9> R4
MOV R5, VR1
ADDR4
BGE R4, R5, ADDR4
STPR <Offset-7> R3
FWD
OUT
DROP

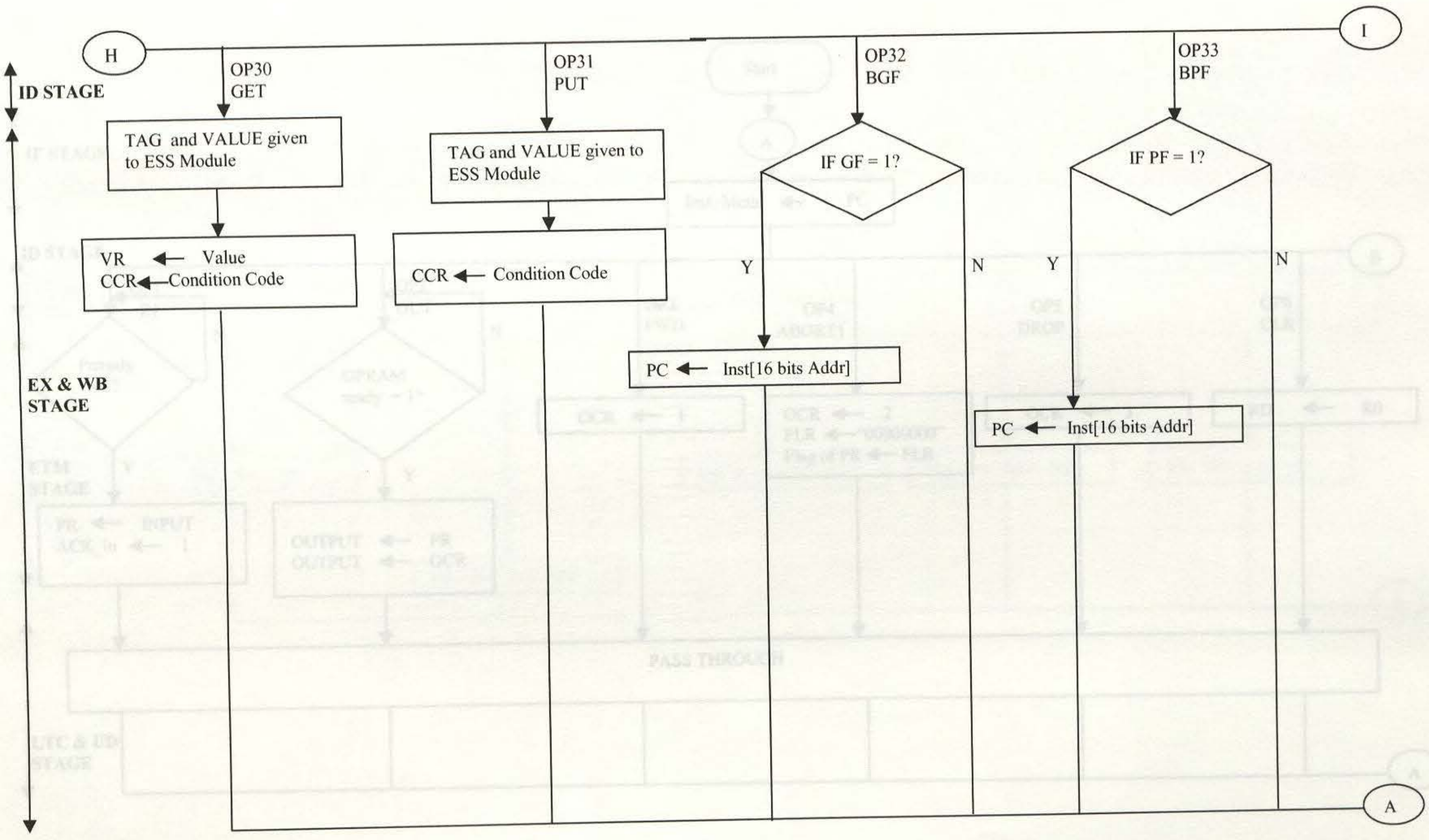

D
RCOLLECT
LFPR <Offset-3> TR1
GET TR1, VR1
BGF ADDR1
LFPR <Offset-5> TR2
GET TR2, VR2
LFPR <Offset-B> R4
BGF ADDR2
MOV R5, VR2
AND R6, R5, R4
BEQ R6, R4, ADDR3
ADDR4
OR R7, R5, R4
MOV VR2, R7
PUT TR2, VR2
BPF ADDR1
ADDR3
LFPR <Offset-7> TR3
GET TR3, VR3
LFPR <Offset-D> R8
BGF ADDR5
MOV R9, VR3
E
E
LFPR <Offset-F> MOR
NOP
VR3 ← R8 <OP> R9
JUMP ADDR6
ADDR1
ABORT2
OUT
ADDR2
MOV VR2, R0
MOV R5, VR2
BNEZ R4, ADDR4
JUMP ADDR3
ADDR5
MOV VR3, R8
ADDR6
PUT TR3, VR3
BPF ADDR1
MOV R10, VR1
MOV R11, VR2
BEQ R10, R11, ADDR7
DROP
F
F
ADDR7
LFPR <Offset-9> TR4
GET TR4, VR4
G
BGF ADDR8
INCR R12, VR4
MOV VR4, R12
PUT TR4, VR4
BPF ADDR1
ADDR10
LFPR <Offset-10> R13
MOV R14, VR4
BGE R13, R14, ADDR9
DROP
G
ADDR8
MOV VR4, R0
PUT TR4, VR4
BPF ADDR1
JUMP ADDR10
ADDR9
STPR <Offset-B> R3
FWD
STPR <Offset-D> VR3
OUT

APPENDIX C
SYSTEM FLOW CHART FOR ESPR.V1 ARCHITECTURE

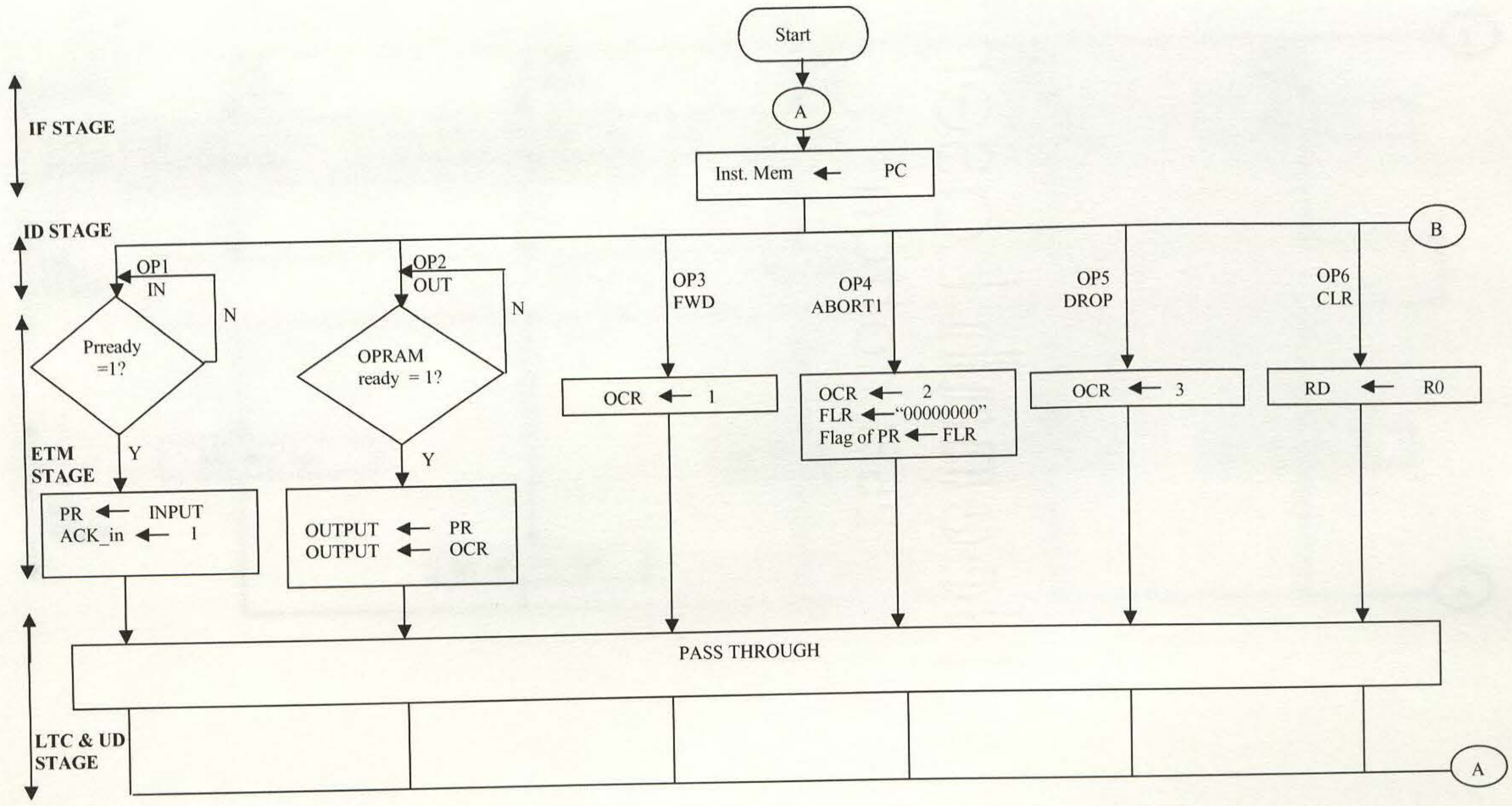


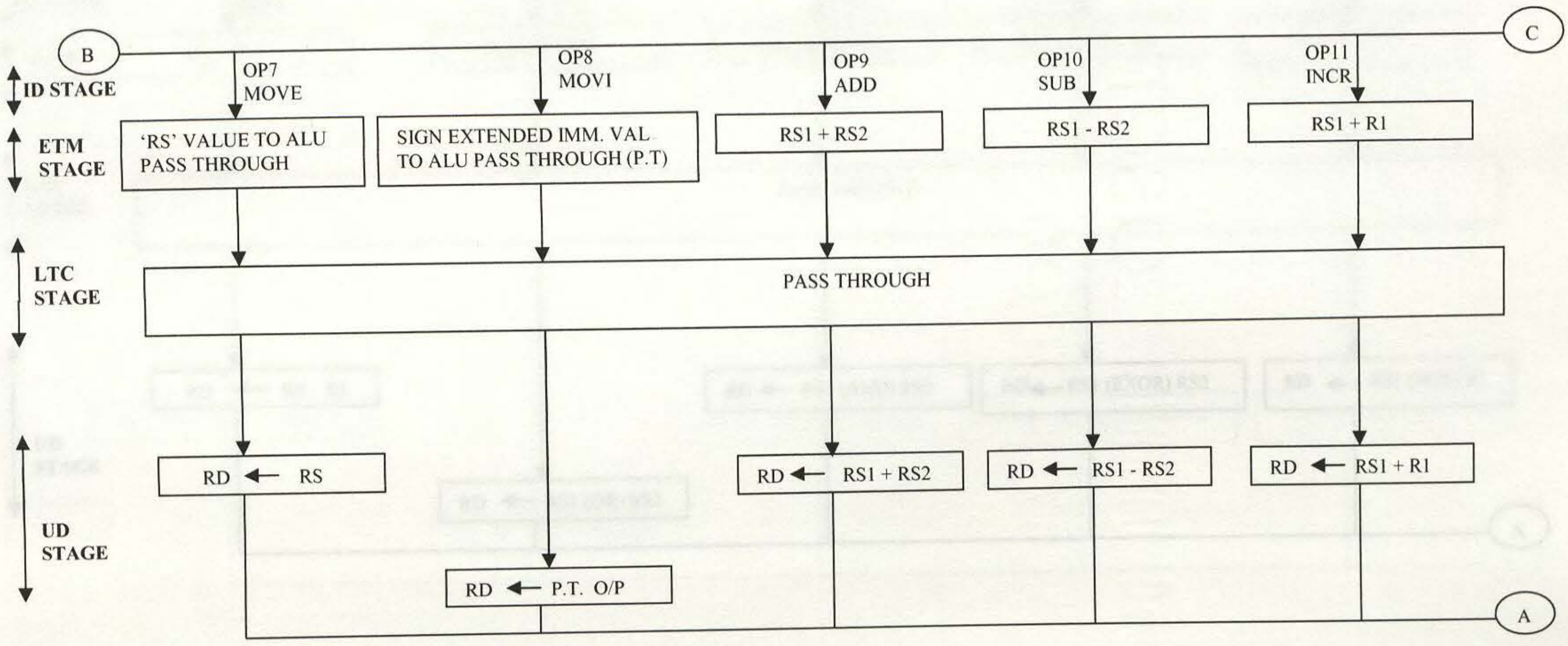

C
D
ID STAGE
EX & WB STAGE
OP12 DECR
OP13 OR
OP14 AND
OP15 EXOR
OP16 NOT
RD ← RS1 - R1
RD ← RS1 (OR) R1
RD ← RS1(AND)R1
RD ← RS1(EXOR)R1
RD ← RS1(NOT)R1
A
D
E
ID STAGE
EX & WB STAGE
OP17 SHL
OP18 SHR
OP19 ROL
OP20 ROR
RD ← RS << SHAMT
RD ← RS << SHAMT
RD ← RS (ROL) SHAMT
RD ← RS (ROL) SHAMT
A
E
F
ID STAGE
EX & WB STAGE
OP21 LFPR
OP22 STPR
OP0 NOP
OP34 ABORT2
OP36 SETLOC
RD ← PR[Off : Off+63]
PR[Off : Off+63] ← RS
NO Operation
OCR ← 4
FLR ← "00000001"
Flag of PR ← FLR
FLR(7 to 5) ← LOC
Flag of PR ← FLR
A

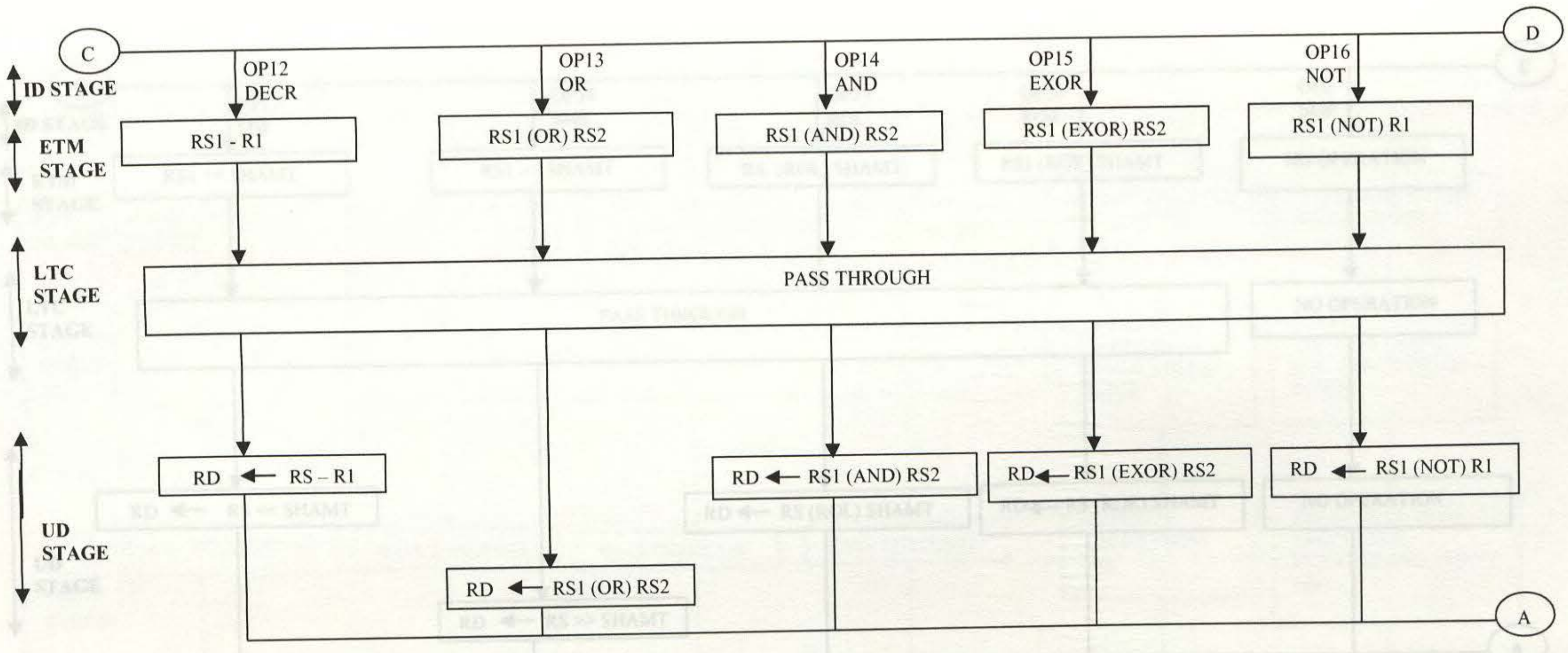

F
G
ID STAGE
EX STAGE
OP23
BRNE
RS1 – RS2
Result = 0?
Y
N
PC ← Inst[16 bits Addr]
OP24
BREQ
RS1 – RS2
Result = 0?
N
Y
PC ← Inst[16 bits Addr]
OP25
BRGE
RS1 – RS2
If Result Greater?
N
Y
PC ← Inst[16 bits Addr]
OP26
BNEZ
RS – R0
Result = 0?
Y
N
PC ← Inst[16 bits Addr]
A
G
H
ID STAGE
EX STAGE
OP27
BEQZ
RS – R0
Result = 0?
Y
N
PC ← Inst[16 bits Addr]
OP35
BLT
RS1 – RS2
If Result Lesser?
N
Y
PC ← Inst[16 bits Addr]
OP28
JMP
PC ← Inst[16 bits Addr]
OP29
RET
PC ← REG
A

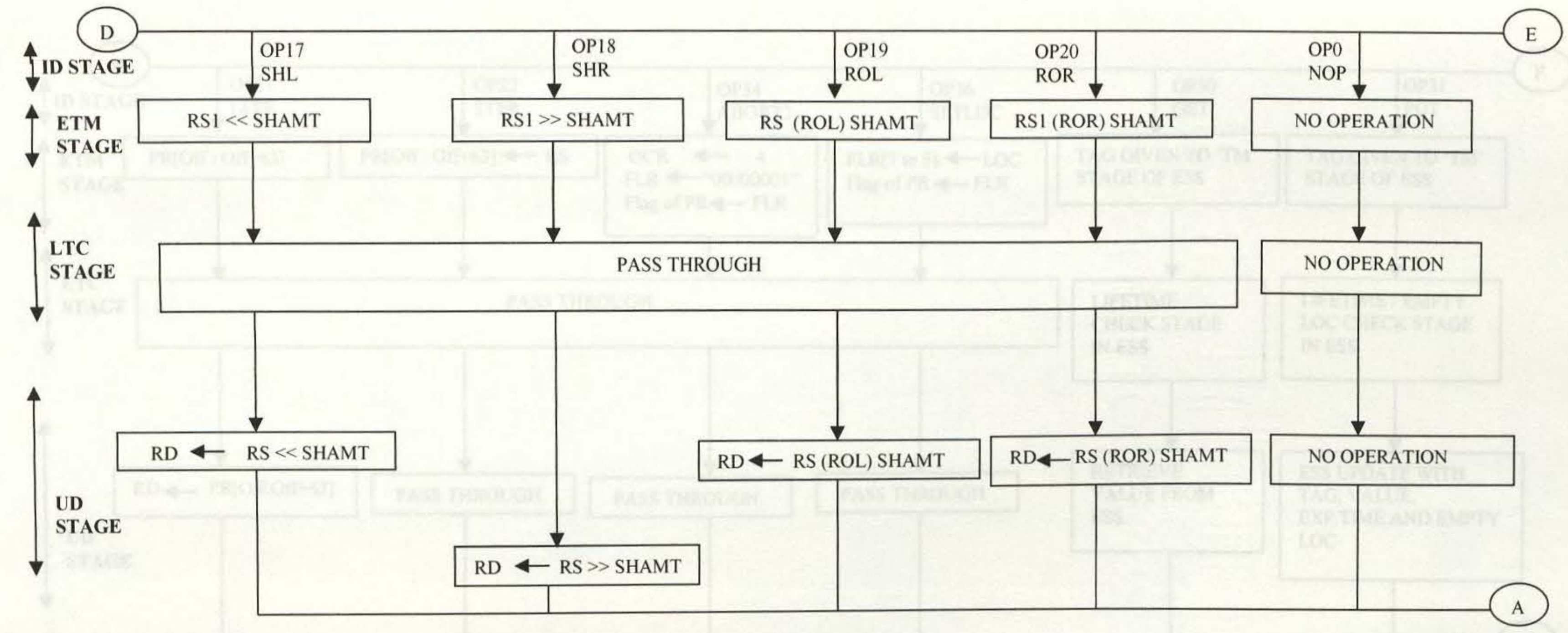

H
I
ID STAGE
EX & WB
STAGE
OP30
GET
TAG and VALUE given to ESS Module
VR ← Value
CCR ← Condition Code
OP31
PUT
TAG and VALUE given to ESS Module
CCR ← Condition Code
OP32
BGF
IF GF = 1?
Y
N
PC ← Inst[16 bits Addr]
OP33
BPF
IF PF = 1?
Y
N
PC ← Inst[16 bits Addr]
A

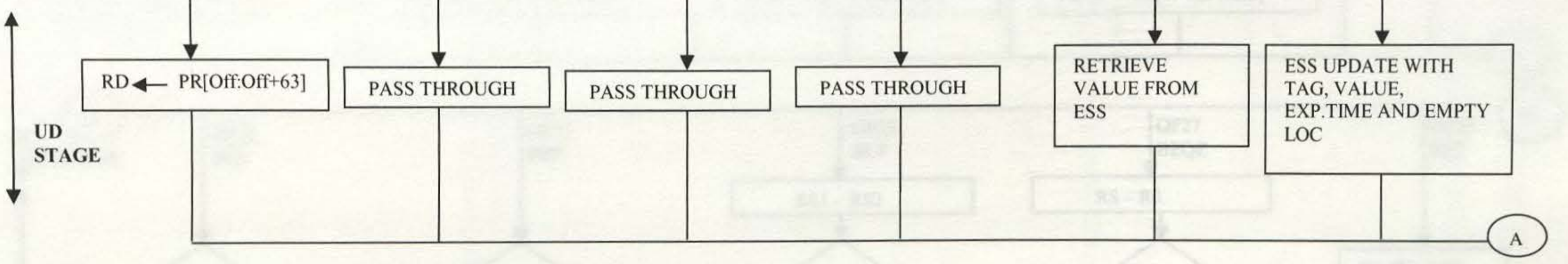
APPENDIX D
SYSTEM FLOW CHART FOR ESPR.V2 ARCHITECTURE

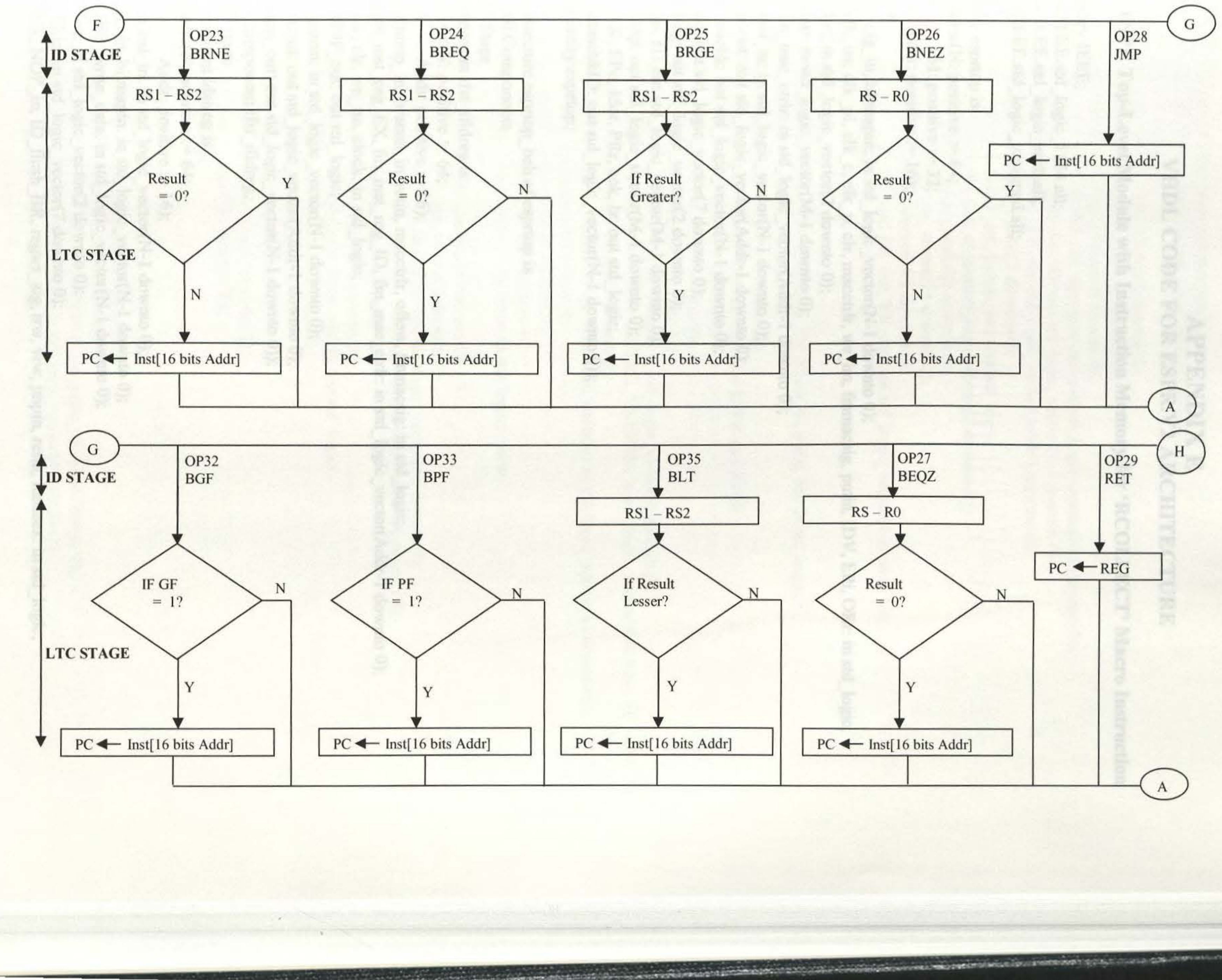



B
C
ID STAGE
ETM STAGE
LTC STAGE
UD STAGE
OP7 MOVE
OP8 MOVI
OP9 ADD
OP10 SUB
OP11 INCR
'RS' VALUE TO ALU PASS THROUGH
SIGN EXTENDED IMM. VAL TO ALU PASS THROUGH (P.T)
RS1 + RS2
RS1 - RS2
RS1 + R1
PASS THROUGH
RD ← RS
RD ← P.T. O/P
RD ← RS1 + RS2
RD ← RS1 - RS2
RD ← RS1 + R1
A


C
D
ID STAGE
ETM STAGE
LTC STAGE
UD STAGE
OP12 DECR
OP13 OR
OP14 AND
OP15 EXOR
OP16 NOT
RS1 - R1
RS1 (OR) RS2
RS1 (AND) RS2
RS1 (EXOR) RS2
RS1 (NOT) R1
PASS THROUGH
RD ← RS – R1
RD ← RS1 (OR) RS2
RD ← RS1 (AND) RS2
RD ← RS1 (EXOR) RS2
RD ← RS1 (NOT) R1
A


D
E
ID STAGE
OP17
SHL
OP18
SHR
OP19
ROL
OP20
ROR
OP0
NOP
ETM STAGE
RS1 << SHAMT
RS1 >> SHAMT
RS (ROL) SHAMT
RS1 (ROR) SHAMT
NO OPERATION
LTC STAGE
PASS THROUGH
NO OPERATION
UD STAGE
RD ← RS << SHAMT
RD ← RS (ROL) SHAMT
RD ← RS (ROR) SHAMT
NO OPERATION
RD ← RS >> SHAMT
A


E
F
ID STAGE
ETM STAGE
LTC STAGE
UD STAGE
OP21 LFPR
OP22 STPR
OP34 ABORT2
OP36 SETLOC
OP30 GET
OP31 PUT
PR[Off : Off+63]
PR[Off : Off+63] ← RS
OCR ← 4
FLR ← "00000001"
Flag of PR ← FLR
FLR(7 to 5) ← LOC
Flag of PR ← FLR
TAG GIVEN TO 'TM' STAGE OF ESS
TAG GIVEN TO 'TM' STAGE OF ESS
PASS THROUGH
LIFETIME CHECK STAGE IN ESS
LIFETIME / EMPTY LOC CHECK STAGE IN ESS
RD ← PR[Off:Off+63]
PASS THROUGH
PASS THROUGH
PASS THROUGH
RETRIEVE VALUE FROM ESS
ESS UPDATE WITH TAG, VALUE, EXP.TIME AND EMPTY LOC
A


F
G
ID STAGE
LTC STAGE
OP23
BRNE
RS1 – RS2
Result = 0?
Y
N
PC ← Inst[16 bits Addr]
OP24
BREQ
RS1 – RS2
Result = 0?
N
Y
PC ← Inst[16 bits Addr]
OP25
BRGE
RS1 – RS2
If Result Greater?
N
Y
PC ← Inst[16 bits Addr]
OP26
BNEZ
RS – R0
Result = 0?
Y
N
PC ← Inst[16 bits Addr]
OP28
JMP
PC ← Inst[16 bits Addr]
A
G
H
ID STAGE
LTC STAGE
OP32
BGF
IF GF = 1?
N
Y
PC ← Inst[16 bits Addr]
OP33
BPF
IF PF = 1?
N
Y
PC ← Inst[16 bits Addr]
OP35
BLT
RS1 – RS2
If Result Lesser?
N
Y
PC ← Inst[16 bits Addr]
OP27
BEQZ
RS – R0
Result = 0?
N
Y
PC ← Inst[16 bits Addr]
OP29
RET
PC ← REG
A

APPENDIX E

VHDL CODE FOR ESPR.V2 ARCHITECTURE

1. ESPR Top-Level Module with Instruction Memory for ‘RCOLLECT’ Macro Instruction

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity esprtop is
generic(N: positive := 64;
          M: positive := 32;
      Addr: positive := 16);
port(
--    cfg_in, bitmapin: in std_logic_vector(N-1 downto 0);
    clk_im, clk_pi, clk_c, clk_p, clr, macctrlr, we_im, fmmacsig, putin, IDV, EPi, ORr: in std_logic;
    loc: in std_logic_vector(2 downto 0);
    inp: in std_logic_vector(M-1 downto 0);
    fm_mac_ctrlr: in std_logic_vector(Addr-1 downto 0);
    inst_in: in std_logic_vector(N-1 downto 0);
    pcout: out std_logic_vector(Addr-1 downto 0);
    instchk: out std_logic_vector(N-1 downto 0);
    oo: out std_logic_vector(7 downto 0);
    stag: out std_logic_vector(2 downto 0);
    po, f1: out std_logic_vector(M-1 downto 0);
    outp: out std_logic_vector(M-1 downto 0);
    AK, EPo, ldor, PRr, cok, lz: out std_logic;
    datachkID: out std_logic_vector(N-1 downto 0));
end entity esprtop;

architecture esprtop_beh of esprtop is
-- All Components
--IF Stage
component ifst_ifidreg is
generic(N: positive := 64;
          Addr: positive := 16);
port(jump_in, branch_in, retin, macctrlr, oflow, fmmacsig: in std_logic;
    fm_inst_reg_EX, fm_inst_reg_ID, fm_mac_ctrlr: in std_logic_vector(Addr-1 downto 0);
    clk, clr, we_im, clock: in std_logic;
    NOP_out: out std_logic;
    instrin: in std_logic_vector(N-1 downto 0);
    pcout: out std_logic_vector(Addr-1 downto 0);
    inst_out: out std_logic_vector(N-1 downto 0));
end component ifst_ifidreg;
--ID stage
component idstreg is
generic(N: positive := 64;
          Addr: positive := 16);
port(inst_in: in std_logic_vector(N-1 downto 0);
--cfg_in, bitmapin: in std_logic_vector(N-1 downto 0);
    WB_write_data: in std_logic_vector(N-1 downto 0);
    loc: in std_logic_vector(2 downto 0);
    ffpin: in std_logic_vector(7 downto 0);
    clk, NOP_in, ID_flush_BR, regwr_sig, trw, vrw, jmpin, retin, lmfmex: in std_logic;
    morfmex: in std_logic_vector(5 downto 0);
    TRDstin, VRDstin, RDstin: in std_logic_vector(4 downto 0);
    IDFout: out std_logic;
    WB_ctrl_out: out std_logic_vector(3 downto 0);
    EX_ctrl_out: out std_logic_vector(12 downto 0);
    PKT_ctrl_out: out std_logic_vector(6 downto 0);
    GPR_read1_out, GPR_read2_out, sign_ext_out: out std_logic_vector(N-1 downto 0);
    TR_read_out_ID, VR_read_out_ID: out std_logic_vector(N-1 downto 0);
    Br_Addr_out, PKT_Offset_out: out std_logic_vector(Addr-1 downto 0);
    shamt_out: out std_logic_vector(5 downto 0);
    lmor_out, TRD_out, VRD_out, jumps, rets: out std_logic;
    ocr_val_out_id, aer_val_out_id: out std_logic_vector(7 downto 0);
    opcodeexout: out std_logic_vector(5 downto 0);
    ctrlsigsoutID: out std_logic_vector(24 downto 0);
    wrdataout: out std_logic_vector(N-1 downto 0);
    RS1_out, RS2_out, RD_out, TR_out, VR_out: out std_logic_vector(4 downto 0));
end component idstreg;
--ETM Stage
component ex3top is
port(clk, clock, clk_pkt, clr, IDV, EPi, ORr, EX_Flush_in, putin, lm: in std_logic;
    iram: in std_logic_vector(31 downto 0);
    flag, ocrID: in std_logic_vector(7 downto 0);
    PKToffid: in std_logic_vector(6 downto 0); -- for LFPR and STPR
    braddrin: in std_logic_vector(15 downto 0);
    ctrlinEX: in std_logic_vector(24 downto 0);
    WBinfmid: in std_logic_vector(3 downto 0);
    RS1rgid, RS2rgid, RDrgid, TRrgid, VRrgid: in std_logic_vector(4 downto 0);
    FSTRD, FSTTRD, FSTVRD, VSTRD, VSTTRD, VSTVRD: in std_logic_vector(4 downto 0); --new
    op_in, prop_in: in std_logic_vector(5 downto 0);
    GPR1id, GPR2id, TRidv, VRidv, extid, WBdatain, aofmex: in std_logic_vector(63 downto 0);
    EXctid: in std_logic_vector(9 downto 0);
    PKTctid: in std_logic_vector(6 downto 0);
    shamt: in std_logic_vector(5 downto 0);
    regrd, trwx, trww, vrwx, vrww, rwx, rww: in std_logic; --new
    alu_O: out std_logic;
    ctrloutEX: out std_logic_vector(24 downto 0);
    opoutEX, mo: out std_logic_vector(5 downto 0);
    aluout, GPR1out, GPR2out, tagsigout: out std_logic_vector(63 downto 0);
    RS1_out, RS2_out, RD_out, TR_out, VR_out: out std_logic_vector(4 downto 0);
    WBct_out: out std_logic_vector(3 downto 0);
    braddrout: out std_logic_vector(15 downto 0);
    gf, pf, ess_full, le, AK, PRr, ldor, EPo, cok, lz: out std_logic;
    outvalue: out std_logic_vector(63 downto 0);
    oo: out std_logic_vector(7 downto 0);
    stag: out std_logic_vector(2 downto 0);
    oram, f1: out std_logic_vector(31 downto 0);
    po: out std_logic_vector(31 downto 0));
end component ex3top;
--LTC Stage
component ex4top is
port(clk: in std_logic;
    WBctrlin: in std_logic_vector(3 downto 0);
    out_fm_alu: in std_logic_vector(63 downto 0);
    RS1in, RS2in, VRin, VSTRD, VSTVRD: in std_logic_vector(4 downto 0);
    RDin_fm4, VRDin_fm4, TRDin_fm4: in std_logic_vector(4 downto 0);
    op_in: in std_logic_vector(5 downto 0);
```

```
  GPRin1, GPRin2, PTin: in std_logic_vector(63 downto 0);
  brtype: in std_logic_vector(2 downto 0);
  ccr_inp, ccr_ing: in std_logic;
  branch: out std_logic;
  WBctout: out std_logic_vector(3 downto 0);
  WBdataout: out std_logic_vector(63 downto 0);
  WBRDout, WBVRDout, WBTRDout: out std_logic_vector(4 downto 0));
end component ex4top;
--UD Stage
component stage5 is
port(WB_in1: in std_logic;
  aluout_fm_ex, essout_fm_st5: in std_logic_vector(63 downto 0);
  dataout: out std_logic_vector(63 downto 0));
end component stage5;

--signals
signal ffpsig: std_logic_vector(7 downto 0);
-- IF
signal instsig: std_logic_vector(63 downto 0);
signal bsig_EX4o, ovf, NOP_IFo: std_logic;
signal brao: std_logic_vector(15 downto 0);
--ID
signal data_WBo: std_logic_vector(63 downto 0);
signal grw_EX4o, trw_EX4o, vrw_EX4o, IDFL, jmp_IDo, ret_IDo: std_logic;
signal RS1o,RS2o,TRo,VRo,RDo: std_logic_vector(4 downto 0);
signal IDFo, Lm, TRWR_IDo, VRWR_IDo: std_logic;
signal WBo: std_logic_vector(3 downto 0);
signal EX34ct_IDo: std_logic_vector(12 downto 0);
signal PKct2o: std_logic_vector(6 downto 0);
signal GR12o, GR22o, se2o, Tda2o, Vda2o, wrdata_IDo: std_logic_vector(N-1 downto 0);
signal PKTOff_IDo: std_logic_vector(Addr-1 downto 0);
signal sh2o: std_logic_vector(5 downto 0);
signal ocro, aero: std_logic_vector(7 downto 0);
signal op2o: std_logic_vector(5 downto 0);
signal ctlo: std_logic_vector(24 downto 0);
--ETM
signal EXFL, rfsig: std_logic;
signal op3o, mo: std_logic_vector(5 downto 0);
signal alu3o, GR13o, GR23o: std_logic_vector(63 downto 0);
signal ctl3o: std_logic_vector(24 downto 0);
signal RS1E3o,RS2E3o,TRE3o,VRE3o,RDE3o: std_logic_vector(4 downto 0);
signal WBct3o: std_logic_vector(3 downto 0);
signal braddr_EX3o: std_logic_vector(15 downto 0);
signal GFo, PFo, EFo, leo: std_logic;
signal POff: std_logic_vector(6 downto 0);
signal EX34cto: std_logic_vector(9 downto 0);
signal f1o: std_logic_vector(31 downto 0);
--LTC
signal WBct4o: std_logic_vector(3 downto 0);
signal TRE4o, VRE4o, RDE4o: std_logic_vector(4 downto 0);
signal da4o: std_logic_vector(63 downto 0);
--UD
signal esso: std_logic_vector(63 downto 0);
--Other signals
signal ts: std_logic_vector(63 downto 0);
```

```
begin

--Output
instchk <= instsig;
datachkID <= data_WBo;
fl <= f1o;
ffpsig <= f1o(7 downto 0); --ID

--other signals
--ID
grw_EX4o <= WBct4o(0); -- reg write
trw_EX4o <= WBct4o(3); -- tag reg write
vrw_EX4o <= WBct4o(2); -- val reg write
IDFL <= bsig_EX4o or ovf;
--ETM
EXFL <= IDFL;
POff <= PKTOff_IDo(6 downto 0);
EX34cto <= EX34ct_IDo(9 downto 0);
--UD
--esso <= (others => '0');

IFCOMP: ifst_ifidreg port
map(jump_in=>jmp_IDo,branch_in=>bsig_EX4o,retin=>ret_IDo,macctrlr=>macctrlr,oflow=>ovf,fmmacsi
g=>fmmacsig,fm_inst_reg_EX=>braddr_EX3o,fm_inst_reg_ID=>brao,fm_mac_ctrlr=>fm_mac_ctrlr,clk=
>clk_pi,clr=>clr,we_im=>we_im,clock=>clk_im,NOP_out=>NOP_IFo,instrin=>inst_in,pcout=>pcout,inst
_out=>instsig);

IDCOMP: idstreg port
map(inst_in=>instsig,WB_write_data=>data_WBo,loc=>loc,ffpin=>ffpsig,clk=>clk_pi,NOP_in=>NOP_IF
o,ID_flush_BR=>IDFL,regwr_sig=>grw_EX4o,trw=>trw_EX4o,vrw=>vrw_EX4o,jmpin=>jmp_IDo,retin
=>ret_IDo,lmfmex=>Lm,morfmex=>mo,TRDstin=>TRE4o,VRDstin=>VRE4o,RDstin=>RDE4o,IDFout=
>IDFo,WB_ctrl_out=>WBo,EX_ctrl_out=>EX34ct_IDo,PKT_ctrl_out=>PKct2o,GPR_read1_out=>GR12
o,GPR_read2_out=>GR22o,sign_ext_out=>se2o,TR_read_out_ID=>Tda2o,VR_read_out_ID=>Vda2o,Br_
Addr_out=>brao,PKT_Offset_out=>PKTOff_IDo,shamt_out=>sh2o,lmor_out=>Lm,TRD_out=>TRWR_I
Do,VRD_out=>VRWR_IDo,jumps=>jmp_IDo,rets=>ret_IDo,ocr_val_out_id=>ocro,aer_val_out_id=>aer
o,opcodeexout=>op2o,ctrlsigsoutID=>ctlo,wrdataout=>wrdata_IDo,RS1_out=>RS1o,RS2_out=>RS2o,RD
_out=>RDo,TR_out=>TRo,VR_out=>VRo);

EX3COMP:ex3top port
map(clk=>clk_pi,clock=>clk_c,clk_pkt=>clk_p,clr=>clr,IDV=>IDV,EPi=>EPi,ORr=>ORr,EX_Flush_in=
>EXFL,putin=>putin,lm=>Lm,iram=>inp,flag=>aero,ocrID=>ocro,PKToffid=>POff,braddrin=>brao,ctrlin
EX=>ctlo,WBinfmid=>WBo,RS1rgid=>RS1o,RS2rgid=>RS2o,RDrgid=>RDo,TRrgid=>TRo,VRrgid=>V
Ro,FSTRD=>RDE3o,FSTTRD=>TRE3o,FSTVRD=>VRE3o,VSTRD=>RDE4o,VSTTRD=>TRE4o,VST
VRD=>VRE4o,op_in=>op2o,prop_in=>op3o,GPR1id=>GR12o,GPR2id=>GR22o,TRidv=>Tda2o,VRidv
=>Vda2o,extid=>se2o,WBdatain=>da4o,aofmex=>alu3o,EXctid=>EX34cto,PKTctid=>PKct2o,shamt=>sh
2o,regrd=>ctlo(22),trwx=>WBct3o(3),trww=>WBct4o(3),vrwx=>WBct3o(2),vrww=>WBct4o(2),rwx=>
WBct3o(0),rww=>WBct4o(0),alu_O=>ovf,ctrloutEX=>ctl3o,opoutEX=>op3o,mo=>mo,aluout=>alu3o,GP
R1out=>GR13o,GPR2out=>GR23o,tagsigout=>ts,RS1_out=>RS1E3o,RS2_out=>RS2E3o,RD_out=>RDE
3o,TR_out=>TRE3o,VR_out=>VRE3o,WBct_out=>WBct3o,braddrout=>braddr_EX3o,gf=>GFo,pf=>PF
o,ess_full=>EFo,le=>leo,AK=>AK,PRr=>PRr,ldor=>ldor,EPo=>EPo,cok=>cok,lz=>lz,outvalue=>esso,oo
=>oo,stag=>stag,oram=>outp,f1=>f1o,po=>po);

EX4COMP: ex4top port
map(clk=>clk_pi,WBctrlin=>WBct3o,out_fm_alu=>alu3o,RS1in=>RS1E3o,RS2in=>RS2E3o,VRin=>VR
E3o,VSTRD=>RDE4o,VSTVRD=>VRE4o,RDin_fm4=>RDE3o,VRDin_fm4=>VRE3o,TRDin_fm4=>T
RE3o,op_in=>op3o,GPRin1=>GR12o,GPRin2=>GR22o,PTin=>da4o,brtype=>ctl3o(19 downto
17),ccr_inp=>PFo,ccr_ing=>GFo,branch=>bsig_EX4o,WBctout=>WBct4o,WBdataout=>da4o,WBRDout
=>RDE4o,WBVRDout=>VRE4o,WBTRDout=>TRE4o);

WBCOMP: stage5 port
map(WB_in1=>WBct4o(1),aluout_fm_ex=>da4o,essout_fm_st5=>esso,dataout=>data_WBo);

end architecture esprtop_beh;
```

2. IF STAGE

```
--Individual Componenets
-- IF STAGE FULL

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ifst_ifidreg is
generic(N: positive := 64;
        Addr: positive := 16);
port(jump_in, branch_in, retin, macctrlr, oflow, fmmacsig: in std_logic;
   fm_inst_reg_EX, fm_inst_reg_ID, fm_mac_ctrlr: in std_logic_vector(Addr-1 downto 0);
   clk, clr, we_im, clock: in std_logic;
   NOP_out: out std_logic;
   instrin: in std_logic_vector(N-1 downto 0);
   pcout: out std_logic_vector(Addr-1 downto 0);
   inst_out: out std_logic_vector(N-1 downto 0));
end entity ifst_ifidreg;

architecture ifidstregbeh of ifst_ifidreg is
-- IF pipe component
component if_pipe is
generic(N: positive := 64;
        Addr: positive :=16); -- 16
port(jump_in, branch_in, retin, macctrlr, oflow, fmmacsig: in std_logic;
   fm_inst_reg, fm_mac_ctrlr: in std_logic_vector(Addr-1 downto 0);
   clk, clr, we_im, clock: in std_logic;
   instrin: in std_logic_vector(N-1 downto 0);
   NOP_out: out std_logic;
   inst_out: out std_logic_vector(N-1 downto 0);
   pc_out: out std_logic_vector(Addr-1 downto 0));
end component if_pipe;
-- IFID register component
component ifidreg is
port(clr, clk: in std_logic;
   instrin: in std_logic_vector(63 downto 0);
   instrouttoid: out std_logic_vector(63 downto 0));
end component ifidreg;
--Incr PC Gen
component ipcchk is
port(opfipcin: in std_logic_vector(5 downto 0);
   opipcout: out std_logic);
end component ipcchk;
--MUX to choose inst reg address
component mux_inst is
```

```
port(a: in STD_LOGIC_VECTOR (15 downto 0);
   b: in STD_LOGIC_VECTOR (15 downto 0);
   s: in STD_LOGIC;
   y: out STD_LOGIC_VECTOR (15 downto 0) );
end component mux_inst;

-- signals
signal instoutsig: std_logic_vector(N-1 downto 0);
signal ipc, fmmacsig1, sinstmux: std_logic;
signal muxinstaddr: std_logic_vector(15 downto 0);

begin

sinstmux <= jump_in or retin;
fmmacsig1 <= fmmacsig and ipc;

ifpipecomp: if_pipe port map(jump_in=>jump_in, branch_in=>branch_in, retin=>retin,
macctrlr=>macctrlr, oflow=>oflow, fmmacsig=>fmmacsig1, fm_inst_reg=>muxinstaddr,
fm_mac_ctrlr=>fm_mac_ctrlr, clk=>clk, clr=>clr, we_im=>we_im, clock=>clock, instrin=>instrin,
NOP_out=>NOP_out, inst_out=>instoutsig, pc_out=>pcout);

ifidregcomp: ifidreg port map(clr=>clr, clk=>clk, instrin=>instoutsig, instrouttoid=>inst_out);
ipcgencomp: ipcchk port map(opfipcin=>instoutsig(63 downto 58), opipcout=>ipc);
instmuxcomp:mux_inst port map(a=>fm_inst_reg_EX,b=>fm_inst_reg_ID,s=>sinstmux,y=>muxinstaddr);

end architecture ifidstregbeh;

-- Incr PC generation

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ipcchk is
port(opfipcin: in std_logic_vector(5 downto 0);
    opipcout: out std_logic);
end entity ipcchk;

architecture ipcchkbeh of ipcchk is
begin

process(opfipcin) is
begin
if(opfipcin = "000010") then
opipcout <= '0';
else
opipcout <= '1';
end if;
end process;
end architecture ipcchkbeh;

--MUX for choosing inst reg addr

library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity mux_inst is
port(a: in STD_LOGIC_VECTOR (15 downto 0);
   b: in STD_LOGIC_VECTOR (15 downto 0);
   s: in STD_LOGIC;
   y: out STD_LOGIC_VECTOR (15 downto 0) );
end entity mux_inst;

architecture mux_inst_arch of mux_inst is
begin

process (a, b, s)
begin
if ( s = '0') then
y <= a;
else y <= b;
end if;
end process;
end architecture mux_inst_arch;

-- IF pipe stage


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity if_pipe is
generic(N: positive := 64;
         Addr: positive :=16); -- 16
port(jump_in, branch_in, retin, macctrlr, oflow, fmmacsig: in std_logic;
   fm_inst_reg, fm_mac_ctrlr: in std_logic_vector(Addr-1 downto 0);
   clk, clr, we_im, clock: in std_logic;
   instrin: in std_logic_vector(N-1 downto 0);
   NOP_out: out std_logic;
   inst_out: out std_logic_vector(N-1 downto 0);
   pc_out: out std_logic_vector(Addr-1 downto 0));
end entity if_pipe;

architecture if_pipe_beh of if_pipe is
--reg below pc
component reg0 is
port(in_fm_pc: in std_logic_vector(15 downto 0);
   jump_in, branch_in, clr, clk: in std_logic;
   out_to_pc: out std_logic_vector(15 downto 0));
end component reg0;
--Mux before pc
component mux_bf_pc is
generic(Addr: positive := 16);
port ( fm_mac_ctrlr, fm_inst_reg, fm_reg, incdpc: in std_logic_vector (Addr-1 downto 0);
     jump_in, branch_in, retin, macctrlr, oflow, incpc: in std_logic;
     pcaddr: out std_logic_vector (Addr-1 downto 0) );
end component mux_bf_pc;
-- Instruction memory
component INSTMEM is
port(clk, we, en, rst: in std_logic;
   addr: in std_logic_vector(7 downto 0);
```

```
    inst_in: in std_logic_vector(63 downto 0);
    inst_out: out std_logic_vector(63 downto 0));
end component INSTMEM;
-- program counter
component pc is
port(clk,clr,lpc, incpc: in std_logic;
    in_addr: in std_logic_vector(Addr-1 downto 0);
    out_addr: out std_logic_vector(Addr-1 downto 0));
end component pc;
-- IF stage signals
component ifsigfmbr is
port(branchsig, jsig, rsig, macctrlr, fmmacsig: in std_logic;
    lpc_out, NOP_out, incpcout: out std_logic);
end component ifsigfmbr;

signal sigreg, sigincrpc, siginpc, sigoutpc: std_logic_vector(Addr-1 downto 0);
signal incrpcsig, lpc, oneen: std_logic;

begin

pc_out <= sigoutpc;
oneen <= '1';

muxpc: mux_bf_pc port map(fm_mac_ctrlr=>fm_mac_ctrlr, fm_inst_reg=>fm_inst_reg, fm_reg=>sigreg,
incdpc=>sigoutpc, jump_in=>jump_in, branch_in=>branch_in, retin=>retin, macctrlr=>macctrlr,
oflow=>oflow, incpc=>incrpcsig, pcaddr=>siginpc);

pctr: pc port map(clk=>clk, clr=>clr, lpc=>lpc, incpc=>incrpcsig, in_addr=>siginpc, out_addr=>sigoutpc);
pcreg: reg0 port map(in_fm_pc=>sigoutpc, jump_in=>jump_in, branch_in=>branch_in, clr=>clr, clk=>clk,
out_to_pc=>sigreg);

instrmemnew: INSTMEM port map(clk=>clock, we=>we_im, en=>oneen, rst=>clr, addr=>sigoutpc(7
downto 0), inst_in=>instrin, inst_out=>inst_out);

IFsigs: ifsigfmbr port map(branchsig=>branch_in, jsig=>jump_in, rsig=>retin, macctrlr=>macctrlr,
fmmacsig=>fmmacsig, lpc_out=>lpc, NOP_out=>NOP_out, incpcout=>incrpcsig);

end architecture if_pipe_beh;

-- register below pc

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity reg0 is
port(in_fm_pc: in std_logic_vector(15 downto 0);
    jump_in, branch_in, clr, clk: in std_logic;
    out_to_pc: out std_logic_vector(15 downto 0));
end entity reg0;

architecture reg_beh of reg0 is
signal lreg: std_logic;
signal cl: std_logic_vector(1 downto 0);
signal out_to_pcs: std_logic_vector(15 downto 0);
```

```
begin

lreg <= jump_in or branch_in;
cl <= clr & lreg;

process(clk, cl, in_fm_pc, out_to_pcs) is
begin
if (rising_edge(clk)) then
case cl is
when "10" => out_to_pcs <= (others => '0');
when "11" => out_to_pcs <= (others => '0');
when "01" => out_to_pcs <= in_fm_pc;
when "00" => out_to_pcs <= out_to_pcs;
when others => null;
end case;
end if;
out_to_pc <= out_to_pcs;
end process;
end architecture reg_beh;

-- Mux before PC

library IEEE;
use IEEE.std_logic_1164.all;

entity mux_bf_pc is
generic(Addr: positive := 16);
port ( fm_mac_ctrlr, fm_inst_reg, fm_reg, incdpc: in std_logic_vector (Addr-1 downto 0);
     jump_in, branch_in, retin, macctrlr, oflow, incpc: in std_logic;
     pcaddr: out std_logic_vector (Addr-1 downto 0) );
end entity mux_bf_pc;

architecture mux_arch of mux_bf_pc is
signal jb_ret_mac: std_logic_vector(4 downto 0);
signal jorb_in: std_logic;
signal pcsig: std_logic_vector(Addr-1 downto 0);
begin

jorb_in <= jump_in or branch_in;
jb_ret_mac <= jorb_in & retin & macctrlr & oflow & incpc;

process (fm_mac_ctrlr, fm_inst_reg, fm_reg, jb_ret_mac, pcsig, incdpc) is
begin

case jb_ret_mac is

when "00000" => pcsig <= (others => '0');
when "00001" => pcsig <= incdpc;
when "00010" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "00011" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "00100" => pcsig <= fm_mac_ctrlr;
when "00101" => pcsig <= fm_mac_ctrlr;
when "00110" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
```

```
when "00111" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "01000" => pcsig <= fm_reg;
when "01001" => pcsig <= incdpc;
when "01010" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "01011" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "01100" => pcsig <= fm_mac_ctrlr;
when "01101" => pcsig <= fm_mac_ctrlr;
when "01110" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "01111" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "10000" => pcsig <= fm_inst_reg;
when "10001" => pcsig <= fm_inst_reg;
when "10010" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "10011" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "10100" => pcsig <= fm_inst_reg;
when "10101" => pcsig <= fm_inst_reg;
when "10110" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "10111" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "11000" => pcsig <= fm_inst_reg;
when "11001" => pcsig <= fm_inst_reg;
when "11010" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "11011" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "11100" => pcsig <= fm_inst_reg;
when "11101" => pcsig <= fm_inst_reg;
when "11110" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when "11111" => pcsig <= "0000100000000000"; --"10000"; --"0000100000000000"; -- Overflow
exception, this Address has abort and out instructions
when others => null;
end case;
pcaddr <= pcsig;
end process;
end architecture mux_arch;

-- Full Instruction Memory Design -Initialised for 'RCOLLECT' with the inital PUT


library IEEE;
use IEEE.std_logic_1164.all;

--synopsys translate_off;
library unisim;
use unisim.vcomponents.all;
--synopsys translate_on;
entity INSTMEM is
port(clk, we, en, rst: in std_logic;
    addr: in std_logic_vector(7 downto 0);
    inst_in: in std_logic_vector(63 downto 0);
    inst_out: out std_logic_vector(63 downto 0));
end entity INSTMEM;

architecture behavioural of INSTMEM is

component RAMB4_S16 is
port(ADDR: in std_logic_vector(7 downto 0);
    CLK: in std_logic;
    DI: in std_logic_vector(15 downto 0);
    DO: out std_logic_vector(15 downto 0);
    EN, RST, WE: in std_logic);
end component RAMB4_S16;

attribute INIT_00: string;
attribute INIT_01: string;
attribute INIT_02: string;
attribute INIT_03: string;
attribute INIT_04: string;
attribute INIT_05: string;
attribute INIT_06: string;
attribute INIT_07: string;
attribute INIT_08: string;
attribute INIT_09: string;
attribute INIT_0A: string;
attribute INIT_0B: string;
attribute INIT_0C: string;
attribute INIT_0D: string;
attribute INIT_0E: string;
attribute INIT_0F: string;

attribute INIT_00 of Instram0 : label is
"0000014000000124000000000C000000000104000000000000000000000400000000";
attribute INIT_01 of Instram0 : label is
"034000000AC0000001C000001240000000000000006C00000000002C000000940";
attribute INIT_02 of Instram0 : label is
"000124000000000000340000006C00580000000000002C000000B40000003C00000";
attribute INIT_03 of Instram0 : label is
"000040000001240000000000000000010C0000000240000000000D4000000000";
attribute INIT_04 of Instram0 : label is
"00000340000002C0000000000000000000F80000012400000000000000000001300";
attribute INIT_05 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_0C of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram1 : label is
"8000000000000008000000000000000000000000810001000100010000000000";
attribute INIT_01 of Instram1 : label is
"0100000000080000000000000000000800001000000001000100010000000000";
attribute INIT_02 of Instram1 : label is
"0000000000080000100000000000001008000010000000000000800000800100";
attribute INIT_03 of Instram1 : label is
"0100010000000000080000100000000000800000000000000000000001000100";
attribute INIT_04 of Instram1 : label is
"0000000000000000000000000000000000000000000000008000000000000000";
attribute INIT_05 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram2 : label is
"008200A000000410060000000000000000000041000100601800000000000000";
attribute INIT_01 of Instram2 : label is
"00000000000000C300E000000000082000220002000200000002000000000000";
attribute INIT_02 of Instram2 : label is
"0000000000C30003000000000000000000020002000000000000480300000003";
attribute INIT_03 of Instram2 : label is
"0004000000000000010400040004000000001040120000000000580000020001";
attribute INIT_04 of Instram2 : label is
"0000003000000000000000000000000000000000000104000400000000007000";
attribute INIT_05 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_07 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram3 : label is
"7800540000008000780054000000000084007C001C051C0524A4208000000400";
attribute INIT_01 of Instram3 : label is
"55000000800078005400000084007C001C0734E5600638C51CA0548000008000";
attribute INIT_02 of Instram3 : label is
"000084007C001C08550000007000 5C041CA01C005480000070000008 54001D20";
attribute INIT_03 of Instram3 : label is
"1DC055A0000084007C001C0C2D80000080007800540000001400600A1D601D40";
attribute INIT_04 of Instram3 : label is
"080058000C00580300008008800000007000000084007C001C0000001400640D";
attribute INIT_05 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

begin

Instram0: RAMB4_S16
--synopsys translate_off
```

```
GENERIC MAP (
INIT_00 => X"0000140000012400000000C0000000001040000000000000000000004000000000",
INIT_01 => X"034000000AC0000001C000001240000000000000006C0000000002C000000940",
INIT_02 => X"0000124000000000003400000006C005800000000002C000000B40000003C00000",
INIT_03 => X"0000040000001240000000000000000010C000000240000000000D4000000000",
INIT_04 => X"00000340000002C000000000000000000F8000001240000000000000000001300",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(15 downto 0), DO=>inst_out(15 downto 0), EN=>en,
RST=>rst, WE=>we);

Instram1: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"8000000000000000800000000000000000000000081000100010001000000000",
INIT_01 => X"0100000000008000000000000000000080000100000001000100010000000000",
INIT_02 => X"0000000000080000100000000000000010080000100000000008000000800100",
INIT_03 => X"0100010000000000000080000100000000008000000000000000000001000100",
INIT_04 => X"0000000000000000000000000000000000000000000080000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(31 downto 16), DO=>inst_out(31 downto 16), EN=>en,
RST=>rst, WE=>we);

Instram2: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"008200A000000000041006000000000000000041000100601800000000000000",
INIT_01 => X"00000000000000C300E000000000008200022000200020000002000000000000",
INIT_02 => X"0000000000C30003000000000000000000000020002000000000000480300000003",
INIT_03 => X"0004000000000000104000400040000000001040120000000000580000020001",
INIT_04 => X"0000000300000000000000000000000000000000000010400040000000007000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(47 downto 32), DO=>inst_out(47 downto 32), EN=>en,
RST=>rst, WE=>we);

Instram3: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"7800540000008000780054000000000084007C001C051C0524A4208000000400",
INIT_01 => X"55000000800078005400000084007C001C0734E5600638C51CA0548000008000",
INIT_02 => X"000084007C001C085500000070005C041CA01C00548000007000000854001D20",
INIT_03 => X"1DC055A0000084007C001C0C2D80000080007800540000001400600A1D601D40",
INIT_04 => X"080058000C00580300000080088000000700000084007C001C0000001400640D",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(63 downto 48), DO=>inst_out(63 downto 48), EN=>en,
RST=>rst, WE=>we);
end architecture behavioural;

-- Program counter

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity pc is
port(clk,clr,lpc, incpc: in std_logic;
    in_addr: in std_logic_vector(15 downto 0);
    out_addr: out std_logic_vector(15 downto 0));
end entity pc;

architecture behavioral of pc is
signal clipc: std_logic_vector(2 downto 0);
signal out_addrs: std_logic_vector(15 downto 0);
begin

clipc <= clr & lpc & incpc;
```

```
process(clk, clipc, in_addr, out_addrs) is
begin
if (rising_edge(clk)) then
case clipc is
when "110" => out_addrs <= (others => '0');
when "111" => out_addrs <= (others => '0');
when "101" => out_addrs <= (others => '0');
when "100" => out_addrs <= (others => '0');
when "010" => out_addrs <= in_addr;
when "001" => out_addrs <= in_addr + 1;
when "011" => out_addrs <= in_addr + 1;
when "000" => out_addrs <= out_addrs;
when others => null;
end case;
end if;

out_addr <= out_addrs;
end process;
end architecture behavioral;

-- IF stage signals

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ifsigfmbr is
port(branchsig, jsig, rsig, macctrlr, fmmacsig: in std_logic;
    lpc_out, NOP_out, incpcout: out std_logic);
end entity ifsigfmbr;

architecture ifsigfmbr_beh of ifsigfmbr is
signal bjr: std_logic;
signal bf: std_logic_vector(1 downto 0);
begin

bjr <= branchsig or jsig or rsig or macctrlr;
bf <= bjr & fmmacsig;

process(bf) is
begin
case bf is

when "00" =>
lpc_out <= '0';
NOP_out <= '0';
incpcout <= '0';

when "01" =>
lpc_out <= '1';
NOP_out <= '0';
incpcout <= '1';

when "10" =>
lpc_out <= '1';
```

```
NOP_out <= '1';
incpcout <= '0';

when "11" =>
lpc_out <= '1';
NOP_out <= '1';
incpcout <= '0';

when others =>
lpc_out <= '0';
NOP_out <= '0';
incpcout <= '0';
end case;
end process;
end architecture ifsigfmbr_beh;

-- IF-ID stage register

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ifidreg is
port(clr, clk: in std_logic;
    instrin: in std_logic_vector(63 downto 0);
    instrouttoid: out std_logic_vector(63 downto 0));
end entity ifidreg;

architecture ifidreg_beh of ifidreg is
begin

rpr:process(clk, clr, instrin) is
begin
if(falling_edge(clk)) then
case clr is
when '1' =>
instrouttoid <= (others => '0');
when '0' =>
instrouttoid <= instrin;
when others => null;
end case;
end if;
end process rpr;
end architecture ifidreg_beh;
```

3. ID STAGE

```
-- ID/ETM stage components and register

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
```

```
entity idstreg is
generic(N: positive := 64;
        Addr: positive := 16);
port(inst_in: in std_logic_vector(N-1 downto 0);
--cfg_in, bitmapin: in std_logic_vector(N-1 downto 0);
  WB_write_data: in std_logic_vector(N-1 downto 0);
  loc: in std_logic_vector(2 downto 0);
  ffpin: in std_logic_vector(7 downto 0);
  clk, NOP_in, ID_flush_BR, regwr_sig, trw, vrw, jmpin, retin, lmfmex: in std_logic;
  morfmex: in std_logic_vector(5 downto 0);
  TRDstin, VRDstin, RDstin: in std_logic_vector(4 downto 0);
  IDFout: out std_logic;
  WB_ctrl_out: out std_logic_vector(3 downto 0);
  EX_ctrl_out: out std_logic_vector(12 downto 0);
  PKT_ctrl_out: out std_logic_vector(6 downto 0);
  GPR_read1_out, GPR_read2_out, sign_ext_out: out std_logic_vector(N-1 downto 0);
  TR_read_out_ID, VR_read_out_ID: out std_logic_vector(N-1 downto 0);
  Br_Addr_out, PKT_Offset_out: out std_logic_vector(Addr-1 downto 0);
  shamt_out: out std_logic_vector(5 downto 0);
  lmor_out, TRD_out, VRD_out, jumps, rets: out std_logic;
  ocr_val_out_id, aer_val_out_id: out std_logic_vector(7 downto 0);
  opcodeexout: out std_logic_vector(5 downto 0);
  ctrlsigsoutID: out std_logic_vector(24 downto 0);
  wrdataout: out std_logic_vector(N-1 downto 0);
  RS1_out, RS2_out, RD_out, TR_out, VR_out: out std_logic_vector(4 downto 0));
end entity idstreg;

architecture idstreg_beh of idstreg is
-- ID stage component
component id_stage is
generic(N: positive := 64;
        Addr: positive := 16);
port(inst_in: in std_logic_vector(N-1 downto 0);
  WB_write_data: in std_logic_vector(N-1 downto 0);
  loc: in std_logic_vector(2 downto 0);
  ffpin: in std_logic_vector(7 downto 0);
  clk, NOP_in, ID_flush_BR, regwr_sig, jmpin, retin, lmfmex: in std_logic;
  morfmex: in std_logic_vector(5 downto 0);
  trw, vrw: in std_logic;
  RDstin: in std_logic_vector(4 downto 0);
  TRDstin, VRDstin: in std_logic_vector(4 downto 0);
  opcodeout: out std_logic_vector(5 downto 0);
  ID_Flush: out std_logic;
  WB_ctrl_out: out std_logic_vector(3 downto 0);
  EX_ctrl_out: out std_logic_vector(12 downto 0);
  PKT_ctrl_out: out std_logic_vector(6 downto 0);
  GPR_read1_out, GPR_read2_out, sign_ext_out: out std_logic_vector(N-1 downto 0);
  TR_read_out, VR_read_out: out std_logic_vector(N-1 downto 0);
  Br_Addr, PKT_Offset: out std_logic_vector(Addr-1 downto 0);
  shamt_out: out std_logic_vector(5 downto 0);
  lmor_out, TRD, VRD, jump, ret: out std_logic;
  ocr_val_stout, aer_val_out: out std_logic_vector(7 downto 0);
  ctrlsigsout: out std_logic_vector(24 downto 0);
  wrdataout: out std_logic_vector(N-1 downto 0);
  RS1_out, RS2_out, RD_out, TR_out, VR_out: out std_logic_vector(4 downto 0));
end component id_stage;
```

```
-- ID/ETM register component
component ess_idexreg is
generic(N: positive := 64;
        Addr: positive := 16);
port(clk, ID_Flush: in std_logic;
   ctrlin: in std_logic_vector(24 downto 0);
   WB_in: in std_logic_vector(3 downto 0);
   EX_in: in std_logic_vector(12 downto 0);
   PKT_in: in std_logic_vector(6 downto 0);
   GPR_read1_in, GPR_read2_in, sign_ext_in: in std_logic_vector(N-1 downto 0);
   TR_read_in, VR_read_in: in std_logic_vector(N-1 downto 0);
   Br_Addr_in, PKT_Offset_in: in std_logic_vector(Addr-1 downto 0);
   shamt_in: in std_logic_vector(5 downto 0);
   lmor_in, jin_id, rin_id: in std_logic;
   ocr_in_id, aer_in_id: in std_logic_vector(7 downto 0);
   RS1_in, RS2_in, RD_in, TR_in, VR_in: in std_logic_vector(4 downto 0);
   opcodein: in std_logic_vector(5 downto 0);
   opcodeexout: out std_logic_vector(5 downto 0);
   ctrlout: out std_logic_vector(24 downto 0);
   WB_out: out std_logic_vector(3 downto 0);
   EX_out: out std_logic_vector(12 downto 0);
   PKT_out: out std_logic_vector(6 downto 0);
   GPR_read1_out, GPR_read2_out, sign_ext_out: out std_logic_vector(N-1 downto 0);
   TR_read_out_ID, VR_read_out_ID: out std_logic_vector(N-1 downto 0);
   Br_Addr_out, PKT_Offset_out: out std_logic_vector(Addr-1 downto 0);
   shamt_out: out std_logic_vector(5 downto 0);
   lmor_out, TRD_out, VRD_out, jout_id, rout_id: out std_logic;
   ocr_out_id, aer_out_id: out std_logic_vector(7 downto 0);
   RS1_out, RS2_out, RD_out, TR_out, VR_out: out std_logic_vector(4 downto 0));
end component ess_idexreg;

-- JBR component
component jbrchk is
port(clk, IDFin: in std_logic;
   IDF_out1: out std_logic);
end component jbrchk;

-- signals declaration
signal IDFs, IDFs1, IDFs2: std_logic;
signal WBs: std_logic_vector(3 downto 0);
signal EXs: std_logic_vector(12 downto 0);
signal PKTs: std_logic_vector(6 downto 0);
signal TRrs, VRrs, GPR1s, GPR2s, signs: std_logic_vector(N-1 downto 0);
signal BrAddrs, PKTOs: std_logic_vector(Addr-1 downto 0);
signal shamts: std_logic_vector(5 downto 0);
signal lmors, TRDs, VRDs, js, rs: std_logic;
signal ocrs, aers: std_logic_vector(7 downto 0);
signal RS1s, RS2s, RDs, TRns, VRns: std_logic_vector(4 downto 0);
signal ops: std_logic_vector(5 downto 0);
signal ctrls: std_logic_vector(24 downto 0);

begin

IDFout <= IDFs2;
IDFs2 <= IDFs or IDFs1;
```

```
idstagecomp: id_stage port map(inst_in=>inst_in, WB_write_data=>WB_write_data, loc=>loc,
ffpin=>ffpin, clk=>clk, NOP_in=>NOP_in, ID_flush_BR=>ID_flush_BR, regwr_sig=>regwr_sig,
jmpin=>jmpin, retin=>retin, lmfmex=>lmfmex, morfmex=>morfmex, trw=>trw, vrw=>vrw,
RDstin=>RDstin, TRDstin=>TRDstin, VRDstin=>VRDstin, opcodeout=>ops, ID_Flush=>IDFs,
WB_ctrl_out=>WBs, EX_ctrl_out=>EXs, PKT_ctrl_out=>PKTs, GPR_read1_out=>GPR1s,
GPR_read2_out=>GPR2s, sign_ext_out=>signs, TR_read_out=>TRrs, VR_read_out=>VRrs,
Br_Addr=>BrAddrs, PKT_Offset=>PKTOs, shamt_out=>shamts, lmor_out=>lmors, TRD=>TRDs,
VRD=>VRDs, jump=>js, ret=>rs, ocr_val_stout=>ocrs, aer_val_out=>aers, ctrlsigsout=>ctrls,
wrdataout=>wrdataout, RS1_out=>RS1s, RS2_out=>RS2s, RD_out=>RDs, TR_out=>TRns,
VR_out=>VRns);

idexregcomp: ess_idexreg port map(clk=>clk, ID_Flush=>IDFs2, ctrlin=>ctrls, WB_in=>WBs,
EX_in=>EXs, PKT_in=>PKTs, GPR_read1_in=>GPR1s, GPR_read2_in=>GPR2s, sign_ext_in=>signs,
TR_read_in=>TRrs, VR_read_in=>VRrs, Br_Addr_in=>BRAddrs, PKT_Offset_in=>PKTOs,
shamt_in=>shamts, lmor_in=>lmors, jin_id=>js, rin_id=>rs, ocr_in_id=>ocrs, aer_in_id=>aers,
RS1_in=>RS1s, RS2_in=>RS2s, RD_in=>RDs, TR_in=>TRns, VR_in=>VRns, opcodein=>ops,
opcodeexout=>opcodeexout,  ctrlout=>ctrlsigsoutID, WB_out=>WB_ctrl_out, EX_out=>EX_ctrl_out,
PKT_out=>PKT_ctrl_out, GPR_read1_out=>GPR_read1_out, GPR_read2_out=>GPR_read2_out,
sign_ext_out=>sign_ext_out, TR_read_out_ID=>TR_read_out_ID, VR_read_out_ID=>VR_read_out_ID,
Br_Addr_out=>Br_Addr_out, PKT_Offset_out=>PKT_Offset_out, shamt_out=>shamt_out,
lmor_out=>lmor_out, TRD_out=>TRD_out, VRD_out=>VRD_out, jout_id=>jumps, rout_id=>rets,
ocr_out_id=>ocr_val_out_id, aer_out_id=>aer_val_out_id, RS1_out=>RS1_out, RS2_out=>RS2_out,
RD_out=>RD_out, TR_out=>TR_out, VR_out=>VR_out);
jbrchkcomp: jbrchk port map(clk=>clk, IDFin=>IDFs, IDF_out1=>IDFs1);

end architecture idstreg_beh;

--Individual components

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity id_stage is
generic(N: positive := 64;
        Addr: positive := 16);
port(inst_in: in std_logic_vector(N-1 downto 0);
--cfg_in, bitmapin: in std_logic_vector(N-1 downto 0);
    WB_write_data: in std_logic_vector(N-1 downto 0);
    loc: in std_logic_vector(2 downto 0);
    ffpin: in std_logic_vector(7 downto 0);
    clk, NOP_in, ID_flush_BR, regwr_sig, jmpin, retin, lmfmex: in std_logic;
    morfmex: in std_logic_vector(5 downto 0);
    trw, vrw: in std_logic;
    RDstin: in std_logic_vector(4 downto 0);
    TRDstin, VRDstin: in std_logic_vector(4 downto 0);
    opcodeout: out std_logic_vector(5 downto 0);
    ID_Flush: out std_logic;
    WB_ctrl_out: out std_logic_vector(3 downto 0);
    EX_ctrl_out: out std_logic_vector(12 downto 0);
    PKT_ctrl_out: out std_logic_vector(6 downto 0);
    GPR_read1_out, GPR_read2_out, sign_ext_out: out std_logic_vector(N-1 downto 0);
    TR_read_out, VR_read_out: out std_logic_vector(N-1 downto 0);
    Br_Addr, PKT_Offset: out std_logic_vector(Addr-1 downto 0);
    shamt_out: out std_logic_vector(5 downto 0);
    lmor_out, TRD, VRD, jump, ret: out std_logic;
    ocr_val_stout, aer_val_out: out std_logic_vector(7 downto 0);
    ctrlsigsout: out std_logic_vector(24 downto 0);
    wrdataout: out std_logic_vector(N-1 downto 0);
    RS1_out, RS2_out, RD_out, TR_out, VR_out: out std_logic_vector(4 downto 0));
end entity id_stage;

architecture id_stage_beh of id_stage is
--components
--tag and val reg file
component tagregfile is
port(TRNUMS, TRNUMD: in std_logic_vector(4 downto 0);
    tag_in: in std_logic_vector(63 downto 0);
    clk, tr_write: in std_logic;
    tag_out: out std_logic_vector(63 downto 0));
end component tagregfile;
--controller
component cntunit0 is
port(opcode: in std_logic_vector(5 downto 0);
    loc: in std_logic_vector(2 downto 0);
    ffpin: in std_logic_vector(7 downto 0);
    ocr_val, aer_val: out std_logic_vector(7 downto 0); -- for 8 bit output code register
    ctrlsigs: out std_logic_vector(24 downto 0));
end component cntunit0;
--GPR file
component regfile is
port(RD, RS1, RS2: in std_logic_vector(4 downto 0);
--    cfg_in, bitmapin: in std_logic_vector(N-1 downto 0);
    writedata: in std_logic_vector(63 downto 0);
    clk, regwrite, regread: in std_logic;
--    rdwrdataout: out std_logic_vector(63 downto 0); (need to have later)
    readdata1, readdata2: out std_logic_vector(63 downto 0));
end component regfile;
-- SIGN EXT UNIT COMP
component signext is
generic(N: positive := 64;
            imm: positive := 16);
port(immval: in std_logic_vector(imm-1 downto 0);
    sign: in std_logic;
    extdval: out std_logic_vector(N-1 downto 0));
end component signext;
-- extra comp
component idextra is
generic(N: positive := 64);
port(regw, trwr, vrwr: in std_logic;
    wrregin, wrtagin, wrvalin: in std_logic_vector(N-1 downto 0);
    wrdataout: out std_logic_vector(N-1 downto 0));
end component idextra;
-- opcode for controller
component mux_ct is
port (n_op, lm_op: in STD_LOGIC_VECTOR (5 downto 0);
    lm: in STD_LOGIC;
    optoct: out STD_LOGIC_VECTOR (5 downto 0) );
end component mux_ct;

signal opcodesig, optoctsig: std_logic_vector(5 downto 0);
signal RS1_sig, RS2_sig, TR_sig, VR_sig: std_logic_vector(4 downto 0);
signal immsig: std_logic_vector(Addr-1 downto 0);
signal temp_ctrl_sigs: std_logic_vector(24 downto 0);
signal regrdsig: std_logic;
signal wrtagsig, wrvalsig, wrregsig: std_logic_vector(N-1 downto 0);
begin

opcodesig <= inst_in(63 downto 58);
opcodeout <= optoctsig;
ctrlsigsout <= temp_ctrl_sigs;
RS1_sig <= inst_in(52 downto 48);
RS2_sig <= inst_in(47 downto 43);
TR_sig <= inst_in(42 downto 38);
VR_sig <= inst_in(36 downto 32);
immsig <= inst_in(21 downto 6);
regrdsig <= temp_ctrl_sigs(22);

ID_Flush <= ID_flush_BR or NOP_in or jmpin or retin;
jump <= temp_ctrl_sigs(21);
ret <= temp_ctrl_sigs(20);
WB_ctrl_out <= inst_in(37) & inst_in(31) & temp_ctrl_sigs(5) & inst_in(24);
EX_ctrl_out <= temp_ctrl_sigs(19 downto 17) & temp_ctrl_sigs(14 downto 6) & temp_ctrl_sigs(1);
PKT_ctrl_out <= temp_ctrl_sigs(24 downto 23) & temp_ctrl_sigs(16 downto 15) & temp_ctrl_sigs(3
downto 2) & temp_ctrl_sigs(0);
Br_Addr <= inst_in(21 downto 6);
PKT_Offset <= inst_in(21 downto 6);
shamt_out <= inst_in(5 downto 0);
lmor_out <= inst_in(23);
TRD <= inst_in(37);
VRD <= inst_in(31);
RD_out <= inst_in(57 downto 53);
RS1_out <= inst_in(52 downto 48);
RS2_out <= inst_in(47 downto 43);
TR_out <= inst_in(42 downto 38);
VR_out <= inst_in(36 downto 32);

GPR_read1_out <= wrregsig;
TR_read_out <= wrtagsig;
VR_read_out <= wrvalsig;

GPRfile: regfile port map(RD=>RDstin,RS1=>RS1_sig,RS2=>RS2_sig,writedata=>WB_write_data,
clk=>clk, regwrite=>regwr_sig, regread=>regrdsig, readdata1=>wrregsig, readdata2=>GPR_read2_out);

TRfile: tagregfile port
map(TRNUMS=>TR_sig,TRNUMD=>TRDstin,tag_in=>WB_write_data,clk=>clk,tr_write=>trw,tag_out=
>wrtagsig);

VRfile: tagregfile port
map(TRNUMS=>VR_sig,TRNUMD=>VRDstin,tag_in=>WB_write_data,clk=>clk,tr_write=>vrw,tag_out
=>wrvalsig);

Control: cntunit0 port map(opcode=>optoctsig, loc=>loc, ffpin=>ffpin, ocr_val=>ocr_val_stout,
aer_val=>aer_val_out, ctrlsigs=>temp_ctrl_sigs);

signextunit: signext port map(immval=>immsig, sign=>inst_in(22), extdval=>sign_ext_out);
```

```
extraunit: idextra port map(regw=>regwr_sig, trwr=>trw, vrwr=>vrw, wrregin=>wrregsig,
wrtagin=>wrtagsig, wrvalin=>wrvalsig, wrdataout=>wrdataout);

muxctcomp: mux_ct port map(n_op=>opcodesig, lm_op=>morfmex, lm=>lmfmex, optoct=>optoctsig);

end architecture id_stage_beh;

--Individual Components
-- New Design using block ram- TAG/VAL reg file

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity tagregfile is
port(TRNUMS, TRNUMD: in std_logic_vector(4 downto 0);
    tag_in: in std_logic_vector(63 downto 0);
    clk, tr_write: in std_logic;
    tag_out: out std_logic_vector(63 downto 0));
end entity tagregfile;

architecture tagregfile_beh of tagregfile is
--components
component tag_block is
port(addr: in std_logic_vector(4 downto 0);
    din: in std_logic_vector(31 downto 0);
    dout: out std_logic_vector(31 downto 0);
    clk: in std_logic;
    wtr: in std_logic);
end component tag_block;
component muxreg1 is
port(SRC, DST: in std_logic_vector(4 downto 0);
    s_wr: in std_logic;
    RSRD: out std_logic_vector(4 downto 0));
end component muxreg1;

--signals
signal regaddress: std_logic_vector(4 downto 0);

begin
muxreg_comp_tag: muxreg1 port map(SRC=>TRNUMS, DST=>TRNUMD, s_wr=>tr_write,
RSRD=>regaddress);
tag_blk_comp0: tag_block port map(addr=>regaddress, din=>tag_in(63 downto 32), dout=>tag_out(63
downto 32), clk=>clk, wtr=>tr_write);
tag_blk_comp1: tag_block port map(addr=>regaddress, din=>tag_in(31 downto 0), dout=>tag_out(31
downto 0), clk=>clk, wtr=>tr_write);

end architecture tagregfile_beh;

--Individual Components
-- Tag reg Design using Block RAM

library IEEE;
use IEEE.std_logic_1164.all;
entity tag_block is
```

```
port(addr: in std_logic_vector(4 downto 0);
    din: in std_logic_vector(31 downto 0);
    dout: out std_logic_vector(31 downto 0);
    clk: in std_logic;
    wtr: in std_logic);
end entity tag_block;

architecture tag_behave of tag_block is

component RAMB4_S16_S16 is
port(ADDRA, ADDRB: in std_logic_vector(7 downto 0);
    CLKA, CLKB: in std_logic;
    DIA, DIB: in std_logic_vector(15 downto 0);
    DOA, DOB: out std_logic_vector(15 downto 0);
    ENA, ENB, RSTA, RSTB, WEA, WEB: in std_logic);
end component RAMB4_S16_S16;

signal vcc, gnd: std_logic;
signal addr_ablk, addr_bblk: std_logic_vector(7 downto 0);
begin

vcc <= '1';
gnd <= '0';

addr_ablk <= "00" & addr & vcc;
addr_bblk <= "00" & addr & gnd;

tagram0: RAMB4_S16_S16 port map(ADDRA=>addr_ablk, ADDRB=>addr_bblk, CLKA=>clk,
CLKB=>clk, DIA=>din(31 downto 16), DIB=>din(15 downto 0), DOA=>dout(31 downto 16),
DOB=>dout(15 downto 0), ENA=>vcc, ENB=>vcc, RSTA=>gnd, RSTB=>gnd, WEA=>wtr, WEB=>wtr);

end architecture tag_behave;

-- MUX fro choosing btw RS1 and RD

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity muxreg1 is
port(SRC, DST: in std_logic_vector(4 downto 0);
    s_wr: in std_logic;
    RSRD: out std_logic_vector(4 downto 0));
end entity muxreg1;

architecture muxreg1_beh of muxreg1 is
begin

process(SRC, DST, s_wr) is
begin
case s_wr is
when '0' => RSRD <= SRC;
when '1' => RSRD <= DST;
when others => null;
end case;
end process;
end architecture muxreg1_beh;

-- GPR REG FILE
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity regfile is
port(RD, RS1, RS2: in std_logic_vector(4 downto 0);
--    cfg_in, bitmapin: in std_logic_vector(N-1 downto 0);
    writedata: in std_logic_vector(63 downto 0);
    clk, regwrite, regread: in std_logic;
--    rdwrdataout: out std_logic_vector(63 downto 0); (need to have later)
    readdata1, readdata2: out std_logic_vector(63 downto 0));
end entity regfile;

architecture reg_beh of regfile is
-- components
component blockram is
port(addr1, addr2: in std_logic_vector(4 downto 0); -- for RS1 and RS2, need to have mux for choosing
either RS1 or RD
    din1, din2: in std_logic_vector(15 downto 0);
    dout1, dout2: out std_logic_vector(15 downto 0);
    clk: in std_logic;
    wr1, enr1, enr2: in std_logic); --wr1 - write is always in port 1, enr1, enr2 - for reg reads from 2 ports
end component blockram;

component muxreg is
port(SRC, DST: in std_logic_vector(4 downto 0);
    s_wr: in std_logic;
    RSRD: out std_logic_vector(4 downto 0));
end component muxreg;

--signals
signal regaddr1: std_logic_vector(4 downto 0);
signal en1, en2: std_logic;

begin

en1 <= regread or regwrite;
en2 <= regread;

muxreg_comp: muxreg port map(SRC=>RS1, DST=>RD, s_wr=>regwrite, RSRD=>regaddr1);

bram_comp1: blockram port map(addr1=>regaddr1, addr2=>RS2, din1=>writedata(63 downto 48),
din2=>writedata(63 downto 48), dout1=>readdata1(63 downto 48), dout2=>readdata2(63 downto 48),
clk=>clk, wr1=>regwrite, enr1=>en1, enr2=>en2);

bram_comp2: blockram port map(addr1=>regaddr1, addr2=>RS2, din1=>writedata(47 downto 32),
din2=>writedata(47 downto 32), dout1=>readdata1(47 downto 32), dout2=>readdata2(47 downto 32),
clk=>clk, wr1=>regwrite, enr1=>en1, enr2=>en2);
bram_comp3: blockram port map(addr1=>regaddr1, addr2=>RS2, din1=>writedata(31 downto 16),
din2=>writedata(31 downto 16), dout1=>readdata1(31 downto 16), dout2=>readdata2(31 downto 16),
clk=>clk, wr1=>regwrite, enr1=>en1, enr2=>en2);
```

```
bram_comp4: blockram port map(addr1=>regaddr1, addr2=>RS2, din1=>writedata(15 downto 0),
din2=>writedata(15 downto 0), dout1=>readdata1(15 downto 0), dout2=>readdata2(15 downto 0),
clk=>clk, wr1=>regwrite, enr1=>en1, enr2=>en2);

end architecture reg_beh;

--Individual components
-- MUX for choosing btw RS1 and RD

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity muxreg is
port(SRC, DST: in std_logic_vector(4 downto 0);
     s_wr: in std_logic;
     RSRD: out std_logic_vector(4 downto 0));
end entity muxreg;

architecture muxreg_beh of muxreg is
begin

process(SRC, DST, s_wr) is
begin

case s_wr is
when '0' => RSRD <= SRC;
when '1' => RSRD <= DST;
when others => null;
end case;

end process;
end architecture muxreg_beh;

-- Block Ram

library IEEE;
use IEEE.std_logic_1164.all;

entity blockram is
port(addr1, addr2: in std_logic_vector(4 downto 0); -- for RS1 and RS2, need to have mux for choosing
either RS1 or RD
     din1, din2: in std_logic_vector(15 downto 0);
     dout1, dout2: out std_logic_vector(15 downto 0);
     clk: in std_logic;
     wr1, enr1, enr2: in std_logic); --wr1 - write is always in port 1, enr1, enr2 - for reg reads from 2 ports
end entity blockram;
architecture ram_behave of blockram is
component RAMB4_S16_S16 is
port(ADDRA, ADDRB: in std_logic_vector(7 downto 0);
     CLKA, CLKB: in std_logic;
     DIA, DIB: in std_logic_vector(15 downto 0);
     DOA, DOB: out std_logic_vector(15 downto 0);
     ENA, ENB, RSTA, RSTB, WEA, WEB: in std_logic);
end component RAMB4_S16_S16;
```

```
signal gnd: std_logic;
signal addr_ablk, addr_bblk: std_logic_vector(7 downto 0);
begin

gnd <= '0';
addr_ablk <= "000" & addr1;
addr_bblk <= "000" & addr2;

gpregram0: RAMB4_S16_S16 port map(ADDRA=>addr_ablk, ADDRB=>addr_bblk, CLKA=>clk,
CLKB=>clk, DIA=>din1, DIB=>din2, DOA=>dout1, DOB=>dout2, ENA=>enr1, ENB=>enr2,
RSTA=>gnd, RSTB=>gnd, WEA=>wr1, WEB=>gnd);

end architecture ram_behave;

--Sign Extend Unit

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity signext is
generic(N: positive := 64;
          imm: positive := 16);
port(immval: in std_logic_vector(imm-1 downto 0);
   sign: in std_logic;
   extdval: out std_logic_vector(N-1 downto 0));
end entity signext;

architecture signextd_beh of signext is
signal stoint, intval1: integer;
begin

process(immval, sign, stoint, intval1)
begin
stoint <= conv_integer(immval);

case sign is
when '0' => intval1 <= stoint;
when '1' => intval1 <= -stoint;
when others => null;
end case;

extdval <= conv_std_logic_vector(intval1, N);
end process;
end architecture signextd_beh;

-- micro instructions controller for ESPR

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity cntunit0 is
port(opcode: in std_logic_vector(5 downto 0);
    loc: in std_logic_vector(2 downto 0);
    ffpin: in std_logic_vector(7 downto 0);
    ocr_val, aer_val: out std_logic_vector(7 downto 0); -- for 8 bit output code register
    ctrlsigs: out std_logic_vector(24 downto 0));
end entity cntunit0;

architecture cntbeh of cntunit0 is
begin

process(opcode, loc, ffpin) is
begin
case opcode is
when "000000" =>
ctrlsigs <= (others => '0');  -- NOP for opcode '000000'
ocr_val <= (others => '0'); -- No Status
aer_val <= (others => '0');
when "000001" =>
-- IN goes to inp_pkt_ctrlr
ctrlsigs <= "0100000000000000000000001";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "000010" =>
-- OUT
ctrlsigs <= "1000000000000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "000011" =>
-- FWD goes to outp_pkt_ctrlr
ctrlsigs <= "0000000000000000000001100";
ocr_val <= "00000001";
aer_val <= ffpin;
when "000100" =>
-- ABORT1 goes to outp_pkt_ctrlr -- ABORT1-sets LOC bits to '0'
ctrlsigs <= "0000000000000000000001100";
ocr_val <= "00000010";
aer_val(3) <= '0';
aer_val(7 downto 4) <= ffpin(7 downto 4); -- AER = Unused(7-5), R(4), E(3), LOC(2-0)
aer_val(2 downto 0) <= ffpin(2 downto 0);
when "000101" =>
-- DROP
ctrlsigs <= "0000000000000000000001000";
ocr_val <= "00000011";
aer_val <= (others => '0');
when "000110" =>
-- CLR - for GPRs - has reg write ctrl signal on
ctrlsigs <= "0010000000000000000110000"; -- S6 - 1 for ALU out in WB, 0 for ESS out in WB
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "000111" =>
-- MOVE for GPRS - has reg write ctrl signal on
ctrlsigs <= "0010000000000000000110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "001000" =>
-- MOVI for GPRS
ctrlsigs <= "0000000000000000000110000";
```

```
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "001001" =>
-- ADD for GPRS last bit is for load status reg
ctrlsigs <= "0010000000000000100110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "001010" =>
-- SUB for GPRS last bit is for load status reg
ctrlsigs <= "0010000000000000101110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "001011" =>
-- INCR for GPRS last bit is for load status reg
ctrlsigs <= "0010000000000000110110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "001100" =>
-- DECR for GPRS last bit is for load status reg
ctrlsigs <= "0010000000000000111110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "001101" =>
-- OR for GPRS last bit is for load status reg
ctrlsigs <= "0010000000000001010110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "001110" =>
-- AND for GPRS last bit is for load status reg
ctrlsigs <= "0010000000000001011110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "001111" =>
-- EXOR for GPRS last bit is for load status reg
ctrlsigs <= "0010000000000001100110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "010000" =>
-- ONES COMP for GPRS last bit is for load status reg
ctrlsigs <= "0010000000000000010110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "010001" =>
--  SHL for GPRS
ctrlsigs <= "0010000000000010000110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "010010" =>
--  SHR for GPRS
ctrlsigs <= "0010000000000100000110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "010011" =>
--  ROL for GPRS
ctrlsigs <= "0010000000000110000110000";
ocr_val <= (others => '0');
```

```
aer_val <= (others => '0');
when "010100" =>
-- ROR for GPRS
ctrlsigs <= "001000000000100000110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "010101" =>
-- LFPR
ctrlsigs <= "000000001000000000110000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "010110" =>
-- STPR
ctrlsigs <= "001000000100000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "010111" =>
-- BRNE
ctrlsigs <= "001000010000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "011000" =>
-- BREQ
ctrlsigs <= "001000100000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "011001" =>
-- BRGE
ctrlsigs <= "001000110000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "011010" =>
-- BNEZ
ctrlsigs <= "001001000000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "011011" =>
-- BEQZ
ctrlsigs <= "001001010000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "011100" =>
-- JMP
ctrlsigs <= "000100000000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "011101" =>
-- RET
ctrlsigs <= "000010000000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "011110" =>
-- GET
ctrlsigs <= "001000000000100000000010";
ocr_val <= (others => '0');
aer_val <= (others => '0');
```

```
when "011111" =>
-- PUT
ctrlsigs <= "00100000000100000000000010";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "100000" =>
-- BGF -- have this as branch type instr - no connection with ESS ctrl
ctrlsigs <= "00000110000000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "100001" =>
-- BPF -- have this as branch type instr - no connection with ESS ctrl
ctrlsigs <= "00000111000000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
-- NEWLY ADDED as on 5-6-02
when "100010" =>
-- ABORT2 goes to outp_pkt_ctrlr -- ABORT2-sets LOC bits to '0' and sets E bit to '1'
ctrlsigs <= "00000000000000000000001100";
ocr_val <= "00000100";
-- AER = Unused(7-5), R(4), E(3), LOC(2-0)
aer_val(3) <= '1';
aer_val(7 downto 4) <= ffpin(7 downto 4); -- AER = Unused(7-5), R(4), E(3), LOC(2-0)
aer_val(2 downto 0) <= ffpin(2 downto 0);
when "100011" =>
-- BLT
ctrlsigs <= "00100000000000000000000000";
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "100100" =>
-- SETLOC
ctrlsigs <= "00000000000000000000000100";
ocr_val <= (others => '0');
aer_val(2 downto 0) <= loc;
aer_val(7 downto 3) <= ffpin(7 downto 3);
when "100101" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "100110" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "100111" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "101000" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "101001" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "101010" =>
```

```
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "101011" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "101100" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "101101" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "101110" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "101111" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "110000" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "110001" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "110010" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "110011" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "110100" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "110101" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "110110" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "110111" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "111000" =>
```

```
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "111001" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "111010" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "111011" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "111100" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "111101" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "111110" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when "111111" =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
when others =>
ctrlsigs <= (others => '0');
ocr_val <= (others => '0');
aer_val <= (others => '0');
end case;
end process;
end architecture cntbeh;

-- extra circuit for getting written values

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity idextra is
generic(N: positive := 64);
port(regw, trwr, vrwr: in std_logic;
    wrregin, wrtagin, wrvalin: in std_logic_vector(N-1 downto 0);
    wrdataout: out std_logic_vector(N-1 downto 0));
end entity idextra;
architecture idextra_beh of idextra is
signal wrtv: std_logic_vector(2 downto 0);
begin
```

```
wrtv <= regw&trwr&vrwr;
process(wrtv, wrregin, wrtagin, wrvalin) is
begin

case wrtv is
when "100" => wrdataout <= wrregin;
when "010" => wrdataout <= wrtagin;
when "001" => wrdataout <= wrvalin;
when others => wrdataout <= (others => '0');
end case;
end process;
end architecture idextra_beh;

-- MUX for choosing the INST opcode for controller

library IEEE;
use IEEE.std_logic_1164.all;

entity mux_ct is
port (n_op, lm_op: in STD_LOGIC_VECTOR (5 downto 0);
    lm: in STD_LOGIC;
    optoct: out STD_LOGIC_VECTOR (5 downto 0) );
end entity mux_ct;

architecture mux_ct_arch of mux_ct is
begin

process (n_op, lm_op, lm) is
begin
case lm is
when '0' => optoct <= n_op;
when '1' => optoct <= lm_op;
when others => optoct <= (others => '0');
end case;
end process;
end mux_ct_arch;

-- ID/EX stage Regsiter

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ess_idexreg is
generic(N: positive := 64;
            Addr: positive := 16);
port(clk, ID_Flush: in std_logic;
    ctrlin: in std_logic_vector(24 downto 0);
    WB_in: in std_logic_vector(3 downto 0);
    EX_in: in std_logic_vector(12 downto 0);
    PKT_in: in std_logic_vector(6 downto 0);
    GPR_read1_in, GPR_read2_in, sign_ext_in: in std_logic_vector(N-1 downto 0);
    TR_read_in, VR_read_in: in std_logic_vector(N-1 downto 0);
    Br_Addr_in, PKT_Offset_in: in std_logic_vector(Addr-1 downto 0);
    shamt_in: in std_logic_vector(5 downto 0);
    lmor_in, jin_id, rin_id: in std_logic;
    ocr_in_id, aer_in_id: in std_logic_vector(7 downto 0);
    RS1_in, RS2_in, RD_in, TR_in, VR_in: in std_logic_vector(4 downto 0);
    opcodein: in std_logic_vector(5 downto 0);
    opcodeexout: out std_logic_vector(5 downto 0);
    ctrlout: out std_logic_vector(24 downto 0);
    WB_out: out std_logic_vector(3 downto 0);
    EX_out: out std_logic_vector(12 downto 0);
    PKT_out: out std_logic_vector(6 downto 0);
    GPR_read1_out, GPR_read2_out, sign_ext_out: out std_logic_vector(N-1 downto 0);
    TR_read_out_ID, VR_read_out_ID: out std_logic_vector(N-1 downto 0);
    Br_Addr_out, PKT_Offset_out: out std_logic_vector(Addr-1 downto 0);
    shamt_out: out std_logic_vector(5 downto 0);
    lmor_out, TRD_out, VRD_out, jout_id, rout_id: out std_logic;
    ocr_out_id, aer_out_id: out std_logic_vector(7 downto 0);
    RS1_out, RS2_out, RD_out, TR_out, VR_out: out std_logic_vector(4 downto 0));
end entity ess_idexreg;

architecture ess_idexreg_beh of ess_idexreg is
begin

process(clk, ID_Flush, ctrlin, WB_in, EX_in, PKT_in, GPR_read1_in, GPR_read2_in, sign_ext_in,
TR_read_in, VR_read_in, jin_id, rin_id, Br_Addr_in, PKT_Offset_in, lmor_in, shamt_in, ocr_in_id,
aer_in_id, RS1_in, RS2_in, RD_in) is
begin

if (falling_edge(clk)) then
case ID_Flush is
when '0' =>
WB_out <= WB_in;
EX_out <= EX_in;
PKT_out <= PKT_in;
GPR_read1_out <= GPR_read1_in;
GPR_read2_out <= GPR_read2_in;
TR_read_out_ID <= TR_read_in;
VR_read_out_ID <= VR_read_in;
sign_ext_out <= sign_ext_in;
Br_Addr_out <= Br_Addr_in;
PKT_Offset_out <= PKT_Offset_in;
shamt_out <= shamt_in;
lmor_out <= lmor_in;
TRD_out <= WB_in(3);
VRD_out <= WB_in(2);
ocr_out_id <= ocr_in_id;
aer_out_id <= aer_in_id;
RS1_out <= RS1_in;
RS2_out <= RS2_in;
RD_out <= RD_in;
TR_out <= TR_in;
VR_out <= VR_in;
opcodeexout <= opcodein;
jout_id <= jin_id;
rout_id <= rin_id;
ctrlout <= ctrlin;
```

```
    shamt: in std_logic_vector(5 downto 0);
    regrd, trwx, trww, vrwx, vrww, rwx, rww: in std_logic; --new
    alu_O: out std_logic;
    ctrloutEX: out std_logic_vector(24 downto 0);
    opoutEX, mo: out std_logic_vector(5 downto 0);
    aluout, GPR1out, GPR2out, tagsigout: out std_logic_vector(63 downto 0);
    RS1_out, RS2_out, RD_out, TR_out, VR_out: out std_logic_vector(4 downto 0);
    WBct_out: out std_logic_vector(3 downto 0);
    braddrout: out std_logic_vector(15 downto 0);
    gf, pf, ess_full, le, AK, PRr, ldor, EPo, cok, lz: out std_logic;
    outvalue: out std_logic_vector(63 downto 0);
    oo: out std_logic_vector(7 downto 0);
    stag: out std_logic_vector(2 downto 0);
    oram, f1: out std_logic_vector(31 downto 0);
    po: out std_logic_vector(31 downto 0));
end entity ex3top;

architecture ex3top_beh of ex3top is
--components
component ex3stage is
port(clk, clock, clk_pkt, clr, IDV, EOP_in, OPRAMready: in std_logic;
    inp_fm_ram: in std_logic_vector(31 downto 0);
    flag, ocrID: in std_logic_vector(7 downto 0);
    PKToffid: in std_logic_vector(6 downto 0); -- for LFPR and STPR
    RS1rgid, RS2rgid, TRrgid, VRrgid: in std_logic_vector(4 downto 0);
    FSTRD, FSTTRD, FSTVRD, VSTRD, VSTTRD, VSTVRD: in std_logic_vector(4 downto 0); --new
    op_in, prop_in: in std_logic_vector(5 downto 0);
    GPR1id, GPR2id, TRidv, VRidv, extid, WBdatain, aofmex: in std_logic_vector(63 downto 0);
    EXctid: in std_logic_vector(9 downto 0);  --12 downto 10(branch) get in next stage
    PKTctid: in std_logic_vector(6 downto 0);
    shamt: in std_logic_vector(5 downto 0);
    regrd, trwx, trww, vrwx, vrww, rwx, rww, putin, lmor: in std_logic; --new
    alu_O, ACK_in, PRready, ldopram, EOP_out, crcchkok, locz: out std_logic;
    ashout, a1out, a2out, tagmuxout, pkttoregs, outvalue: out std_logic_vector(63 downto 0);
    oo: out std_logic_vector(7 downto 0);
    stag: out std_logic_vector(2 downto 0);
    gf, pf, ess_full, le: out std_logic;
    mopregout: out std_logic_vector(5 downto 0);
    out_to_ram, firstoutp: out std_logic_vector(31 downto 0);
    pkt_out: out std_logic_vector(31 downto 0));
end component ex3stage;

component ex3_ex4_reg is
port(clk, EX_Flush_in: in std_logic;
    braddrin: in std_logic_vector(15 downto 0);
    ctrlinEX: in std_logic_vector(24 downto 0);
    opinEX: in std_logic_vector(5 downto 0);
    WB_in_fm_ex: in std_logic_vector(3 downto 0);
    RS1_in_fm_ex, RS2_in_fm_ex, RD_in_fm_ex, TR_in_fm_ex, VR_in_fm_ex: in std_logic_vector(4
downto 0);
    aluout_fm_ex, pktout_fm_ex, GPR1in, GPr2in: in std_logic_vector(63 downto 0);
    braddrout: out std_logic_vector(15 downto 0);
    ctrloutEX: out std_logic_vector(24 downto 0);
    opoutEX: out std_logic_vector(5 downto 0);
    aluout_to_wb, pktout_to_wb, GPR1out, GPR2out: out std_logic_vector(63 downto 0);
    RS1_out_to_regs, RS2_out_to_regs, RD_out_to_regs, TR_out_to_regs, VR_out_to_regs: out
std_logic_vector(4 downto 0);
    WB_out_fm_wb: out std_logic_vector(3 downto 0));
end component ex3_ex4_reg;

--signals
signal ashoutsig, a1sig, a2sig, pktinsig, pktoutsig, taginsig: std_logic_vector(63 downto 0);

begin
tagsigout <= taginsig;

ex3comp: ex3stage port
map(clk=>clk,clock=>clock,clk_pkt=>clk_pkt,clr=>clr,IDV=>IDV,EOP_in=>EPi,OPRAMready=>ORr,in
p_fm_ram=>iram,flag=>flag,ocrID=>ocrID,PKToffid=>PKToffid,RS1rgid=>RS1rgid,RS2rgid=>RS2rgid,
TRrgid=>TRrgid,VRrgid=>VRrgid,FSTRD=>FSTRD,FSTTRD=>FSTTRD,FSTVRD=>FSTVRD,VSTR
D=>VSTRD,VSTTRD=>VSTTRD,VSTVRD=>VSTVRD,op_in=>op_in,prop_in=>prop_in,GPR1id=>GP
R1id,GPR2id=>GPR2id,TRidv=>TRidv,VRidv=>VRidv,extid=>extid,WBdatain=>WBdatain,aofmex=>ao
fmex,EXctid=>EXctid,PKTctid=>PKTctid,shamt=>shamt,regrd=>regrd,trwx=>trwx,trww=>trww,vrwx=>
vrwx,vrww=>vrww,rwx=>rwx,rww=>rww,putin=>putin,lmor=>lm,alu_O=>alu_O,outvalue=>outvalue,A
CK_in=>AK,PRready=>PRr,ldopram=>ldor,EOP_out=>EPo,crcchkok=>cok,locz=>lz,ashout=>ashoutsig,
a1out=>a1sig,a2out=>a2sig,tagmuxout=>taginsig,pkttoregs=>pktinsig,oo=>oo,stag=>stag,gf=>gf,pf=>pf,e
ss_full=>ess_full,le=>le,mopregout=>mo,out_to_ram=>oram,firstoutp=>f1,pkt_out=>po);

ex3regcomp: ex3_ex4_reg port
map(clk=>clk,EX_Flush_in=>EX_Flush_in,braddrin=>braddrin,ctrlinEX=>ctrlinEX,opinEX=>op_in,WB
_in_fm_ex=>WBinfmid,RS1_in_fm_ex=>RS1rgid,RS2_in_fm_ex=>RS2rgid,RD_in_fm_ex=>RDrgid,TR
_in_fm_ex=>TRrgid,VR_in_fm_ex=>VRrgid,aluout_fm_ex=>ashoutsig,pktout_fm_ex=>pktinsig,GPR1in
=>GPR1id,GPR2in=>GPR2id,braddrout=>braddrout,ctrloutEX=>ctrloutEX,opoutEX=>opoutEX,aluout_t
o_wb=>aluout,pktout_to_wb=>pktoutsig,GPR1out=>GPR1out,GPR2out=>GPR2out,RS1_out_to_regs=>R
S1_out,RS2_out_to_regs=>RS2_out,RD_out_to_regs=>RD_out,TR_out_to_regs=>TR_out,VR_out_to_re
gs=>VR_out,WB_out_fm_wb=>WBct_out);

end architecture ex3top_beh;

--Individual components
-- EX 3rd STAGE

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ex3stage is
port(clk, clock, clk_pkt, clr, IDV, EOP_in, OPRAMready: in std_logic;
    inp_fm_ram: in std_logic_vector(31 downto 0);
    flag, ocrID: in std_logic_vector(7 downto 0);
    PKToffid: in std_logic_vector(6 downto 0); -- for LFPR and STPR
    RS1rgid, RS2rgid, TRrgid, VRrgid: in std_logic_vector(4 downto 0);
    FSTRD, FSTTRD, FSTVRD, VSTRD, VSTTRD, VSTVRD: in std_logic_vector(4 downto 0); --new
    op_in, prop_in: in std_logic_vector(5 downto 0);
    GPR1id, GPR2id, TRidv, VRidv, extid, WBdatain, aofmex: in std_logic_vector(63 downto 0);
    EXctid: in std_logic_vector(9 downto 0);  --12 downto 10(branch) get in next stage
    PKTctid: in std_logic_vector(6 downto 0);
    shamt: in std_logic_vector(5 downto 0);
    regrd, trwx, trww, vrwx, vrww, rwx, rww, putin, lmor: in std_logic; --new
    alu_O, ACK_in, PRready, ldopram, EOP_out, crcchkok, locz: out std_logic;
    ashout, a1out, a2out, tagmuxout, pkttoregs, outvalue: out std_logic_vector(63 downto 0);
    oo: out std_logic_vector(7 downto 0);
    stag: out std_logic_vector(2 downto 0);
    gf, pf, ess_full, le: out std_logic;
    mopregout: out std_logic_vector(5 downto 0);
    out_to_ram, firstoutp: out std_logic_vector(31 downto 0);
    pkt_out: out std_logic_vector(31 downto 0));
end entity ex3stage;

architecture ex3_beh of ex3stage is
--Components
--ALU
component alu_chk0 is
generic (N: integer :=64);
port (a, b: in std_logic_vector(N-1 downto 0);
    S3,S4,S5,Cin: in std_logic;
    result: out std_logic_vector(N-1 downto 0);
    o: out std_logic);
end component alu_chk0;
--Shifter
component shift is
generic (N: positive := 64;
            M: positive := 6);
port(input: in std_logic_vector (N-1 downto 0);
    S0, S1, S2: in std_logic;
    shamt: in std_logic_vector (M-1 downto 0);
    output: out std_logic_vector (N-1 downto 0));
end component shift;
--MUX before ALUSH
component muxalush is
port(GPR_in, TR_in, VR_in, ALU_Sh_out, ext_in, FST_out, PR_in: in std_logic_vector(63 downto 0);
    S8: in std_logic_vector(2 downto 0);
    alumuxout: out std_logic_vector(63 downto 0));
end component muxalush;
--MUX after ALUSH
component muxout is
port(aluout_in, shout_in: in std_logic_vector(63 downto 0);
    Sout: in std_logic;
    alu_sh_out: out std_logic_vector(63 downto 0));
end component muxout;
--FWD
component fwd_new is
port(curop_in, prevop_in: in std_logic_vector(5 downto 0);
    regrd, trwx, trww, vrwx, vrww, rwx, rww: in std_logic;
    RS1_in, RS2_in, EX_WB_RD_in, EX_WB_TRD_in, EX_WB_VRD_in, RDoswbtryout,
VRDoswbtryout, TRDoswbtryout, TR_in, VR_in: in std_logic_vector(4 downto 0);
    pktmuxtopk: out std_logic_vector(2 downto 0);
    essmux_tag: out std_logic_vector(1 downto 0);
    S8: out std_logic_vector(2 downto 0);
    S9: out std_logic_vector(2 downto 0);
    SSh: out std_logic_vector(2 downto 0);
    Salush_out: out std_logic);
end component fwd_new;
--MUX before ESS/TAG
component muxtag is
port(TR_in, ALU_Sh_out, FST_out, PR_in: in std_logic_vector(63 downto 0);
    Stag: in std_logic_vector(2 downto 0);
    tagmuxout: out std_logic_vector(63 downto 0));
end component muxtag;
--MUX before PKT
component muxpkt is
port(GPR_in, TR_in, VR_in, ALU_Sh_out, FST_out, PR_in: in std_logic_vector(63 downto 0);
    Spkt: in std_logic_vector(2 downto 0);
    pktmuxout: out std_logic_vector(63 downto 0));
end component muxpkt;
--PKT PROC UNIT
component pktproc is
generic(M: positive := 32;
        N: positive := 64);
port(clk, ininst_p, IDV, EOP_in_p, outinst_p, OPRAMready, lfpr_p, stpr_p: in std_logic;
    inp_fm_ram: in std_logic_vector(M-1 downto 0);
    inp_fm_mux: in std_logic_vector(N-1 downto 0);
    flaginp: in std_logic_vector(7 downto 0);
    crcchkok: out std_logic;
    lfstoff: in std_logic_vector(6 downto 0);
    ldopram, EOP_out, PRready, ACK_in, locz: out std_logic;
    foutp: out std_logic_vector(M-1 downto 0);
    out_to_regs: out std_logic_vector(N-1 downto 0);
    out_to_ram, pktout: out std_logic_vector(M-1 downto 0));
end component pktproc;
--aereg
component aereg is
port(clk, ldaer : in std_logic;
    flagval_in : in std_logic_vector(7 downto 0);
    aerout  : out std_logic_vector(7 downto 0));
end component aereg;
--ocreg
component ocreg is
port(clk, ldocr : in std_logic;
    val_in : in std_logic_vector(7 downto 0);
    ocrout  : out std_logic_vector(7 downto 0));
end component ocreg;
--moreg
component moreg is
port(clk, ldmor : in std_logic;
    mop_fmpkt_in : in std_logic_vector(5 downto 0);
    mopout  : out std_logic_vector(5 downto 0));
end component moreg;
--ESS
--In order not to mess up with the existing one, I have the whole of ESS here
component esstop0 is
port(tag_in, value_in: in std_logic_vector(63 downto 0);
    clk, clock, ess_we, ess_re, putin: in std_logic;
    gf, pf, ess_full, le: out std_logic;
    outvalue: out std_logic_vector(63 downto 0));
end component esstop0;

-- Signals
signal s0_sh, s1_sh, s2_sh, s3_alu, s4_alu, s5_alu, cin_alu: std_logic;
signal lfpr_pctrl, stpr_pctrl, ldpkreg_pctrl, ldocr_pctrl, ldaer_pctrl, in_pctrl, out_pctrl: std_logic;
signal aeroutsig, moroutsig, morout, ocrout: std_logic_vector(7 downto 0);
signal gf_fm_ess, pf_fm_ess, ccroutsigg, ccroutsigp, Osig: std_logic;
signal aluin1sig, aluin2sig, aluoutsig, shftinsig, shftoutsig, essoutvalue, tag_to_ess, val_to_ess, bdusig1,
bdusig2, PRmuxsigin, pktmuxoutsig, tagmuxoutsig: std_logic_vector(63 downto 0);
signal S8sig, S9sig, SShsig: std_logic_vector(2 downto 0);
signal stag_sig, spkt_sig: std_logic_vector(2 downto 0);
signal Ssig, we_ess, re_ess: std_logic;
signal stag_fmfwd: std_logic_vector(1 downto 0);
begin

alu_O <= Osig;
a1out <= aluin1sig;
a2out <= aluin2sig;
pkttoregs <= PRmuxsigin;
stag <= stag_sig;
tagmuxout <= tagmuxoutsig;

lfpr_pctrl <= PKTctid(4); -- ctrlsigs(16)
stpr_pctrl <= PKTctid(3); -- ctrlsigs(15)
we_ess <= EXctid(9); -- ctrlsigs(14)
re_ess <= EXctid(8); -- ctrlsigs(13)
ldocr_pctrl <= PKTctid(2); -- ctrlsigs(3)
ldaer_pctrl <= PKTctid(1); -- ctrlsigs(2)
ldpkreg_pctrl <= PKTctid(0); -- ctrlsigs(0)
in_pctrl <= PKTctid(5); --ctrlsigs(23)
out_pctrl <= PKTctid(6); --ctrlsigs(24)
s0_sh <= EXctid(7);  -- ctrlsigs(12)
s1_sh <= EXctid(6);  -- ctrlsigs(11)
s2_sh <= EXctid(5);  -- ctrlsigs(10)
s3_alu <= EXctid(4);  -- ctrlsigs(9)
s4_alu <= EXctid(3);  -- ctrlsigs(8)
s5_alu <= EXctid(2);  -- ctrlsigs(7)
cin_alu <= EXctid(1); -- ctrlsigs(6)
oo <= ocrout;
stag_sig <= clr & stag_fmfwd;

-- Mapping
alucomp: alu_chk0 port map(a=>aluin1sig, b=>aluin2sig, S3=>s3_alu, S4=>s4_alu, S5=>s5_alu,
Cin=>cin_alu, result=>aluoutsig, o=>Osig);

alumux1comp: muxalush port map(GPR_in=>GPR1id, TR_in=>TRidv, VR_in=>VRidv,
ALU_Sh_out=>aofmex, ext_in=>extid, FST_out=>WBdatain, PR_in=>PRmuxsigin, S8=>S8sig,
alumuxout=>aluin1sig);
alumux2comp: muxalush port map(GPR_in=>GPR2id, TR_in=>TRidv, VR_in=>VRidv,
ALU_Sh_out=>aofmex, ext_in=>extid, FST_out=>WBdatain, PR_in=>PRmuxsigin, S8=>S9sig,
alumuxout=>aluin2sig);

shiftcomp: shift port map(input=>shftinsig, S0=>s0_sh, S1=>s1_sh, S2=>s2_sh, shamt=>shamt,
output=>shftoutsig);

shmuxcomp: muxalush port map(GPR_in=>GPR1id, TR_in=>TRidv, VR_in=>VRidv,
ALU_Sh_out=>aofmex, ext_in=>extid, FST_out=>WBdatain, PR_in=>PRmuxsigin, S8=>SShsig,
alumuxout=>shftinsig);
alushmuxoutcomp: muxout port map(aluout_in=>aluoutsig, shout_in=>shftoutsig, Sout=>Ssig,
alu_sh_out=>ashout);

fwdcomp: fwd_new port map(curop_in=>op_in, prevop_in=>prop_in, regrd=>regrd, trwx=>trwx,
trww=>trww, vrwx=>vrwx, vrww=>vrww, rwx=>rwx, rww=>rww, RS1_in=>RS1rgid,
RS2_in=>RS2rgid, EX_WB_RD_in=>FSTRD, EX_WB_TRD_in=>FSTTRD,
EX_WB_VRD_in=>FSTVRD, RDoswbtryout=>VSTRD, VRDoswbtryout=>VSTVRD,
TRDoswbtryout=>VSTTRD, TR_in=>TRrgid, VR_in=>VRrgid, pktmuxtopk=>spkt_sig,
essmux_tag=>stag_fmfwd, S8=>S8sig, S9=>S9sig, SSh=>SShsig, Salush_out=>Ssig);
tagmuxcomp: muxtag port map(TR_in=>TRidv, ALU_Sh_out=>aofmex, FST_out=>WBdatain,
PR_in=>PRmuxsigin, Stag=>stag_sig, tagmuxout=>tagmuxoutsig);

pktmuxcomp: muxpkt port map(GPR_in=>GPR1id, TR_in=>TRidv, VR_in=>VRidv,
ALU_Sh_out=>aofmex, FST_out=>WBdatain, PR_in=>PRmuxsigin, Spkt=>spkt_sig,
pktmuxout=>pktmuxoutsig);

pktcomp: pktproc port map(clk=>clk_pkt, ininst_p=>in_pctrl, IDV=>IDV, EOP_in_p=>EOP_in,
outinst_p=>out_pctrl, OPRAMready=>OPRAMready, lfpr_p=>lfpr_pctrl, stpr_p=>stpr_pctrl,
inp_fm_ram=>inp_fm_ram, inp_fm_mux=>pktmuxoutsig, flaginp=>aeroutsig, crcchkok=>crcchkok,
lfstoff=>PKToffid, ldopram=>ldopram, EOP_out=>EOP_out, PRready=>PRready, ACK_in=>ACK_in,
locz=>locz, foutp=>firstoutp, out_to_regs=>PRmuxsigin, out_to_ram=>out_to_ram, pktout=>pkt_out);

aeregcomp: aereg port map(clk=>clk, ldaer=>ldaer_pctrl, flagval_in=>flag, aerout=>aeroutsig);
ocregcomp: ocreg port map(clk=>clk, ldocr=>ldocr_pctrl, val_in=>ocrID, ocrout=>ocrout);

esscomp: esstop0 port map(tag_in=>tagmuxoutsig, value_in=>VRidv, clk=>clk, clock=>clock,
ess_we=>we_ess, ess_re=>re_ess, putin=>putin, gf=>gf, pf=>pf, ess_full=>ess_full, le=>le,
outvalue=>outvalue);

morcomp: moreg port map(clk=>clk, ldmor=>lmor, mop_fmpkt_in=>PRmuxsigin(5 downto 0),
mopout=>mopregout);

end architecture ex3_beh;

--Individual Components
-- Behavioral level description
-- overflow table
-- 1stnum    2ndnum    sign    o
--    +         +         -     1 -- addition
--    -         -         +     1 -- addition
--    +         -         -     1 -- subtraction
--    -         +         +     1 -- subtraction

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity alu_chk0 is
generic (N: integer :=64);
port (a, b: in std_logic_vector(N-1 downto 0);
     S3,S4,S5,Cin: in std_logic;
     result: out std_logic_vector(N-1 downto 0);
     o: out std_logic);
end entity alu_chk0;
architecture behavioral of alu_chk0 is
signal sig: std_logic_vector(N-1 downto 0);
signal sel : std_logic_vector(3 downto 0);

begin
sel <= S3&S4&S5&Cin;
```

```
addsubprocess: process(a, b, sel, sig) is
begin

case sel is
when "0000" =>
sig <= a;
o <= '0';

when "0001" =>
sig <= b;
o <= '0';

when "0010" =>
sig <= not a;
o <= '0';

when "0011" =>
sig <= not b;
o <= '0';

when "0100" =>
sig <= a+b;
if( a(N-1) = '0' and b(N-1) = '0' and sig(N-1) = '1') then
o <= '1';
elsif( a(N-1) = '1' and b(N-1) = '1' and sig(N-1) = '0') then
o <= '1';
else
o <= '0';
end if;

when "0101" =>
sig <= a-b;
if( a(N-1) = '0' and b(N-1) = '1' and sig(N-1) = '1') then
o <= '1';
elsif( a(N-1) = '1' and b(N-1) = '0' and sig(N-1) = '0') then
o <= '1';
else
o <= '0';
end if;

when "0110" =>
sig <= a+"0000000000000000000000000000000000000000000000000000000000000001";
if( a(N-1) = '0' and sig(N-1) = '1') then
o <= '1';
else
o <= '0';
end if;

when "1000" =>
sig <= b+"0000000000000000000000000000000000000000000000000000000000000001";
if( b(N-1) = '0' and sig(N-1) = '1') then
o <= '1';
else
o <= '0';
end if;
```

```
when "0111" =>
sig <= a-"0000000000000000000000000000000000000000000000000000000000000001";
if( a(N-1) = '1' and sig(N-1) = '0') then
o <= '1';
else
o <= '0';
end if;

when "1001" =>
sig <= b-"0000000000000000000000000000000000000000000000000000000000000001";
if( b(N-1) = '1' and sig(N-1) = '0') then
o <= '1';
else
o <= '0';
end if;

when "1010" =>
sig <= a or b;
o <= '0';

when "1011" =>
sig <= a and b;
o <= '0';

when "1100" =>
sig <= a xor b;
o <= '0';

when "1101" =>
sig <= a;
o <= '0';

when "1110" =>
sig <= a;
o <= '0';

when "1111" =>
sig <= a;
o <= '0';

when others => null;
end case;

result <= sig;
end process addsubprocess;
end architecture behavioral;

--Shifter

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity shift is
generic (N: positive := 64;
            M: positive := 6);
port(input: in std_logic_vector (N-1 downto 0);
    S0, S1, S2: in std_logic;
    shamt: in std_logic_vector (M-1 downto 0);
    output: out std_logic_vector (N-1 downto 0));
end entity shift;

architecture shifter_beh of shift is
signal s: std_logic_vector (2 downto 0);
begin

s <= S0&S1&S2;

shftprocess:process(shamt, input, s) is
variable shft, inpt: integer;
variable shftout: std_logic_vector(N-1 downto 0);
variable inpu, outpu: unsigned(N-1 downto 0);
variable shfu: unsigned(M-1 downto 0);
variable in_var, temp_reg: std_logic_vector (N-1 downto 0);
begin

shft := conv_integer(shamt);
inpt := conv_integer(input); -- unsigned.all

inpu := conv_unsigned(inpt, N); --arith.all
shfu := conv_unsigned(shft, M);
in_var := input;
temp_reg := input;

case s is
when "000" => shftout := input; -- pass thru

-- LEFT SHIFT
when "001" =>
outpu := shl(inpu, shfu);
shftout := conv_std_logic_vector(outpu, N);

-- RIGHT SHIFT
when "010" =>
outpu := shr(inpu, shfu);
shftout := conv_std_logic_vector(outpu, N);

-- ROTATE LEFT
when "011" =>
for i in shamt'low to shamt'high loop
if (shamt(i) = '1') then
for j in 0 to ((2**i)-1) loop
temp_reg(j) := in_var((N-(2**i))+j);
end loop;
for k in (2**i) to N-1 loop
temp_reg(k) := in_var(k-(2**i));
end loop;
in_var := temp_reg;
end if;
end loop;
shftout := temp_reg;
```

```
--ROTATE RIGHT
when "100" =>
for i in shamt'low to shamt'high loop
if (shamt(i) = '1') then
for j in N-1 downto N-(2**i) loop
temp_reg(j) := in_var(j-(N-(2**i)));
end loop;
for k in ((N-(2**i))-1) downto 0 loop
temp_reg(k) := in_var(k+(2**i));
end loop;
in_var := temp_reg;
end if;
end loop;
shftout := temp_reg;

when "101" => shftout := input; -- pass thru
when "110" => shftout := input; -- pass thru
when "111" => shftout := input; -- pass thru
when others => shftout := input; -- pass thru
end case;
output <= shftout;
end process;
end architecture shifter_beh;

--MUX used as mux before ALU/Shifter

library IEEE;
use IEEE.std_logic_1164.all;

entity muxalush is
port(GPR_in, TR_in, VR_in, ALU_Sh_out, ext_in, FST_out, PR_in: in std_logic_vector(63 downto 0);
     S8: in std_logic_vector(2 downto 0);
     alumuxout: out std_logic_vector(63 downto 0));
end entity muxalush;

architecture muxalush_beh of muxalush is
signal alumuxout1: std_logic_vector(63 downto 0);
begin

process(S8, GPR_in, TR_in, VR_in, ALU_Sh_out, ext_in, FST_out, PR_in, alumuxout1) is
begin

case S8 is
when "000" => alumuxout1 <= GPR_in;
when "001" => alumuxout1 <= TR_in;
when "010" => alumuxout1 <= VR_in;
when "011" => alumuxout1 <= ALU_Sh_out;
when "101" => alumuxout1 <= FST_out;
when "110" => alumuxout1 <= PR_in;
when "111" => alumuxout1 <= ext_in;
when others => alumuxout1 <= alumuxout1;
end case;
alumuxout <= alumuxout1;
end process;
end architecture muxalush_beh;
```

```
--FWD Unit

library IEEE;
use IEEE.std_logic_1164.all;

entity fwd_new is
port(curop_in, prevop_in: in std_logic_vector(5 downto 0);
    regrd, trwx, trww, vrwx, vrww, rwx, rww: in std_logic;
    RS1_in, RS2_in, EX_WB_RD_in, EX_WB_TRD_in, EX_WB_VRD_in, RDoswbtryout,
VRDoswbtryout, TRDoswbtryout, TR_in, VR_in: in std_logic_vector(4 downto 0);
    pktmuxtopk: out std_logic_vector(2 downto 0);
    essmux_tag: out std_logic_vector(1 downto 0);
    S8: out std_logic_vector(2 downto 0);
    S9: out std_logic_vector(2 downto 0);
    SSh: out std_logic_vector(2 downto 0);
    Salush_out: out std_logic);
end entity fwd_new;

architecture fwd_new_beh of fwd_new is
begin

-- FOR ALU MUX1
s8p:process(regrd, trwx, trww, vrwx, vrww, rwx, rww, RS1_in, RS2_in, EX_WB_RD_in,
EX_WB_TRD_in, EX_WB_VRD_in, RDoswbtryout, TRDoswbtryout, VRDoswbtryout, curop_in,
prevop_in, TR_in, VR_in) is
begin
if(prevop_in = "010101") then --LFPR (LFPR o/p in PKt proc unit will change, so giving from there itself
to ALU as passthru)
S8 <= "110"; -- PKTREG o/p as ALU input
elsif(curop_in = "001000") then-- MOVI
S8 <= "111"; -- sign ext val as ALU input
elsif((regrd = '1' and RS1_in /= "00000" and EX_WB_RD_in = RS1_in and rwx = '1') or (regrd = '1' and
TR_in /= "00000" and EX_WB_TRD_in = TR_in and trwx = '1') or (regrd = '1' and VR_in /= "00000" and
EX_WB_VRD_in = VR_in and vrwx = '1')) then
S8 <= "011"; -- ALU output as ALU input
elsif((regrd = '1' and RS1_in /= "00000" and RDoswbtryout = RS1_in and rww = '1') or (regrd = '1' and
TR_in /= "00000" and TRDoswbtryout = TR_in and trww = '1') or (regrd = '1' and VR_in /= "00000" and
VRDoswbtryout = VR_in and vrww = '1')) then
S8 <= "101"; -- 4th stage output as input
elsif(regrd = '1' and TR_in /= "00000" and RS1_in = "00000" and trwx = '0' and trww = '0') then
S8 <= "001"; -- TR as ALU input
elsif(regrd = '1' and VR_in /= "00000" and RS1_in = "00000" and vrwx = '0' and vrww = '0') then
S8 <= "010"; -- VR as ALU input
elsif(regrd = '1' and RS1_in /= "00000" and EX_WB_RD_in /= RS1_in and RDoswbtryout /= RS1_in)
then
S8 <= "000"; -- GPR as ALU input
else
S8 <= "000"; -- GPR as ALU input
end if;
end process s8p;

-- FOR ALU MUX2
s9p:process(regrd, trwx, trww, vrwx, vrww, rwx, rww, RS1_in, RS2_in, EX_WB_RD_in,
EX_WB_TRD_in, EX_WB_VRD_in, RDoswbtryout, TRDoswbtryout, VRDoswbtryout, TR_in, VR_in) is
begin
```

```
if((regrd = '1' and RS2_in /= "00000" and EX_WB_RD_in = RS2_in and rwx = '1') or (regrd = '1' and
TR_in /= "00000" and EX_WB_TRD_in = TR_in and trwx = '1') or (regrd = '1' and VR_in /= "00000" and
EX_WB_VRD_in = VR_in and vrwx = '1')) then
S9 <= "011"; -- ALU output as ALU input
elsif((regrd = '1' and RS2_in /= "00000" and RDoswbtryout = RS2_in and rww = '1') or (regrd = '1' and
TR_in /= "00000" and TRDoswbtryout = TR_in and trww = '1') or (regrd = '1' and VR_in /= "00000" and
VRDoswbtryout = VR_in and vrww = '1')) then
S9 <= "101"; -- 4th stage output as input
elsif(regrd = '1' and TR_in /= "00000" and RS2_in = "00000" and trwx = '0' and trww = '0') then
S9 <= "001"; -- TR as ALU input
elsif(regrd = '1' and VR_in /= "00000" and RS2_in = "00000" and vrwx = '0' and vrww = '0') then
S9 <= "010"; -- VR as ALU input
elsif(regrd = '1' and RS2_in /= "00000" and EX_WB_RD_in /= RS2_in and RDoswbtryout /= RS2_in)
then
S9 <= "000"; -- GPR as ALU input
else
S9 <= "000"; -- GPR as ALU input
end if;
end process s9p;

-- For Shifter MUX
sshp:process(regrd, trwx, trww, vrwx, vrww, rwx, rww, RS1_in, RS2_in, EX_WB_RD_in,
EX_WB_TRD_in, EX_WB_VRD_in, RDoswbtryout, TRDoswbtryout, VRDoswbtryout, curop_in,
prevop_in, TR_in, VR_in) is
begin
if(prevop_in = "010101") then --LFPR (LFPR o/p in PKt proc unit will change, so giving from there itself
to ALU as passthru)
SSh <= "110"; -- PKTREG o/p as ALU input
elsif(curop_in = "001000") then-- MOVI
SSh <= "111"; -- sign ext val as ALU input
elsif((regrd = '1' and RS1_in /= "00000" and EX_WB_RD_in = RS1_in and rwx = '1') or (regrd = '1' and
TR_in /= "00000" and EX_WB_TRD_in = TR_in and trwx = '1') or (regrd = '1' and VR_in /= "00000" and
EX_WB_VRD_in = VR_in and vrwx = '1')) then
SSh <= "011"; -- ALU output as ALU input
elsif((regrd = '1' and RS1_in /= "00000" and RDoswbtryout = RS1_in and rww = '1') or (regrd = '1' and
TR_in /= "00000" and TRDoswbtryout = TR_in and trww = '1') or (regrd = '1' and VR_in /= "00000" and
VRDoswbtryout = VR_in and vrww = '1')) then
SSh <= "101"; -- 4th stage output as input
elsif(regrd = '1' and TR_in /= "00000" and RS1_in = "00000" and trwx = '0' and trww = '0') then
SSh <= "001"; -- TR as ALU input
elsif(regrd = '1' and VR_in /= "00000" and RS1_in = "00000" and vrwx = '0' and vrww = '0') then
SSh <= "010"; -- VR as ALU input
elsif(regrd = '1' and RS1_in /= "00000" and EX_WB_RD_in /= RS1_in and RDoswbtryout /= RS1_in)
then
SSh <= "000"; -- GPR as ALU input
else
SSh <= "000"; -- GPR as ALU input
end if;
end process sshp;

aluoutp:process(curop_in) is
begin
if(curop_in = "010001" or curop_in = "010010" or curop_in = "010011" or curop_in = "010100") then -- all
the shift operations
Salush_out <= '1';
else
```

```
Salush_out <= '0';
end if;
end process aluoutp;

-- ESS MUX FOR TAGREG
emtp:process(regrd, trwx, trww, vrwx, vrww, rwx, rww, RS1_in, RS2_in, EX_WB_TRD_in,
TRDoswbtryout, curop_in, prevop_in, TR_in) is
begin
if(curop_in = "011110" or curop_in = "011111") then
if(prevop_in = "010101") then --LFPR (LFPR o/p in PKt proc unit will change, so giving from there itself
to ALU as passthru)
essmux_tag <= "11";  --PKTREG o/p as tag input
elsif(regrd = '1' and TR_in /= "00000" and EX_WB_TRD_in = TR_in and trwx = '1') then
essmux_tag <= "01"; -- ALU output as ESS input
elsif(regrd = '1' and TR_in /= "00000" and EX_WB_TRD_in /= TR_in and TRDoswbtryout = TR_in and
trww = '1') then
essmux_tag <= "10"; -- FST output as ESS input
elsif(regrd = '1' and TR_in /= "00000" and EX_WB_TRD_in /= TR_in and TRDoswbtryout /= TR_in )
then
essmux_tag <= "00"; -- TR as ESS input
else
essmux_tag <= "00"; --normal TR as input
end if;
end if;
end process emtp;

-- PKREG mux
pmp:process(regrd, trwx, trww, vrwx, vrww, rwx, rww, RS1_in, RS2_in, EX_WB_VRD_in,
VRDoswbtryout, EX_WB_RD_in, RDoswbtryout, EX_WB_TRD_in, TRDoswbtryout, curop_in,
prevop_in, TR_in, VR_in) is
begin
if(curop_in = "010110") then-- STPR
if(prevop_in = "010101") then --LFPR (LFPR o/p in PKt proc unit will change, so giving from there itself
to ALU as passthru)
pktmuxtopk <= "101";  --PKTREG o/p as pkt input
elsif((regrd = '1' and RS1_in /= "00000" and EX_WB_RD_in = RS1_in and rwx = '1') or (regrd = '1' and
TR_in /= "00000" and EX_WB_TRD_in = TR_in and trwx = '1') or (regrd = '1' and VR_in /= "00000" and
EX_WB_VRD_in = VR_in and vrwx = '1')) then
pktmuxtopk  <= "011"; -- ALU output as PKT input
elsif((regrd = '1' and RS1_in /= "00000" and EX_WB_RD_in /= RS1_in and RDoswbtryout = RS1_in and
rww = '1') or (regrd = '1' and TR_in /= "00000" and EX_WB_TRD_in /= TR_in and TRDoswbtryout =
TR_in and trww = '1') or (regrd = '1' and VR_in /= "00000" and EX_WB_VRD_in /= VR_in and
VRDoswbtryout = VR_in and vrww = '1')) then
pktmuxtopk  <= "100"; -- FST data as PKT input
elsif(regrd = '1' and TR_in /= "00000" and RS1_in = "00000" and trwx = '0' and trww = '0') then
pktmuxtopk  <= "001"; -- TR as PKT input
elsif(regrd = '1' and VR_in /= "00000" and RS1_in = "00000" and vrwx = '0' and vrww = '0') then
pktmuxtopk  <= "010"; -- VR as PKT input
elsif(regrd = '1' and RS1_in /= "00000" and EX_WB_RD_in /= RS1_in and RDoswbtryout /= RS1_in)
then
pktmuxtopk  <= "000"; -- GPR as PKT input
else
pktmuxtopk  <= "110"; -- hold on to prev value
end if;
else
pktmuxtopk  <= "111"; -- zero it out
```

```
end if;
end process pmp;
end architecture fwd_new_beh;

--MUX used as mux after ALU/Shifter output

library IEEE;
use IEEE.std_logic_1164.all;

entity muxout is
port(aluout_in, shout_in: in std_logic_vector(63 downto 0);
    Sout: in std_logic;
    alu_sh_out: out std_logic_vector(63 downto 0));
end entity muxout;

architecture muxout_beh of muxout is
signal alush1: std_logic_vector(63 downto 0);
begin

process(Sout, aluout_in, shout_in, alush1) is
begin
case Sout is
when '0' => alush1 <= aluout_in;
when '1' => alush1 <= shout_in;
when others => alush1 <= alush1;
end case;
alu_sh_out <= alush1;
end process;
end architecture muxout_beh;

--MUX used as mux before ESS-TAG

library IEEE;
use IEEE.std_logic_1164.all;

entity muxtag is
port(TR_in, ALU_Sh_out, FST_out, PR_in: in std_logic_vector(63 downto 0);
    Stag: in std_logic_vector(2 downto 0);
    tagmuxout: out std_logic_vector(63 downto 0));
end entity muxtag;

architecture muxtag_beh of muxtag is
begin

process(Stag, TR_in, ALU_Sh_out, FST_out, PR_in) is
begin
case Stag is
when "100" => tagmuxout <= (others => '0'); --first bit is 'clr'
when "101" => tagmuxout <= (others => '0');
when "110" => tagmuxout <= (others => '0');
when "111" => tagmuxout <= (others => '0');
when "000" => tagmuxout <= TR_in;
when "001" => tagmuxout <= ALU_Sh_out;
when "010" => tagmuxout <= FST_out;
when "011" => tagmuxout <= PR_in;
when others => null;
```

```
end case;
end process;
end architecture muxtag_beh;

--MUX used as mux before PKT

library IEEE;
use IEEE.std_logic_1164.all;

entity muxpkt is
port(GPR_in, TR_in, VR_in, ALU_Sh_out, FST_out, PR_in: in std_logic_vector(63 downto 0);
    Spkt: in std_logic_vector(2 downto 0);
    pktmuxout: out std_logic_vector(63 downto 0));
end entity muxpkt;
architecture muxpkt_beh of muxpkt is
signal pktmuxout1: std_logic_vector(63 downto 0);
begin

process(Spkt, GPR_in, TR_in, VR_in, ALU_Sh_out, FST_out, PR_in, pktmuxout1) is
begin

case Spkt is
when "111" => pktmuxout1 <= (others => '0');
when "000" => pktmuxout1 <= GPR_in;
when "001" => pktmuxout1 <= TR_in;
when "010" => pktmuxout1 <= VR_in;
when "110" => pktmuxout1 <= pktmuxout1;
when "011" => pktmuxout1 <= ALU_Sh_out;
when "100" => pktmuxout1 <= FST_out;
when "101" => pktmuxout1 <= PR_in;
when others => pktmuxout1 <= pktmuxout1;
end case;
pktmuxout <= pktmuxout1;
end process;
end architecture muxpkt_beh;

-- PKT PROCESSING TOP MODULE

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity pktproc is
generic(M: positive := 32;
      N: positive := 64);
port(clk, ininst_p, IDV, EOP_in_p, outinst_p, OPRAMready, lfpr_p, stpr_p: in std_logic;
    inp_fm_ram: in std_logic_vector(M-1 downto 0);
    inp_fm_mux: in std_logic_vector(N-1 downto 0);
    flaginp: in std_logic_vector(7 downto 0);
    crcchkok: out std_logic;
    lfstoff: in std_logic_vector(6 downto 0);
    ldopram, EOP_out, PRready, ACK_in, locz: out std_logic;
    foutp: out std_logic_vector(M-1 downto 0);
    out_to_regs: out std_logic_vector(N-1 downto 0);
    out_to_ram, pktout: out std_logic_vector(M-1 downto 0));
end entity pktproc;
```

```
architecture pktproc_beh of pktproc is
-- Main PKt PROC
component pktram is
port(off_addr: in std_logic_vector(6 downto 0);
    din: in std_logic_vector(31 downto 0);
    dout: out std_logic_vector(31 downto 0);
    clk: in std_logic;
    wepr: in std_logic);
end component pktram;
-- PKT CTRLR
component pktctrl is
port(clk, ininst, IDV, EOP_in, outinst, OPRAMready, zsig, lfpr, stpr: in std_logic;
    weipr, ldfreg, incr_ag, clrag, ldopram, ldlenreg, EOP_out, subo, lfclk, lsclk, sfclk, ssclk, ackin, ldf_FR,
LD_CRCreg, crc_Z, outcrc, clrcrc: out std_logic);
end component pktctrl;
-- ADDR GEN
component addgen is
port(clk, clr, incag: in std_logic;
    inad_ag: in std_logic_vector(6 downto 0);
    outad_ag: out std_logic_vector(6 downto 0));
end component addgen;
-- FIRST REG
component freg is
port(ldfreg, clk: in std_logic;
    addr: in std_logic_vector(6 downto 0);
    inp: in std_logic_vector(31 downto 0);
    outp: out std_logic_vector(31 downto 0));
end component freg;
-- Length Reg
component lenreg0 is
generic(al: positive := 16);
port(leninp1: in std_logic_vector(al-1 downto 0);
    clk, ldlenreg, subsig: in std_logic;
    lenoutp: out std_logic_vector(al-1 downto 0));
end component lenreg0;
-- Offset Length Equality
component oflleneq is
generic(al: positive := 16);
port(lenin: in std_logic_vector(al-1 downto 0);
    zo: out std_logic);
end component offleneq;
-- LA sigs
component la is
port(ai1, ai2, ai3: in std_logic_vector(6 downto 0);
    ls: in std_logic_vector(1 downto 0);
    ao: out std_logic_vector(6 downto 0));
end component la;
-- Length Selection
component lsel is
port(ss: in std_logic;
    flensig, slensig: in std_logic_vector(15 downto 0);
    lselout: out std_logic_vector(15 downto 0));
end component lsel;
--LFPR CKT
component lfprckt is
port(lfc, lsc: in std_logic;
    inp32: in std_logic_vector(31 downto 0); -- for LFPR
    offsi: in std_logic_vector(6 downto 0);
    offso: out std_logic_vector(6 downto 0);
    outp64: out std_logic_vector(63 downto 0));
end component lfprckt;
-- STPR CKT
component stprckt is
port(sfc, ssc: in std_logic;
    inp64: in std_logic_vector(63 downto 0); -- for STPR
    offi: in std_logic_vector(6 downto 0);
    offo: out std_logic_vector(6 downto 0);
    outp32: out std_logic_vector(31 downto 0));
end component stprckt;
-- STPR SEL
component ssel is
port(inp1, inp2, fos, outinp: in std_logic_vector(31 downto 0);
    fin: in std_logic_vector(7 downto 0);
    sts, ldfmfr, oc: in std_logic;
    inpo: out std_logic_vector(31 downto 0));
end component ssel;
-- CRC MODULE
component crcmod is
port(ldcr, crcz: in std_logic;
    infmpkt: in std_logic_vector(31 downto 0);
    crcinfmpkt, crcin: in std_logic_vector(31 downto 0);
    crccalc_out: out std_logic_vector(31 downto 0);
    crcchkok: out std_logic);
end component crcmod;
-- CRC STORE
component crcst is
port(clk : in std_logic;
    crc_calc_in : in std_logic_vector(31 downto 0);
    crc_calc_out  : out std_logic_vector(31 downto 0));
end component crcst;
-- OPRAM DATA OUT
component crcout_ram is
port(epout: in std_logic;
    crc_cin, outramin: in std_logic_vector(31 downto 0);
    outramout: out std_logic_vector(31 downto 0));
end component crcout_ram;
-- Activating Inst
component tstin is
port(in_inst, clk: in std_logic;
    ininstout: out std_logic);
end component tstin;

signal clrsig, incagsig, weprsig, zsig1, crcensig, ldfregsig, ldlenrsig, subsig, lfsig, lssig, sfsig, sssig, lforls,
sforss, EOP_outsig, lffsig, ldcrsig, crczero, ocsig, clrcrcsig: std_logic;
signal lfssfs: std_logic_vector(1 downto 0);
signal add_off, outagsig, offsig, offsig1: std_logic_vector(6 downto 0);
signal lengthsig, lengthoutpsig: std_logic_vector(15 downto 0);
signal foutpsig, outram, toram, dintoram: std_logic_vector(M-1 downto 0);
signal calccrcin, crcintomod: std_logic_vector(M-1 downto 0);
signal ininst_s,  outinst_s, lfpr_s, stpr_s, EOP_in_s: std_logic;
begin
```

```
foutp <= foutpsig;
out_to_ram <= outram;
EOP_out <= EOP_outsig;
PRready <= EOP_outsig;
locz <= not(foutpsig(0) or foutpsig(1) or foutpsig(2));
lforls <= lfsig or lssig;
sforss <= sfsig or sssig;
lfssfs <= lforls & sforss;

-- Activating instructions
INcomp: tstin port map(in_inst=>ininst_p, clk=>clk, ininstout=>ininst_s);
OUTcomp: tstin port map(in_inst=>outinst_p, clk=>clk, ininstout=>outinst_s);
LFPRcomp1: tstin port map(in_inst=>lfpr_p, clk=>clk, ininstout=>lfpr_s);
STPRcomp1: tstin port map(in_inst=>stpr_p, clk=>clk, ininstout=>stpr_s);
EOPcomp: tstin port map(in_inst=>EOP_in_p, clk=>clk, ininstout=>EOP_in_s);

--PKT PROC COMPONENTS
addgencomp: addgen port map(clk=>clk, clr=>clrsig, incag=>incagsig, inad_ag=>add_off,
outad_ag=>outagsig);
pktramcomp: pktram port map(off_addr=>add_off, din=>dintoram, dout=>outram, clk=>clk,
wepr=>weprsig);

pktctrlcomp: pktctrl port map(clk=>clk, ininst=>ininst_s, IDV=>IDV, EOP_in=>EOP_in_s,
outinst=>outinst_s, OPRAMready=>OPRAMready, zsig=>zsig1, lfpr=>lfpr_s, stpr=>stpr_s,
weipr=>weprsig, ldfreg=>ldfregsig, incr_ag=>incagsig, clrag=>clrsig, ldopram=>ldopram,
ldlenreg=>ldlenrsig, EOP_out=>EOP_outsig, subo=>subsig, lfclk=>lfsig, lsclk=>lssig, sfclk=>sfsig,
ssclk=>sssig, ackin=>ACK_in, ldf_FR=>lffsig, LD_CRCreg=>ldcrsig, crc_Z=>open, outcrc=>ocsig,
clrcrc=>clrcrcsig);

fregcomp: freg port map(ldfreg=>ldfregsig, clk=>clk, addr=>outagsig, inp=>dintoram, outp=>foutpsig);

lengthreg: lenreg0 port map(leninp1=>lengthoutpsig, clk=>clk, ldlenreg=>ldlenrsig, subsig=>subsig,
lenoutp=>lengthsig);

offleneqcalc: offleneq port map(lenin=>lengthsig, zo=>zsig1);

lselcomp: lsel port map(ss=>subsig, flensig=>foutpsig(31 downto 16), slensig=>lengthsig,
lselout=>lengthoutpsig);
lacomp: la port map(ai1=>outagsig, ai2=>offsig, ai3=>offsig1, ls=>lfssfs, ao=>add_off);

lfprcomp: lfprckt port map(lfc=>lfsig, lsc=>lssig, inp32=>outram, offsi=>lfstoff, offso=>offsig,
outp64=>out_to_regs);

stprcomp: stprckt port map(sfc=>sfsig, ssc=>sssig, inp64=>inp_fm_mux, offi=>lfstoff, offo=>offsig1,
outp32=>toram);

stprselcomp: ssel port map(inp1=>inp_fm_ram, inp2=>toram, fos=>foutpsig, outinp=>outram,
fin=>flaginp, sts=>sforss, ldfmfr=>lffsig, oc=>ocsig, inpo=>dintoram);

CRCmodcomp: crcmod port map(ldcr=>ldcrsig, crcz=>clrcrcsig, infmpkt=>dintoram,
crcinfmpkt=>dintoram, crcin=>crcintomod, crccalc_out=>calccrcin, crchkok=>crcchkok);
CRCstorecomp: crcst port map(clk=>clk, crc_calc_in=>calccrcin, crc_calc_out=>crcintomod);
outramcomp: crcout_ram port map(epout=>EOP_outsig, crc_cin=>calccrcin, outramin=>outram,
outramout=>pktout);

end architecture pktproc_beh;
```

```
--For PKT RAM
-- Using RAM128X1 for 128X32 RAM

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity pktram is
port(off_addr: in std_logic_vector(6 downto 0);
    din: in std_logic_vector(31 downto 0);
    dout: out std_logic_vector(31 downto 0);
    clk: in std_logic;
    wepr: in std_logic);
end entity pktram;

architecture pktram_beh of pktram is

component ram_128x1s is
port(clk, we: in std_logic;
     addr: in std_logic_vector(6 downto 0);
     data_in: in std_logic;
     data_out: out std_logic);
end component ram_128x1s;
begin

R1281: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(0), data_out=>dout(0));
R1282: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(1), data_out=>dout(1));
R1283: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(2), data_out=>dout(2));
R1284: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(3), data_out=>dout(3));
R1285: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(4), data_out=>dout(4));
R1286: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(5), data_out=>dout(5));
R1287: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(6), data_out=>dout(6));
R1288: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(7), data_out=>dout(7));
R1289: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(8), data_out=>dout(8));
R12810: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(9),
data_out=>dout(9));
R12811: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(10),
data_out=>dout(10));
R12812: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(11),
data_out=>dout(11));
R12813: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(12),
data_out=>dout(12));
R12814: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(13),
data_out=>dout(13));
R12815: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(14),
data_out=>dout(14));
R12816: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(15),
data_out=>dout(15));
R12817: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(16),
data_out=>dout(16));
R12818: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(17),
data_out=>dout(17));
R12819: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(18),
data_out=>dout(18));
```

```
R12820: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(19),
data_out=>dout(19));
R12821: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(20),
data_out=>dout(20));
R12822: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(21),
data_out=>dout(21));
R12823: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(22),
data_out=>dout(22));
R12824: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(23),
data_out=>dout(23));
R12825: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(24),
data_out=>dout(24));
R12826: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(25),
data_out=>dout(25));
R12827: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(26),
data_out=>dout(26));
R12828: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(27),
data_out=>dout(27));
R12829: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(28),
data_out=>dout(28));
R12830: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(29),
data_out=>dout(29));
R12831: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(30),
data_out=>dout(30));
R12832: ram_128x1s port map(clk=>clk, we=>wepr, addr=>off_addr, data_in=>din(31),
data_out=>dout(31));
end architecture pktram_beh;

--For PKT RAM -RAM128X1

library IEEE;
use IEEE.std_logic_1164.all;

entity ram_128x1s is
port(clk, we: in std_logic;
    addr: in std_logic_vector(6 downto 0);
    data_in: in std_logic;
    data_out: out std_logic);
end entity ram_128x1s;

architecture behvram of ram_128x1s is

component RAM128x1S is
port(WE, D, WCLK, A0, A1, A2, A3, A4, A5, A6: in std_logic;
    O: out std_logic);
end component RAM128x1S;
begin

R1281: RAM128x1S port map(WE=>we, D=>data_in, WCLK=>clk, A0=>addr(0), A1=>addr(1),
A2=>addr(2), A3=>addr(3), A4=>addr(4), A5=>addr(5), A6=>addr(6), O=>data_out);
end architecture behvram;

-- PKT Controller

library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity pktctrl is
port(clk, ininst, IDV, EOP_in, outinst, OPRAMready, zsig, lfpr, stpr: in std_logic;
    weipr, ldfreg, incr_ag, clrag, ldopram, ldlenreg, EOP_out, subo, lfclk, lsclk, sfclk, ssclk, ackin, ldf_FR,
LD_CRCreg, crc_Z, outcrc, clrcrc: out std_logic);
end entity pktctrl;

architecture pktctrl_beh of pktctrl is

component FD is
port(D, C: in std_logic;
    Q: out std_logic);
end component FD;

component FD_1 is
port(D, C: in std_logic;
    Q: out std_logic);
end component FD_1;

signal idv_bar, eopi_bar, oprbar, zbar: std_logic;
signal pd0, pd1, pd2, pd3, pd4, pd5, pd6, pd7, pd8, pd9, pd10, pd11, pd12, pd13: std_logic;
signal pt0, pt1, pt2, pt3, pt4, pt5, pt6, pt7, pt8, pt9, pt10, pt11, pt12, pt13: std_logic;
begin

idv_bar <= not(IDV);
eopi_bar <= not(EOP_in);
oprbar <= not(OPRAMready);
zbar <= not(zsig);

dff_p0: FD port map(D=>pd0, C=>clk, Q=>pt0);
dff_p1: FD port map(D=>pd1, C=>clk, Q=>pt1);
dff_p2: FD port map(D=>pd2, C=>clk, Q=>pt2);
dff_p3: FD port map(D=>pd3, C=>clk, Q=>pt3);
dff_p4: FD port map(D=>pd4, C=>clk, Q=>pt4);
dff_p5: FD port map(D=>pd5, C=>clk, Q=>pt5);
dff_p6: FD port map(D=>pd6, C=>clk, Q=>pt6);
dff_p7: FD port map(D=>pd7, C=>clk, Q=>pt7);
dff_p8: FD port map(D=>pd8, C=>clk, Q=>pt8);
dff_p9: FD port map(D=>pd9, C=>clk, Q=>pt9);
dff_p10: FD port map(D=>pd10, C=>clk, Q=>pt10);
dff_p11: FD_1 port map(D=>pd11, C=>clk, Q=>pt11);
dff_p12: FD port map(D=>pd12, C=>clk, Q=>pt12);
dff_p13: FD_1 port map(D=>pd13, C=>clk, Q=>pt13);

--next state equations
pd0 <= (ininst or (idv_bar and pt0));
pd1 <= ( (IDV and pt0) or (eopi_bar and pt2) );
pd2 <= pt1;
pd3 <= EOP_in and pt2;
pd4 <= pt3;
pd5 <= (outinst or (oprbar and pt5));
pd6 <= (OPRAMready and pt5);
pd7 <= (pt6 or (zbar and pt8) );

pd8 <= pt7;
pd9 <= zsig and pt8;
pd10 <= lfpr;
```

```
pd11 <= pt10;
pd12 <= stpr;
pd13 <= pt12;

--output equations
weipr <= pt1 or pt12 or stpr or pt13 or pt6;
incr_ag <= pt1 or pt7;
ldfreg <= pt1;
clrag <= pt9 or pt6 or pt4;
LD_CRCreg <= pt0 or pt2 or pt3 or pt4 or pt5 or pt8 or pt9;
ldopram <= pt7 or pt9;
ldlenreg <= pt6;
subo <= pt7;
EOP_out <= pt9;
lfclk <= pt10;
lsclk <= pt11;
sfclk <= pt12;
ssclk <= pt13;
ackin <= pt1;
ldf_FR <= pt6;
crc_Z <= pt6;
clrcrc <= pt6;
outcrc <= pt6 or pt7 or pt8;
end architecture pktctrl_beh;

-- Address Generator for pktram

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity addgen is
port(clk, clr, incag: in std_logic;
    inad_ag: in std_logic_vector(6 downto 0);
    outad_ag: out std_logic_vector(6 downto 0));
end entity addgen;

architecture addgen_beh of addgen is
signal ciag: std_logic_vector(1 downto 0);
signal outad_ags: std_logic_vector(6 downto 0);
begin

ciag <= clr & incag;

process(clk, ciag, inad_ag, outad_ags) is
begin
if (rising_edge(clk)) then
case ciag is
when "10" => outad_ags <= (others => '0');
when "11" => outad_ags <= (others => '0');
when "01" => outad_ags <= inad_ag + 1;
when "00" => outad_ags <= outad_ags;
when others => null;
end case;
end if;
```

```
outad_ag <= outad_ags;
end process;
end architecture addgen_beh;

-- For Firstregister

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity freg is
port(ldfreg, clk: in std_logic;
    addr: in std_logic_vector(6 downto 0);
    inp: in std_logic_vector(31 downto 0);
    outp: out std_logic_vector(31 downto 0));
end entity freg;

architecture freg_beh of freg is
begin

process(clk, ldfreg, addr, inp) is
begin
if(rising_edge(clk)) then
if(ldfreg = '1') then
if(addr = "0000000") then
outp <= inp;
end if;
end if;
end if;
end process;
end architecture freg_beh;

-- Length Reg
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity lenreg0 is
generic(al: positive := 16); --16
port(leninp1: in std_logic_vector(al-1 downto 0);
    clk, ldlenreg, subsig: in std_logic;
    lenoutp: out std_logic_vector(al-1 downto 0));
end entity lenreg0;

architecture lenreg0_beh of lenreg0 is
begin

process(clk, ldlenreg, leninp1, subsig) is
begin
if(rising_edge(clk)) then
if(ldlenreg = '1') then
lenoutp <= leninp1;
elsif(subsig = '1') then
lenoutp <= leninp1 - 1;
```

```
end if;
end if;
end process;
end architecture lenreg0_beh;

-- Equality check unit
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity offleneq is
generic(al: positive := 16); --4
port(lenin: in std_logic_vector(al-1 downto 0);
    zo: out std_logic);
end entity offleneq;

architecture offleneq_beh of offleneq is
signal leninsig: std_logic_vector(al-1 downto 0);
begin

process(lenin, leninsig) is
variable ole_or: std_logic;
begin
ole_or := '0';

for i in al-1 downto 0 loop
ole_or := ole_or or lenin(i);
end loop;

zo <= not (ole_or);
end process;
end architecture offleneq_beh;

-- For len and add signals

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity la is
port(ai1, ai2, ai3: in std_logic_vector(6 downto 0);
    ls: in std_logic_vector(1 downto 0);
    ao: out std_logic_vector(6 downto 0));
end entity la;

architecture la_beh of la is
begin

process(ls, ai1, ai2, ai3) is
begin
case ls is
when "10" =>
ao <= ai2; -- LFPR Offset
```

```
when "01" =>
ao <= ai3; -- STPR Offset
when "00" =>
ao <= ai1; -- PKRAM address
when "11" =>
ao <= (others => '0');
when others =>
ao <= (others => '0');
end case;
end process;
end architecture la_beh;

-- MUX for length sel

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity lsel is
port(ss: in std_logic;
     flensig, slensig: in std_logic_vector(15 downto 0);
     lselout: out std_logic_vector(15 downto 0));
end entity lsel;

architecture lsel_beh of lsel is
begin

process(ss, flensig, slensig) is
begin
case ss is
when '0' => lselout <= flensig;
when '1' => lselout <= slensig;
when others => lselout <= (others => '0');
end case;
end process;
end architecture lsel_beh;

-- FOR LFPR CKT
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity lfprckt is
port(lfc, lsc: in std_logic;
     inp32: in std_logic_vector(31 downto 0); -- for LFPR
     offsi: in std_logic_vector(6 downto 0);
     offso: out std_logic_vector(6 downto 0);
     outp64: out std_logic_vector(63 downto 0));
end entity lfprckt;

architecture lfprckt_beh of lfprckt is
signal sig64: std_logic_vector(63 downto 0);
signal lfsc: std_logic_vector(1 downto 0);
signal offs: std_logic_vector(6 downto 0);
```

```
begin

lfsc <= lfc & lsc;
process(lfsc, inp32, sig64, offs, offsi) is
begin
case lfsc is
when "10" =>
sig64(31 downto 0) <= inp32;
sig64(63 downto 32) <= sig64(63 downto 32);
offs <= offsi;
when "01" =>
sig64(63 downto 32) <= inp32;
sig64(31 downto 0) <= sig64(31 downto 0);
offs <= offsi + 1;

when "00" =>
sig64 <= sig64;
offs <= offs;

when "11" =>
sig64 <= sig64;
offs <= offs;

when others =>
sig64 <= sig64;
offs <= (others => '0');
end case;

offso <= offs;
outp64 <= sig64;
end process;
end architecture lfprckt_beh;

-- FOR STPR CKT
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity stprckt is
port(sfc, ssc: in std_logic;
    inp64: in std_logic_vector(63 downto 0); -- for STPR
    offi: in std_logic_vector(6 downto 0);
    offo: out std_logic_vector(6 downto 0);
    outp32: out std_logic_vector(31 downto 0));
end entity stprckt;

architecture stprckt_beh of stprckt is
signal sfsc: std_logic_vector(1 downto 0);
signal ofs: std_logic_vector(6 downto 0);
begin

sfsc <= sfc & ssc;
process(sfsc, inp64, offi, ofs) is
begin
```

```
case sfsc is
when "10" =>
outp32 <= inp64(31 downto 0);
ofs <= offi;

when "01" =>
outp32 <= inp64(63 downto 32);
ofs <= offi + 1;

when "00" =>
outp32 <= (others => '0');
ofs <= ofs;

when "11" =>
outp32 <= (others => '0');
ofs <= ofs;

when others =>
outp32 <= (others => '0');
ofs <= (others => '0');

end case;
offo <= ofs;
end process;
end architecture stprckt_beh;

-- For STPR SEL

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ssel is
port(inp1, inp2, fos, outinp: in std_logic_vector(31 downto 0);
    fin: in std_logic_vector(7 downto 0);
    sts, ldfmfr, oc: in std_logic;
    inpo: out std_logic_vector(31 downto 0));
end entity ssel;

architecture ssel_beh of ssel is
signal slo: std_logic_vector(2 downto 0);
begin

slo <= sts & ldfmfr & oc;
process(slo, inp1, inp2, fos, fin, outinp) is
begin
case slo is
when "100" =>
inpo <= inp2; -- LFPR, STPR Offset
when "000" =>
inpo <= inp1; -- PKRAM address
when "011" =>
inpo <= fos(31 downto 8)&fin;
when "010" =>
inpo <= fos(31 downto 8)&fin;
```

```
when "001" =>
inpo <= outinp;
when others =>
inpo <= (others => '0');
end case;
end process;
end architecture ssel_beh;

-- CRC MODULE

library IEEE;
use IEEE.std_logic_1164.all;

entity crcmod is
port(ldcr, crcz: in std_logic;
    infmpkt: in std_logic_vector(31 downto 0);
    crcinfmpkt, crcin: in std_logic_vector(31 downto 0);
    crccalc_out: out std_logic_vector(31 downto 0);
    crcchkok: out std_logic);
end entity crcmod;

architecture crcmod_beh of crcmod is
-- components for CRC-32 calculation
component crc32w16 is
port( crcin: in std_logic_vector(31 downto 0);
     Data_in:  in std_logic_vector(31 downto 0);
     CRCout: out std_logic_vector(31 downto 0));
end component crc32w16;

component compcrc is
port(calccrc, crcin: in std_logic_vector(31 downto 0);
    crcchkout: out std_logic);
end component compcrc;

component crcout is
port(ldcr, crcz: in std_logic;
    crcoutin, cin2: in std_logic_vector(31 downto 0);
    crcoutout: out std_logic_vector(31 downto 0));
end component crcout;

signal CRCsignal, crcc_out: std_logic_vector(31 downto 0);
begin
crccalc_out <= crcc_out;

CRCoutcomp: crcout port map(ldcr=>ldcr, crcz=>crcz, crcoutin=>CRCsignal, cin2=>crcin,
crcoutout=>crcc_out);
CRC32: crc32w16 port map(crcin=>crcin, Data_in=>infmpkt, CRCout=>CRCsignal);
CRCcmp: compcrc port map(calccrc=>crcin, crcin=>crcinfmpkt, crcchkout=>crcchkok);

end architecture crcmod_beh;

-- CRCtop

library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity crc32w16 is
port( crcin: in std_logic_vector(31 downto 0);
    Data_in: in std_logic_vector(31 downto 0);
    CRCout: out std_logic_vector(31 downto 0));
end entity crc32w16;

architecture crc32w16_beh of crc32w16 is

component nextCRC32_D32  is
port( Data:  in std_logic_vector(31 downto 0);
    CRC:   in std_logic_vector(31 downto 0);
    NewCRC: out std_logic_vector(31 downto 0));
end component nextCRC32_D32;
begin

crc32comp: nextCRC32_D32 port map(Data=>Data_in(31 downto 0), CRC=>crcin, NewCRC=>CRCout);
end architecture crc32w16_beh;

-- CRC-32 for 32-input width

library IEEE;
use IEEE.std_logic_1164.all;

entity nextCRC32_D32  is
port( Data:  in std_logic_vector(31 downto 0);
    CRC:   in std_logic_vector(31 downto 0);
    NewCRC: out std_logic_vector(31 downto 0));
end entity nextCRC32_D32;

architecture crc_beh of nextCRC32_D32 is
signal D: std_logic_vector(31 downto 0);
signal C: std_logic_vector(31 downto 0);
begin

process(Data, CRC, D, C) is
begin
  D <= Data;
  C <= CRC;
  NewCRC(0) <= D(31) xor D(30) xor D(29) xor D(28) xor D(26) xor D(25) xor
          D(24) xor D(16) xor D(12) xor D(10) xor D(9) xor D(6) xor
          D(0) xor C(0) xor C(6) xor C(9) xor C(10) xor C(12) xor
          C(16) xor C(24) xor C(25) xor C(26) xor C(28) xor C(29) xor
          C(30) xor C(31);
  NewCRC(1) <= D(28) xor D(27) xor D(24) xor D(17) xor D(16) xor D(13) xor
          D(12) xor D(11) xor D(9) xor D(7) xor D(6) xor D(1) xor
          D(0) xor C(0) xor C(1) xor C(6) xor C(7) xor C(9) xor
          C(11) xor C(12) xor C(13) xor C(16) xor C(17) xor C(24) xor
          C(27) xor C(28);
  NewCRC(2) <= D(31) xor D(30) xor D(26) xor D(24) xor D(18) xor D(17) xor
          D(16) xor D(14) xor D(13) xor D(9) xor D(8) xor D(7) xor
          D(6) xor D(2) xor D(1) xor D(0) xor C(0) xor C(1) xor
          C(2) xor C(6) xor C(7) xor C(8) xor C(9) xor C(13) xor
          C(14) xor C(16) xor C(17) xor C(18) xor C(24) xor C(26) xor
          C(30) xor C(31);
  NewCRC(3) <= D(31) xor D(27) xor D(25) xor D(19) xor D(18) xor D(17) xor
          D(15) xor D(14) xor D(10) xor D(9) xor D(8) xor D(7) xor
```

```
        D(3) xor D(2) xor D(1) xor C(1) xor C(2) xor C(3) xor
        C(7) xor C(8) xor C(9) xor C(10) xor C(14) xor C(15) xor
        C(17) xor C(18) xor C(19) xor C(25) xor C(27) xor C(31);
NewCRC(4) <= D(31) xor D(30) xor D(29) xor D(25) xor D(24) xor D(20) xor
        D(19) xor D(18) xor D(15) xor D(12) xor D(11) xor D(8) xor
        D(6) xor D(4) xor D(3) xor D(2) xor D(0) xor C(0) xor
        C(2) xor C(3) xor C(4) xor C(6) xor C(8) xor C(11) xor
        C(12) xor C(15) xor C(18) xor C(19) xor C(20) xor C(24) xor
        C(25) xor C(29) xor C(30) xor C(31);
NewCRC(5) <= D(29) xor D(28) xor D(24) xor D(21) xor D(20) xor D(19) xor
        D(13) xor D(10) xor D(7) xor D(6) xor D(5) xor D(4) xor
        D(3) xor D(1) xor D(0) xor C(0) xor C(1) xor C(3) xor
        C(4) xor C(5) xor C(6) xor C(7) xor C(10) xor C(13) xor
        C(19) xor C(20) xor C(21) xor C(24) xor C(28) xor C(29);
NewCRC(6) <= D(30) xor D(29) xor D(25) xor D(22) xor D(21) xor D(20) xor
        D(14) xor D(11) xor D(8) xor D(7) xor D(6) xor D(5) xor
        D(4) xor D(2) xor D(1) xor C(1) xor C(2) xor C(4) xor
        C(5) xor C(6) xor C(7) xor C(8) xor C(11) xor C(14) xor
        C(20) xor C(21) xor C(22) xor C(25) xor C(29) xor C(30);
NewCRC(7) <= D(29) xor D(28) xor D(25) xor D(24) xor D(23) xor D(22) xor
        D(21) xor D(16) xor D(15) xor D(10) xor D(8) xor D(7) xor
        D(5) xor D(3) xor D(2) xor D(0) xor C(0) xor C(2) xor
        C(3) xor C(5) xor C(7) xor C(8) xor C(10) xor C(15) xor
        C(16) xor C(21) xor C(22) xor C(23) xor C(24) xor C(25) xor
        C(28) xor C(29);
NewCRC(8) <= D(31) xor D(28) xor D(23) xor D(22) xor D(17) xor D(12) xor
        D(11) xor D(10) xor D(8) xor D(4) xor D(3) xor D(1) xor
        D(0) xor C(0) xor C(1) xor C(3) xor C(4) xor C(8) xor
        C(10) xor C(11) xor C(12) xor C(17) xor C(22) xor C(23) xor
        C(28) xor C(31);
NewCRC(9) <= D(29) xor D(24) xor D(23) xor D(18) xor D(13) xor D(12) xor
        D(11) xor D(9) xor D(5) xor D(4) xor D(2) xor D(1) xor
        C(1) xor C(2) xor C(4) xor C(5) xor C(9) xor C(11) xor
        C(12) xor C(13) xor C(18) xor C(23) xor C(24) xor C(29);
NewCRC(10) <= D(31) xor D(29) xor D(28) xor D(26) xor D(19) xor D(16) xor
        D(14) xor D(13) xor D(9) xor D(5) xor D(3) xor D(2) xor
        D(0) xor C(0) xor C(2) xor C(3) xor C(5) xor C(9) xor
        C(13) xor C(14) xor C(16) xor C(19) xor C(26) xor C(28) xor
        C(29) xor C(31);
NewCRC(11) <= D(31) xor D(28) xor D(27) xor D(26) xor D(25) xor D(24) xor
        D(20) xor D(17) xor D(16) xor D(15) xor D(14) xor D(12) xor
        D(9) xor D(4) xor D(3) xor D(1) xor D(0) xor C(0) xor
        C(1) xor C(3) xor C(4) xor C(9) xor C(12) xor C(14) xor
        C(15) xor C(16) xor C(17) xor C(20) xor C(24) xor C(25) xor
        C(26) xor C(27) xor C(28) xor C(31);
NewCRC(12) <= D(31) xor D(30) xor D(27) xor D(24) xor D(21) xor D(18) xor
        D(17) xor D(15) xor D(13) xor D(12) xor D(9) xor D(6) xor
        D(5) xor D(4) xor D(2) xor D(1) xor D(0) xor C(0) xor
        C(1) xor C(2) xor C(4) xor C(5) xor C(6) xor C(9) xor
        C(12) xor C(13) xor C(15) xor C(17) xor C(18) xor C(21) xor
        C(24) xor C(27) xor C(30) xor C(31);
NewCRC(13) <= D(31) xor D(28) xor D(25) xor D(22) xor D(19) xor D(18) xor
        D(16) xor D(14) xor D(13) xor D(10) xor D(7) xor D(6) xor
        D(5) xor D(3) xor D(2) xor D(1) xor C(1) xor C(2) xor
        C(3) xor C(5) xor C(6) xor C(7) xor C(10) xor C(13) xor
        C(14) xor C(16) xor C(18) xor C(19) xor C(22) xor C(25) xor
        C(28) xor C(31);
NewCRC(14) <= D(29) xor D(26) xor D(23) xor D(20) xor D(19) xor D(17) xor
        D(15) xor D(14) xor D(11) xor D(8) xor D(7) xor D(6) xor
        D(4) xor D(3) xor D(2) xor C(2) xor C(3) xor C(4) xor
        C(6) xor C(7) xor C(8) xor C(11) xor C(14) xor C(15) xor
        C(17) xor C(19) xor C(20) xor C(23) xor C(26) xor C(29);
NewCRC(15) <= D(30) xor D(27) xor D(24) xor D(21) xor D(20) xor D(18) xor
        D(16) xor D(15) xor D(12) xor D(9) xor D(8) xor D(7) xor
        D(5) xor D(4) xor D(3) xor C(3) xor C(4) xor C(5) xor
        C(7) xor C(8) xor C(9) xor C(12) xor C(15) xor C(16) xor
        C(18) xor C(20) xor C(21) xor C(24) xor C(27) xor C(30);
NewCRC(16) <= D(30) xor D(29) xor D(26) xor D(24) xor D(22) xor D(21) xor
        D(19) xor D(17) xor D(13) xor D(12) xor D(8) xor D(5) xor
        D(4) xor D(0) xor C(0) xor C(4) xor C(5) xor C(8) xor
        C(12) xor C(13) xor C(17) xor C(19) xor C(21) xor C(22) xor
        C(24) xor C(26) xor C(29) xor C(30);
NewCRC(17) <= D(31) xor D(30) xor D(27) xor D(25) xor D(23) xor D(22) xor
        D(20) xor D(18) xor D(14) xor D(13) xor D(9) xor D(6) xor
        D(5) xor D(1) xor C(1) xor C(5) xor C(6) xor C(9) xor
        C(13) xor C(14) xor C(18) xor C(20) xor C(22) xor C(23) xor
        C(25) xor C(27) xor C(30) xor C(31);
NewCRC(18) <= D(31) xor D(28) xor D(26) xor D(24) xor D(23) xor D(21) xor
        D(19) xor D(15) xor D(14) xor D(10) xor D(7) xor D(6) xor
        D(2) xor C(2) xor C(6) xor C(7) xor C(10) xor C(14) xor
        C(15) xor C(19) xor C(21) xor C(23) xor C(24) xor C(26) xor
        C(28) xor C(31);
NewCRC(19) <= D(29) xor D(27) xor D(25) xor D(24) xor D(22) xor D(20) xor
        D(16) xor D(15) xor D(11) xor D(8) xor D(7) xor D(3) xor
        C(3) xor C(7) xor C(8) xor C(11) xor C(15) xor C(16) xor
        C(20) xor C(22) xor C(24) xor C(25) xor C(27) xor C(29);
NewCRC(20) <= D(30) xor D(28) xor D(26) xor D(25) xor D(23) xor D(21) xor
        D(17) xor D(16) xor D(12) xor D(9) xor D(8) xor D(4) xor
        C(4) xor C(8) xor C(9) xor C(12) xor C(16) xor C(17) xor
        C(21) xor C(23) xor C(25) xor C(26) xor C(28) xor C(30);
NewCRC(21) <= D(31) xor D(29) xor D(27) xor D(26) xor D(24) xor D(22) xor
        D(18) xor D(17) xor D(13) xor D(10) xor D(9) xor D(5) xor
        C(5) xor C(9) xor C(10) xor C(13) xor C(17) xor C(18) xor
        C(22) xor C(24) xor C(26) xor C(27) xor C(29) xor C(31);
NewCRC(22) <= D(31) xor D(29) xor D(27) xor D(26) xor D(24) xor D(23) xor
        D(19) xor D(18) xor D(16) xor D(14) xor D(12) xor D(11) xor
        D(9) xor D(0) xor C(0) xor C(9) xor C(11) xor C(12) xor
        C(14) xor C(16) xor C(18) xor C(19) xor C(23) xor C(24) xor
        C(26) xor C(27) xor C(29) xor C(31);
NewCRC(23) <= D(31) xor D(29) xor D(27) xor D(26) xor D(20) xor D(19) xor
        D(17) xor D(16) xor D(15) xor D(13) xor D(9) xor D(6) xor
        D(1) xor D(0) xor C(0) xor C(1) xor C(6) xor C(9) xor
        C(13) xor C(15) xor C(16) xor C(17) xor C(19) xor C(20) xor
        C(26) xor C(27) xor C(29) xor C(31);
NewCRC(24) <= D(30) xor D(28) xor D(27) xor D(21) xor D(20) xor D(18) xor
        D(17) xor D(16) xor D(14) xor D(10) xor D(7) xor D(2) xor
        D(1) xor C(1) xor C(2) xor C(7) xor C(10) xor C(14) xor
        C(16) xor C(17) xor C(18) xor C(20) xor C(21) xor C(27) xor
        C(28) xor C(30);
NewCRC(25) <= D(31) xor D(29) xor D(28) xor D(22) xor D(21) xor D(19) xor
        D(18) xor D(17) xor D(15) xor D(11) xor D(8) xor D(3) xor
        D(2) xor C(2) xor C(3) xor C(8) xor C(11) xor C(15) xor
```

```
           C(17) xor C(18) xor C(19) xor C(21) xor C(22) xor C(28) xor
           C(29) xor C(31);
  NewCRC(26) <= D(31) xor D(28) xor D(26) xor D(25) xor D(24) xor D(23) xor
           D(22) xor D(20) xor D(19) xor D(18) xor D(10) xor D(6) xor
           D(4) xor D(3) xor D(0) xor C(0) xor C(3) xor C(4) xor
           C(6) xor C(10) xor C(18) xor C(19) xor C(20) xor C(22) xor
           C(23) xor C(24) xor C(25) xor C(26) xor C(28) xor C(31);
  NewCRC(27) <= D(29) xor D(27) xor D(26) xor D(25) xor D(24) xor D(23) xor
           D(21) xor D(20) xor D(19) xor D(11) xor D(7) xor D(5) xor
           D(4) xor D(1) xor C(1) xor C(4) xor C(5) xor C(7) xor
           C(11) xor C(19) xor C(20) xor C(21) xor C(23) xor C(24) xor
           C(25) xor C(26) xor C(27) xor C(29);
  NewCRC(28) <= D(30) xor D(28) xor D(27) xor D(26) xor D(25) xor D(24) xor
           D(22) xor D(21) xor D(20) xor D(12) xor D(8) xor D(6) xor
           D(5) xor D(2) xor C(2) xor C(5) xor C(6) xor C(8) xor
           C(12) xor C(20) xor C(21) xor C(22) xor C(24) xor C(25) xor
           C(26) xor C(27) xor C(28) xor C(30);
  NewCRC(29) <= D(31) xor D(29) xor D(28) xor D(27) xor D(26) xor D(25) xor
           D(23) xor D(22) xor D(21) xor D(13) xor D(9) xor D(7) xor
           D(6) xor D(3) xor C(3) xor C(6) xor C(7) xor C(9) xor
           C(13) xor C(21) xor C(22) xor C(23) xor C(25) xor C(26) xor
           C(27) xor C(28) xor C(29) xor C(31);
  NewCRC(30) <= D(30) xor D(29) xor D(28) xor D(27) xor D(26) xor D(24) xor
           D(23) xor D(22) xor D(14) xor D(10) xor D(8) xor D(7) xor
           D(4) xor C(4) xor C(7) xor C(8) xor C(10) xor C(14) xor
           C(22) xor C(23) xor C(24) xor C(26) xor C(27) xor C(28) xor
           C(29) xor C(30);
  NewCRC(31) <= D(31) xor D(30) xor D(29) xor D(28) xor D(27) xor D(25) xor
           D(24) xor D(23) xor D(15) xor D(11) xor D(9) xor D(8) xor
           D(5) xor C(5) xor C(8) xor C(9) xor C(11) xor C(15) xor
           C(23) xor C(24) xor C(25) xor C(27) xor C(28) xor C(29) xor
           C(30) xor C(31);
end process;
end architecture crc_beh;

-- CRC Compare

library IEEE;
use IEEE.std_logic_1164.all;

entity compcrc is
port(calccrc, crcin: in std_logic_vector(31 downto 0);
   crcchkout: out std_logic);
end entity compcrc;

architecture compcrc_beh of compcrc is

component XOR2 is
port(I0, I1: in std_logic;
   O: out std_logic);
end component XOR2;

component OR16 is
port(I6, I9, I8, I7, I5, I4, I3, I2, I15, I14, I13, I12, I11, I10, I1, I0: in std_logic;
   O: out std_logic);
end component OR16;
```

```
signal x: std_logic_vector(31 downto 0);
signal EQ1, EQ2: std_logic;
begin

xorcomp1: XOR2 port map(I0=>calccrc(0), I1=>crcin(0), O=>x(0));
xorcomp2: XOR2 port map(I0=>calccrc(1), I1=>crcin(1), O=>x(1));
xorcomp3: XOR2 port map(I0=>calccrc(2), I1=>crcin(2), O=>x(2));
xorcomp4: XOR2 port map(I0=>calccrc(3), I1=>crcin(3), O=>x(3));
xorcomp5: XOR2 port map(I0=>calccrc(4), I1=>crcin(4), O=>x(4));
xorcomp6: XOR2 port map(I0=>calccrc(5), I1=>crcin(5), O=>x(5));
xorcomp7: XOR2 port map(I0=>calccrc(6), I1=>crcin(6), O=>x(6));
xorcomp8: XOR2 port map(I0=>calccrc(7), I1=>crcin(7), O=>x(7));
xorcomp9: XOR2 port map(I0=>calccrc(8), I1=>crcin(8), O=>x(8));
xorcomp10: XOR2 port map(I0=>calccrc(9), I1=>crcin(9), O=>x(9));
xorcomp11: XOR2 port map(I0=>calccrc(10), I1=>crcin(10), O=>x(10));
xorcomp12: XOR2 port map(I0=>calccrc(11), I1=>crcin(11), O=>x(11));
xorcomp13: XOR2 port map(I0=>calccrc(12), I1=>crcin(12), O=>x(12));
xorcomp14: XOR2 port map(I0=>calccrc(13), I1=>crcin(13), O=>x(13));
xorcomp15: XOR2 port map(I0=>calccrc(14), I1=>crcin(14), O=>x(14));
xorcomp16: XOR2 port map(I0=>calccrc(15), I1=>crcin(15), O=>x(15));
xorcomp17: XOR2 port map(I0=>calccrc(16), I1=>crcin(16), O=>x(16));
xorcomp18: XOR2 port map(I0=>calccrc(17), I1=>crcin(17), O=>x(17));
xorcomp19: XOR2 port map(I0=>calccrc(18), I1=>crcin(18), O=>x(18));
xorcomp20: XOR2 port map(I0=>calccrc(19), I1=>crcin(19), O=>x(19));
xorcomp21: XOR2 port map(I0=>calccrc(20), I1=>crcin(20), O=>x(20));
xorcomp22: XOR2 port map(I0=>calccrc(21), I1=>crcin(21), O=>x(21));
xorcomp23: XOR2 port map(I0=>calccrc(22), I1=>crcin(22), O=>x(22));
xorcomp24: XOR2 port map(I0=>calccrc(23), I1=>crcin(23), O=>x(23));
xorcomp25: XOR2 port map(I0=>calccrc(24), I1=>crcin(24), O=>x(24));
xorcomp26: XOR2 port map(I0=>calccrc(25), I1=>crcin(25), O=>x(25));
xorcomp27: XOR2 port map(I0=>calccrc(26), I1=>crcin(26), O=>x(26));
xorcomp28: XOR2 port map(I0=>calccrc(27), I1=>crcin(27), O=>x(27));
xorcomp29: XOR2 port map(I0=>calccrc(28), I1=>crcin(28), O=>x(28));
xorcomp30: XOR2 port map(I0=>calccrc(29), I1=>crcin(29), O=>x(29));
xorcomp31: XOR2 port map(I0=>calccrc(30), I1=>crcin(30), O=>x(30));
xorcomp32: XOR2 port map(I0=>calccrc(31), I1=>crcin(31), O=>x(31));

orcomp1: OR16 port map(I6=>x(6), I9=>x(9), I8=>x(8), I7=>x(7), I5=>x(5), I4=>x(4), I3=>x(3),
I2=>x(2), I15=>x(15), I14=>x(14), I13=>x(13), I12=>x(12), I11=>x(11), I10=>x(10), I1=>x(1), I0=>x(0),
O=>EQ1);
orcomp2: OR16 port map(I6=>x(16), I9=>x(17), I8=>x(18), I7=>x(19), I5=>x(20), I4=>x(21), I3=>x(22),
I2=>x(23), I15=>x(24), I14=>x(25), I13=>x(26), I12=>x(27), I11=>x(28), I10=>x(29), I1=>x(30),
I0=>x(31), O=>EQ2);

crcchkout <= not (EQ1 or EQ2);
end architecture compcrc_beh;

-- CRC OUT

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity crcout is
port(ldcr, crcz: in std_logic;
    crcoutin, cin2: in std_logic_vector(31 downto 0);
    crcoutout: out std_logic_vector(31 downto 0));
end entity crcout;

architecture crcout_beh of crcout is
signal sig: std_logic_vector(1 downto 0);
begin

sig <= crcz&ldcr;
process(sig, crcoutin, cin2) is
begin
case sig is
when "00" =>  crcoutout <= crcoutin;
when "01" =>  crcoutout <= cin2;
when "10" => crcoutout <= (others => '0');
when "11" => crcoutout <= (others => '0');
when others => crcoutout <= (others => '0');
end case;
end process;
end architecture crcout_beh;

-- CRCCALCULATED VALUE STORE

library IEEE;
use IEEE.std_logic_1164.all;

entity crcst is
port(clk : in std_logic;
    crc_calc_in : in std_logic_vector(31 downto 0);
    crc_calc_out  : out std_logic_vector(31 downto 0));
end entity crcst;

architecture crcst_beh of crcst is
begin

process(clk, crc_calc_in)
begin
if (rising_edge(clk))then
crc_calc_out <= crc_calc_in;
end if;
end process;
end architecture crcst_beh;

-- CRC OUTRAM MODULE
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity crcout_ram is
port(epout: in std_logic;
    crc_cin, outramin: in std_logic_vector(31 downto 0);
    outramout: out std_logic_vector(31 downto 0));
end entity crcout_ram;
```

```
architecture crcout_ram_beh of crcout_ram is
begin

process(epout, crc_cin, outramin) is
begin
case epout is
when '0' =>  outramout <= outramin;
when '1' =>  outramout <= crc_cin;
when others => outramout <= (others => '0');
end case;
end process;
end architecture crcout_ram_beh;

-- For getting instructions
library IEEE;
use IEEE.std_logic_1164.all;

entity tstin is
port(in_inst, clk: in std_logic;
    ininstout: out std_logic);
end entity tstin;

architecture tstin_beh of tstin is
component FD is
port(D, C: in std_logic;
    Q: out std_logic);
end component FD;

component gdi is
port(clk, fb: in std_logic;
    diout: out std_logic);
end component gdi;
signal inbar0, inbar1, in0, in1, ininstout1: std_logic;
begin

dff_st0: FD port map(D=>in_inst, C=>clk, Q=>in0);
dff_st1: FD port map(D=>in0, C=>clk, Q=>in1);
inbar0 <= not(in0);
inbar1 <= not(in1);

gdicomp: gdi port map(clk=>clk, fb=>inbar0, diout=>ininstout1);
ininstout <= in_inst and ininstout1;

end architecture tstin_beh;

-- Getting desired inst
library IEEE;
use IEEE.std_logic_1164.all;

entity gdi is
port(clk, fb: in std_logic;
    diout: out std_logic);
end entity gdi;

architecture gdi_beh of gdi is
begin
```

```
process(clk, fb) is
begin
if(falling_edge(clk)) then
diout <= fb;
end if;
end process;
end architecture gdi_beh;

-- 8 bit FLAG register
library IEEE;
use IEEE.std_logic_1164.all;

entity aereg is
port(clk, ldaer : in std_logic;
    flagval_in : in std_logic_vector(7 downto 0);
    aerout  : out std_logic_vector(7 downto 0));
end entity aereg;

architecture aereg_beh of aereg is
begin
process(clk, ldaer, flagval_in)
begin
if (rising_edge(clk))then
if (ldaer = '1') then
aerout <= flagval_in;
end if;
end if;
end process;
end architecture aereg_beh;

-- 8 bit OCR register
library IEEE;
use IEEE.std_logic_1164.all;

entity ocreg is
port(clk, ldocr : in std_logic;
    val_in : in std_logic_vector(7 downto 0);
    ocrout  : out std_logic_vector(7 downto 0));
end entity ocreg;

architecture ocreg_beh of ocreg is
begin
process(clk, ldocr, val_in)
begin
if (rising_edge(clk))then
if (ldocr = '1') then
ocrout <= val_in;
end if;
end if;
end process;
end architecture ocreg_beh;

-- 6 bit MOR register
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity moreg is
port(clk, ldmor : in std_logic;
    mop_fmpkt_in : in std_logic_vector(5 downto 0);
    mopout  : out std_logic_vector(5 downto 0));
end entity moreg;

architecture moreg_beh of moreg is
begin
process(clk, ldmor, mop_fmpkt_in)
begin
if (rising_edge(clk))then
if (ldmor = '1') then
mopout <= mop_fmpkt_in;
end if;
end if;
end process;
end architecture moreg_beh;

--ESS
--FULL ESS from 3 Stages

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

entity esstop0 is
port(tag_in, value_in: in std_logic_vector(63 downto 0);
    clk, clock, ess_we, ess_re, putin: in std_logic;
    gf, pf, ess_full, le: out std_logic;
    outvalue: out std_logic_vector(63 downto 0));
end entity esstop0;

architecture fulless_beh of esstop0 is
--Components
component First_Stage is
port(tag_in: in std_logic_vector(63 downto 0);
    clk, put, get, empty, putin_fmTS, match_fmTS: in std_logic;
    fmmux_addr: in std_logic_vector(4 downto 0);
    matchout, putout, getout: out std_logic;
    match_outaddr: out std_logic_vector(4 downto 0));
end component First_Stage;

component Second_Stage is
port(clk, clock, get, put, empin_fmTS, lexpd_fmTS, put_fmTS, matchin_fmTS: in std_logic;
    matchin_fmFS: in std_logic;
    mataddr_fmFS: in std_logic_vector(4 downto 0);
    matchout_sec, putout_sec, getout_sec: out std_logic;
    emptysig_out_sec, life_expd_out_sec, GF_out, PF_out: out std_logic;
    mux_outaddr_sec: out std_logic_vector(4 downto 0));
end component Second_Stage;
```

```
component TSPE is
port(clk, get, put: in std_logic;
    GF_fmsec, PF_fmsec, empty_fmsec, lifeexpd_fmsec, match_fmsec: in std_logic;
    value_in: in std_logic_vector(63 downto 0);
    muxaddr_fmsec: in std_logic_vector(4 downto 0);
    GFOUT, PFOUT, ESSFULL, le: out std_logic;
    OUTVALUE: out std_logic_vector(63 downto 0));
end component TSPE;

----signals
----FS signals
signal matchout_FS, getout_FS, putout_FS: std_logic;
signal muxaddr_fmsecst, matchaddr_FS: std_logic_vector(4 downto 0);
--SS signals
signal match_SS, EO_SS, put_SS, get_SS, LE_SS, GF_SS, PF_SS: std_logic;
----TS signals
signal gf_TS, pf_TS: std_logic;
begin

gf <= GF_SS;
pf <= PF_SS;

FS_comp: First_Stage port map(tag_in=>tag_in, clk=>clk, put=>ess_we, get=>ess_re, empty=>EO_SS,
putin_fmTS=>putin, match_fmTS=>match_SS, fmmux_addr=>muxaddr_fmsecst,
matchout=>matchout_FS, putout=>putout_FS, getout=>getout_FS, match_outaddr=>matchaddr_FS);

SS_comp: Second_Stage port map(clk=>clk, clock=>clock, get=>getout_FS, put=>putout_FS,
empin_fmTS=>EO_SS, lexpd_fmTS=>LE_SS, put_fmTS=>put_SS, matchin_fmTS=>match_SS,
matchin_fmFS=>matchout_FS, mataddr_fmFS=>matchaddr_FS, matchout_sec=>match_SS,
putout_sec=>put_SS, getout_sec=>get_SS, emptysig_out_sec=>EO_SS, life_expd_out_sec=>LE_SS,
GF_out=>GF_SS, PF_out=>PF_SS, mux_outaddr_sec=>muxaddr_fmsecst);

TS_comp: TSPE port map(clk=>clk, get=>get_SS, put=>put_SS, GF_fmsec=>GF_SS,
PF_fmsec=>PF_SS, empty_fmsec=>EO_SS, lifeexpd_fmsec=>LE_SS, match_fmsec=>match_SS,
value_in=>value_in, muxaddr_fmsec=>muxaddr_fmsecst, GFOUT=>gf_TS, PFOUT=>pf_TS,
ESSFULL=>ess_full, le=>le, OUTVALUE=>outvalue);

end architecture fulless_beh;

--Individual Components
--First Stage Pipeline ESS

library IEEE;
use IEEE.std_logic_1164.all;

entity First_Stage is
port(tag_in: in std_logic_vector(63 downto 0);
    clk, put, get, empty, putin_fmTS, match_fmTS: in std_logic;
    fmmux_addr: in std_logic_vector(4 downto 0);
    matchout, putout, getout: out std_logic;
    match_outaddr: out std_logic_vector(4 downto 0));
end entity First_Stage;

architecture First_Stage_beh of First_Stage is
--components
component FSPE is
```

```
port(tag_in: in std_logic_vector(63 downto 0);
    clk, put, get, empty, putin_fmTS, match_fmTS: in std_logic;
    fmmux_addr: in std_logic_vector(4 downto 0);
    matchsig: out std_logic;
    matchoutaddr: out std_logic_vector(4 downto 0));
end component FSPE;

component FSL is
port(clk: in std_logic;
    putin, getin, matchin: in std_logic;
    match_inaddr: in std_logic_vector(4 downto 0);
    putout, getout, matchout: out std_logic;
    match_outaddr: out std_logic_vector(4 downto 0));
end component FSL;

--signals
signal mat_signal: std_logic;
signal mat_address: std_logic_vector(4 downto 0);
begin

FSPE_comp: FSPE port map(tag_in=>tag_in, clk=>clk, put=>put, get=>get, empty=>empty,
putin_fmTS=>putin_fmTS, match_fmTS=>match_fmTS, fmmux_addr=>fmmux_addr,
matchsig=>mat_signal, matchoutaddr=>mat_address);
FSL_comp: FSL port map(clk=>clk, putin=>put, getin=>get, matchin=>mat_signal,
match_inaddr=>mat_address, putout=>putout, getout=>getout, matchout=>matchout,
match_outaddr=>match_outaddr);

end architecture First_Stage_beh;

--Individual Components
--First Stage of Pipeline ESS

library IEEE;
use IEEE.std_logic_1164.all;

entity FSPE is
port(tag_in: in std_logic_vector(63 downto 0);
    clk, put, get, empty, putin_fmTS, match_fmTS: in std_logic;
    fmmux_addr: in std_logic_vector(4 downto 0);
    matchsig: out std_logic;
    matchoutaddr: out std_logic_vector(4 downto 0));
end entity FSPE;

architecture FSPE_beh of FSPE is
component camfull is
port(tag_in         : in std_logic_vector(63 downto 0) ;
ADDR                : in std_logic_vector(4 downto 0) ;
WRITE_ENABLE        : in std_logic;
ERASE_WRITE : in std_logic;
WRITE_RAM           : in std_logic;
CLK                 : in std_logic;
MATCH_ENABLE        : in std_logic;
MATCH_RST      : in std_logic;
MATCH_SIG_OUT       : out std_logic;
MATCH_ADDR : out std_logic_vector(4 downto 0));
end component camfull;
```

```
signal pg, write: std_logic;
begin

pg <= put or get;
write <= ( empty and putin_fmTS and (not(match_fmTS)) ); -- has to be empty and put (not empty alone)
camfull_comp: camfull port map(tag_in=>tag_in, ADDR=>fmmux_addr, WRITE_ENABLE=>write,
ERASE_WRITE=>write, WRITE_RAM=>write, CLK=>clk, MATCH_ENABLE=>pg,
MATCH_RST=>pg, MATCH_SIG_OUT=>matchsig, MATCH_ADDR=>matchoutaddr);

end architecture FSPE_beh;

--Full Original CAM
-- single CAM module

library IEEE;
use IEEE.std_logic_1164.all;

entity camfull is
port(     tag_in    : in std_logic_vector(63 downto 0) ;
          ADDR              : in std_logic_vector(4 downto 0) ;
          WRITE_ENABLE            : in std_logic;
          ERASE_WRITE : in std_logic;
          WRITE_RAM     : in std_logic;
          CLK                   : in std_logic;
          MATCH_ENABLE            : in std_logic;
          MATCH_RST     : in std_logic;
          MATCH_SIG_OUT           : out std_logic;
          MATCH_ADDR : out std_logic_vector(4 downto 0));
end entity camfull;

architecture camfull_beh of camfull is
component cam16x64_1 is
port(     tag_in: in std_logic_vector(63 downto 0);
          ADDR              : in std_logic_vector(3 downto 0) ; -- Used by erase/write operation only
          WRITE_ENABLE            : in std_logic; -- Write Enable during 2 clock cycles
          ERASE_WRITE : in std_logic; -- if '0' ERASE else WRITE, generate from WRITE_ENABLE at
the CAMs' top level
          WRITE_RAM     : in std_logic; -- if '1' DATA_IN is WRITE in the RAM16x1s, generate from
WRITE_ENABLE at the CAMs' top level
          CLK                   : in std_logic;
          MATCH_ENABLE            : in std_logic;
          MATCH_RST     : in std_logic; -- Synchronous reset => MATCH = "0000000000000000"
          MATCH                   : out std_logic_vector(15 downto 0));
end component cam16x64_1;

component ENCODE_4_LSB is
port(  BINARY_ADDR   : in std_logic_vector(31 downto 0);
     MATCH_ADDR      : out std_logic_vector(4 downto 0); -- Match address found
     MATCH_OK              : out std_logic); -- '1' if Match found
end component ENCODE_4_LSB;

signal match_sig1: std_logic_vector(15 downto 0);
signal match_sig2: std_logic_vector(15 downto 0);
signal match_sig: std_logic_vector(31 downto 0);
signal WE_1, WE_2, EW_1, EW_2, WR_1, WR_2, adnot: std_logic;
begin
```

```
adnot <= not (ADDR(4));
WE_1 <= adnot and WRITE_ENABLE;
WE_2 <= ADDR(4) and WRITE_ENABLE;
EW_1 <= adnot and ERASE_WRITE;
EW_2 <= ADDR(4) and ERASE_WRITE;
WR_1 <= adnot and WRITE_RAM;
WR_2 <= ADDR(4) and WRITE_RAM;

camfinal0: cam16x64_1 port map(tag_in=>tag_in, ADDR=>ADDR(3 downto 0),
WRITE_ENABLE=>WE_1, ERASE_WRITE=>EW_1, WRITE_RAM=>WR_1, CLK=>CLK,
MATCH_ENABLE=>MATCH_ENABLE, MATCH_RST=>MATCH_RST, MATCH=>match_sig1);
camfinal1: cam16x64_1 port map(tag_in=>tag_in, ADDR=>ADDR(3 downto 0),
WRITE_ENABLE=>WE_2, ERASE_WRITE=>EW_2, WRITE_RAM=>WR_2, CLK=>CLK,
MATCH_ENABLE=>MATCH_ENABLE, MATCH_RST=>MATCH_RST, MATCH=>match_sig2);

match_sig <= match_sig2&match_sig1;
encoder: ENCODE_4_LSB port map(BINARY_ADDR=>match_sig,
MATCH_ADDR=>MATCH_ADDR, MATCH_OK=>MATCH_SIG_OUT);

end architecture camfull_beh;

--CAM 16x64

library IEEE;
use IEEE.std_logic_1164.all;

entity cam16x64_1 is
port(     tag_in: in std_logic_vector(63 downto 0);
          ADDR                : in std_logic_vector(3 downto 0) ; -- Used by erase/write operation only
          WRITE_ENABLE                : in std_logic; -- Write Enable during 2 clock cycles
          ERASE_WRITE : in std_logic; -- if '0' ERASE else WRITE, generate from WRITE_ENABLE at
the CAMs' top level
          WRITE_RAM     : in std_logic; -- if '1' DATA_IN is WRITE in the RAM16x1s, generate from
WRITE_ENABLE at the CAMs' top level
          CLK                 : in std_logic;
          MATCH_ENABLE             : in std_logic;
          MATCH_RST     : in std_logic; -- Synchronous reset => MATCH = "0000000000000000"
          MATCH                       : out std_logic_vector(15 downto 0));
end entity cam16x64_1;

architecture camtry1_beh of cam16x64_1 is

component CAM_RAMB4 is
port(     DATA_IN           : in std_logic_vector(7 downto 0) ; -- Data to compare or to write
          ADDR                : in std_logic_vector(3 downto 0) ; -- Used by erase/write operation only
          WRITE_ENABLE             : in std_logic; -- Write Enable during 2 clock cycles
          ERASE_WRITE : in std_logic; -- if '0' ERASE else WRITE, generate from WRITE_ENABLE at
the CAMs' top level
          WRITE_RAM     : in std_logic; -- if '1' DATA_IN is WRITE in the RAM16x1s, generate from
WRITE_ENABLE at the CAMs' top level
          CLK                 : in std_logic;
          MATCH_ENABLE           : in std_logic;
          MATCH_RST     : in std_logic; -- Synchronous reset => MATCH = "0000000000000000"
          MATCH_OUT     : out std_logic_vector(15 downto 0));
end component CAM_RAMB4;
```

```
signal match_out0, match_out1, match_out2, match_out3, match_out4, match_out5, match_out6,
match_out7: std_logic_vector(15 downto 0);
begin

camtry0: CAM_RAMB4 port map(DATA_IN=>tag_in(63 downto 56), ADDR=>ADDR,
WRITE_ENABLE=>WRITE_ENABLE, ERASE_WRITE=>ERASE_WRITE,
WRITE_RAM=>WRITE_RAM, CLK=>CLK, MATCH_ENABLE=>MATCH_ENABLE,
MATCH_RST=>MATCH_RST, MATCH_OUT=>match_out0);

camtry1: CAM_RAMB4 port map(DATA_IN=>tag_in(55 downto 48), ADDR=>ADDR,
WRITE_ENABLE=>WRITE_ENABLE, ERASE_WRITE=>ERASE_WRITE,
WRITE_RAM=>WRITE_RAM, CLK=>CLK, MATCH_ENABLE=>MATCH_ENABLE,
MATCH_RST=>MATCH_RST, MATCH_OUT=>match_out1);

camtry2: CAM_RAMB4 port map(DATA_IN=>tag_in(47 downto 40), ADDR=>ADDR,
WRITE_ENABLE=>WRITE_ENABLE, ERASE_WRITE=>ERASE_WRITE,
WRITE_RAM=>WRITE_RAM, CLK=>CLK, MATCH_ENABLE=>MATCH_ENABLE,
MATCH_RST=>MATCH_RST, MATCH_OUT=>match_out2);

camtry3: CAM_RAMB4 port map(DATA_IN=>tag_in(39 downto 32), ADDR=>ADDR,
WRITE_ENABLE=>WRITE_ENABLE, ERASE_WRITE=>ERASE_WRITE,
WRITE_RAM=>WRITE_RAM, CLK=>CLK, MATCH_ENABLE=>MATCH_ENABLE,
MATCH_RST=>MATCH_RST, MATCH_OUT=>match_out3);

camtry4: CAM_RAMB4 port map(DATA_IN=>tag_in(31 downto 24), ADDR=>ADDR,
WRITE_ENABLE=>WRITE_ENABLE, ERASE_WRITE=>ERASE_WRITE,
WRITE_RAM=>WRITE_RAM, CLK=>CLK, MATCH_ENABLE=>MATCH_ENABLE,
MATCH_RST=>MATCH_RST, MATCH_OUT=>match_out4);

camtry5: CAM_RAMB4 port map(DATA_IN=>tag_in(23 downto 16), ADDR=>ADDR,
WRITE_ENABLE=>WRITE_ENABLE, ERASE_WRITE=>ERASE_WRITE,
WRITE_RAM=>WRITE_RAM, CLK=>CLK, MATCH_ENABLE=>MATCH_ENABLE,
MATCH_RST=>MATCH_RST, MATCH_OUT=>match_out5);

camtry6: CAM_RAMB4 port map(DATA_IN=>tag_in(15 downto 8), ADDR=>ADDR,
WRITE_ENABLE=>WRITE_ENABLE, ERASE_WRITE=>ERASE_WRITE,
WRITE_RAM=>WRITE_RAM, CLK=>CLK, MATCH_ENABLE=>MATCH_ENABLE,
MATCH_RST=>MATCH_RST, MATCH_OUT=>match_out6);

camtry7: CAM_RAMB4 port map(DATA_IN=>tag_in(7 downto 0), ADDR=>ADDR,
WRITE_ENABLE=>WRITE_ENABLE, ERASE_WRITE=>ERASE_WRITE,
WRITE_RAM=>WRITE_RAM, CLK=>CLK, MATCH_ENABLE=>MATCH_ENABLE,
MATCH_RST=>MATCH_RST, MATCH_OUT=>match_out7);

MATCH <= match_out0 and match_out1 and match_out2 and match_out3 and match_out4 and
match_out5 and match_out6 and match_out7;
end architecture camtry1_beh;

-- Individual CAM module

library IEEE;
use IEEE.std_logic_1164.all;

entity CAM_RAMB4 is
port(     DATA_IN          : in std_logic_vector(7 downto 0) ; -- Data to compare or to write
          ADDR             : in std_logic_vector(3 downto 0) ; -- Used by erase/write operation only
```

```
        WRITE_ENABLE        : in std_logic; -- Write Enable during 2 clock cycles
        ERASE_WRITE : in std_logic; -- if '0' ERASE else WRITE, generate from WRITE_ENABLE at
the CAMs' top level
        WRITE_RAM      : in std_logic; -- if '1' DATA_IN is WRITE in the RAM16x1s, generate from
WRITE_ENABLE at the CAMs' top level
        CLK            : in std_logic;
        MATCH_ENABLE        : in std_logic;
        MATCH_RST      : in std_logic; -- Synchronous reset => MATCH = "0000000000000000"
        MATCH_OUT      : out std_logic_vector(15 downto 0));
end CAM_RAMB4;

architecture CAM_RAMB4_arch of CAM_RAMB4 is
-- Components Declarations:
component INIT_8_RAM16x1s
port(   DATA_IN        : in std_logic_vector(7 downto 0);
        ADDR           : in std_logic_vector(3 downto 0);
        WRITE_RAM      : in std_logic;
        CLK            : in std_logic;
        DATA_WRITE     : out std_logic_vector(7 downto 0));
end component;

component INIT_RAMB4_S1_S16
port(   DIA     : in std_logic;
        ENA     : in std_logic;
        ENB     : in std_logic;
        WEA     : in std_logic;
        RSTB    : in std_logic;
        CLK     : in std_logic;
        ADDRA           : in std_logic_vector (11 downto 0);
        ADDRB           : in std_logic_vector (7 downto 0);
        DOB     : out std_logic_vector (15 downto 0));
end component;

-- Signal Declarations:
signal DATA_WRITE : std_logic_vector(7 downto 0);          -- Data to be written in the RAMB4
signal ADDR_WRITE : std_logic_vector(11 downto 0);         -- Combine write address from ADDR and
DATA_WRITE
signal B_MATCH_RST: std_logic; -- inverter MATCH_RST active high
begin

B_MATCH_RST <= not MATCH_RST;

-- SelectRAM instantiation = 8 x RAM16x1s_1
RAM_ERASE: INIT_8_RAM16x1s
        port map (
        DATA_IN => DATA_IN,
        ADDR => ADDR,
        WRITE_RAM => WRITE_RAM,
        CLK => CLK,
        DATA_WRITE => DATA_WRITE
        );
--
-- Select the write data for addressing
ADDR_WRITE(3 downto 0) <= ADDR(3 downto 0);
ADDR_WRITE(11 downto 4) <= DATA_WRITE(7 downto 0);
```

```
-- Select BlockRAM RAMB4_S1_S16 instantiation
RAMB4 : INIT_RAMB4_S1_S16
        port map (
        DIA => ERASE_WRITE,
        ENA => WRITE_ENABLE,
        ENB => MATCH_ENABLE,
        WEA => WRITE_ENABLE,
        RSTB => B_MATCH_RST,
        CLK => CLK,
        ADDRA => ADDR_WRITE(11 downto 0),
        ADDRB => DATA_IN(7 downto 0),
        DOB => MATCH_OUT(15 downto 0) );

end CAM_RAMB4_arch;

-- Init_RAMB4_S1_S16 module
library IEEE;
use IEEE.std_logic_1164.all;

entity INIT_RAMB4_S1_S16 is
    port (
            DIA     : in std_logic;
            ENA     : in std_logic;
            ENB     : in std_logic;
            WEA     : in std_logic;
            RSTB    : in std_logic;
            CLK     : in std_logic; -- Same clock on ports A & B
            ADDRA           : in std_logic_vector (11 downto 0);
            ADDRB           : in std_logic_vector (7 downto 0);
            DOB     : out std_logic_vector (15 downto 0)
            ); -- unused input ports are tied to GND
end INIT_RAMB4_S1_S16;

architecture INIT_RAMB4_S1_S16_arch of INIT_RAMB4_S1_S16 is
component RAMB4_S1_S16
-- pragma synthesis_off
generic(
      INIT_00 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
      INIT_01 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
      INIT_02 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
      INIT_03 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
      INIT_04 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
      INIT_05 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
      INIT_06 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
      INIT_07 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
      INIT_08 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
```

```
     INIT_09 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
     INIT_0A : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
     INIT_0B : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
     INIT_0C : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
     INIT_0D : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
     INIT_0E : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
     INIT_0F : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000"
  );
-- pragma synthesis_on
   port (
        DIA      : in std_logic_vector(0 downto 0);
        DIB      : in std_logic_vector (15 downto 0);
        ENA      : in std_logic;
        ENB      : in std_logic;
        WEA      : in std_logic;
        WEB      : in std_logic;
        RSTA     : in std_logic;
        RSTB     : in std_logic;
        CLKA     : in std_logic;
        CLKB     : in std_logic;
        ADDRA             : in std_logic_vector (11 downto 0);
        ADDRB             : in std_logic_vector (7 downto 0);
        DOA      : out std_logic_vector (0 downto 0);
        DOB      : out std_logic_vector (15 downto 0)
   );
 end component;

 attribute INIT_00: string;
 attribute INIT_01: string;
 attribute INIT_02: string;
 attribute INIT_03: string;
 attribute INIT_04: string;
 attribute INIT_05: string;
 attribute INIT_06: string;
 attribute INIT_07: string;
 attribute INIT_08: string;
 attribute INIT_09: string;
 attribute INIT_0A: string;
 attribute INIT_0B: string;
 attribute INIT_0C: string;
 attribute INIT_0D: string;
 attribute INIT_0E: string;
 attribute INIT_0F: string;

  attribute INIT_00 of RAMB4: label is
 "0000000000000000000000000000000000000000000000000000000000000000";
  attribute INIT_01 of RAMB4: label is
 "0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_02 of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of RAMB4: label is
"0000000000000000000000000000000000000000000000000000000000000000";

-- Signal Declarations:
signal DIA_TMP : std_logic_vector(0 downto 0);       -- to match RAMB4 input type
signal BUS16_GND : std_logic_vector(15 downto 0);
signal GND : std_logic;

begin
GND <= '0';
BUS16_GND <= (others =>'0');

DIA_TMP(0) <= DIA;

-- Select BlockRAM RAMB4_S1_S16 instantiation
RAMB4 : RAMB4_S1_S16
        port map (
        DIA => DIA_TMP,
        DIB => BUS16_GND ,
        ENA => ENA,
        ENB => ENB,
        WEA => WEA,
        WEB => GND,
        RSTA => GND,
        RSTB => RSTB,
        CLKA => CLK,
        CLKB => CLK,
        ADDRA => ADDRA,
        ADDRB => ADDRB,
--      DOA =>,
            DOB => DOB );
end INIT_RAMB4_S1_S16_arch;

-- Init_8_RAM16x1s module

library IEEE;
use IEEE.std_logic_1164.all;

entity INIT_8_RAM16x1s is
    port (
            DATA_IN         : in std_logic_vector(7 downto 0) ;
            ADDR            : in std_logic_vector(3 downto 0) ; -- Used by erase/write operation only
            WRITE_RAM       : in std_logic;                  -- if '1' DATA_IN is WRITE in the RAM16x1s
            CLK             : in std_logic;
            DATA_WRITE      : out std_logic_vector(7 downto 0)
            );
end INIT_8_RAM16x1s;

architecture INIT_8_RAM16x1s_arch of INIT_8_RAM16x1s is
component RAM16x1s_1
-- pragma synthesis_off
generic(
     INIT : bit_vector(15 downto 0) := X"0000"
  );
-- pragma synthesis_on
   port (
            WE       : in std_logic;
            WCLK     : in std_logic; -- inverted Clock
            D        : in std_logic;
            A0       : in std_logic;
            A1       : in std_logic;
            A2       : in std_logic;
            A3       : in std_logic;
            O        : out std_logic
    );
 end component;

 attribute INIT: string;
 attribute INIT of RAM_ERASE_0: label is "0000";
 attribute INIT of RAM_ERASE_1: label is "0000";
 attribute INIT of RAM_ERASE_2: label is "0000";
 attribute INIT of RAM_ERASE_3: label is "0000";
 attribute INIT of RAM_ERASE_4: label is "0000";
 attribute INIT of RAM_ERASE_5: label is "0000";
 attribute INIT of RAM_ERASE_6: label is "0000";
 attribute INIT of RAM_ERASE_7: label is "0000";

 begin

 -- SelectRAM instantiation = 8 x RAM16x1s
 RAM_ERASE_0 : RAM16x1s_1
            port map (
            WE => WRITE_RAM,
            WCLK => CLK,
            D => DATA_IN(0),
            A0 => ADDR(0),
            A1 => ADDR(1),
            A2 => ADDR(2),
            A3 => ADDR(3),
            O => DATA_WRITE(0)
            );

RAM_ERASE_1 : RAM16x1s_1
        port map (
            WE => WRITE_RAM,
            WCLK => CLK,
            D => DATA_IN(1),
            A0 => ADDR(0),
            A1 => ADDR(1),
            A2 => ADDR(2),
            A3 => ADDR(3),
            O => DATA_WRITE(1)
            );

RAM_ERASE_2 : RAM16x1s_1
        port map (
            WE => WRITE_RAM,
            WCLK => CLK,
            D => DATA_IN(2),
            A0 => ADDR(0),
            A1 => ADDR(1),
            A2 => ADDR(2),
            A3 => ADDR(3),
            O => DATA_WRITE(2)
            );

RAM_ERASE_3 : RAM16x1s_1
        port map (
            WE => WRITE_RAM,
            WCLK => CLK,
            D => DATA_IN(3),
            A0 => ADDR(0),
            A1 => ADDR(1),
            A2 => ADDR(2),
            A3 => ADDR(3),
            O => DATA_WRITE(3)
            );

RAM_ERASE_4 : RAM16x1s_1
        port map (
            WE => WRITE_RAM,
            WCLK => CLK,
            D => DATA_IN(4),
            A0 => ADDR(0),
            A1 => ADDR(1),
            A2 => ADDR(2),
            A3 => ADDR(3),
            O => DATA_WRITE(4)
            );

RAM_ERASE_5 : RAM16x1s_1
        port map (
            WE => WRITE_RAM,
            WCLK => CLK,
            D => DATA_IN(5),
            A0 => ADDR(0),
            A1 => ADDR(1),
            A2 => ADDR(2),
            A3 => ADDR(3),
            O => DATA_WRITE(5)
            );

RAM_ERASE_6 : RAM16x1s_1
        port map (
            WE => WRITE_RAM,
            WCLK => CLK,
            D => DATA_IN(6),
            A0 => ADDR(0),
            A1 => ADDR(1),
            A2 => ADDR(2),
            A3 => ADDR(3),
            O => DATA_WRITE(6)
            );

RAM_ERASE_7 : RAM16x1s_1
        port map (
            WE => WRITE_RAM,
            WCLK => CLK,
            D => DATA_IN(7),
            A0 => ADDR(0),
            A1 => ADDR(1),
            A2 => ADDR(2),
            A3 => ADDR(3),
            O => DATA_WRITE(7)
            );

end INIT_8_RAM16x1s_arch;

-- 32 to 5 encoder
library IEEE;
use IEEE.std_logic_1164.all;

entity ENCODE_4_LSB is
   port (
       BINARY_ADDR      : in std_logic_vector(31 downto 0);
       MATCH_ADDR       : out std_logic_vector(4 downto 0); -- Match address found
       MATCH_OK         : out std_logic -- '1' if MATCH found
   );
end entity ENCODE_4_LSB;
architecture ENCODE_4_LSB_arch of ENCODE_4_LSB is
begin
GENERATE_ADDRESS : process (BINARY_ADDR)
begin
           case BINARY_ADDR(31 downto 0) is
                 when "00000000000000000000000000000001" => MATCH_ADDR <= "00000";
                 when "00000000000000000000000000000010" => MATCH_ADDR <= "00001";
                 when "00000000000000000000000000000100" => MATCH_ADDR <= "00010";
                 when "00000000000000000000000000001000" => MATCH_ADDR <= "00011";
```

```
                        when "00000000000000000000000000010000" => MATCH_ADDR <= "00100";
                        when "00000000000000000000000000100000" => MATCH_ADDR <= "00101";
                        when "00000000000000000000000001000000" => MATCH_ADDR <= "00110";
                        when "00000000000000000000000010000000" => MATCH_ADDR <= "00111";
                        when "00000000000000000000000100000000" => MATCH_ADDR <= "01000";
                        when "00000000000000000000001000000000" => MATCH_ADDR <= "01001";
                        when "00000000000000000000010000000000" => MATCH_ADDR <= "01010";
                        when "00000000000000000000100000000000" => MATCH_ADDR <= "01011";
                        when "00000000000000000001000000000000" => MATCH_ADDR <= "01100";
                        when "00000000000000000010000000000000" => MATCH_ADDR <= "01101";
                        when "00000000000000000100000000000000" => MATCH_ADDR <= "01110";
                        when "00000000000000001000000000000000" => MATCH_ADDR <= "01111";
                        when "00000000000000010000000000000000" => MATCH_ADDR <= "10000";
                        when "00000000000000100000000000000000" => MATCH_ADDR <= "10001";
                        when "00000000000001000000000000000000" => MATCH_ADDR <= "10010";
                        when "00000000000010000000000000000000" => MATCH_ADDR <= "10011";
                        when "00000000000100000000000000000000" => MATCH_ADDR <= "10100";
                        when "00000000001000000000000000000000" => MATCH_ADDR <= "10101";
                        when "00000000010000000000000000000000" => MATCH_ADDR <= "10110";
                        when "00000000100000000000000000000000" => MATCH_ADDR <= "10111";
                        when "00000001000000000000000000000000" => MATCH_ADDR <= "11000";
                        when "00000010000000000000000000000000" => MATCH_ADDR <= "11001";
                        when "00000100000000000000000000000000" => MATCH_ADDR <= "11010";
                        when "00001000000000000000000000000000" => MATCH_ADDR <= "11011";
                        when "00010000000000000000000000000000" => MATCH_ADDR <= "11100";
                        when "00100000000000000000000000000000" => MATCH_ADDR <= "11101";
                        when "01000000000000000000000000000000" => MATCH_ADDR <= "11110";
                        when "10000000000000000000000000000000" => MATCH_ADDR <= "11111";
                        when others =>
                                MATCH_ADDR <= ( others => 'X');


        end case;
end process GENERATE_ADDRESS;

-- Generate the match signal if one or more matche(s) is/are found
GENERATE_MATCH : process (BINARY_ADDR)
begin
if (BINARY_ADDR = "00000000000000000000000000000000") then
        MATCH_OK <= '0';
else
        MATCH_OK <= '1';
end if;
end process GENERATE_MATCH;

end architecture ENCODE_4_LSB_arch;

--First Stage Latch of Pipeline ESS


library IEEE;
use IEEE.std_logic_1164.all;

entity FSL is
port(clk: in std_logic;
    putin, getin, matchin: in std_logic;
    match_inaddr: in std_logic_vector(4 downto 0);
    putout, getout, matchout: out std_logic;
    match_outaddr: out std_logic_vector(4 downto 0));
end entity FSL;

architecture FSL_beh of FSL is
signal psig, gsig: std_logic;
begin

FSL_Process1: process(clk, putin, getin) is
begin
if (rising_edge(clk)) then
psig <= putin;
gsig <= getin;
end if;
end process FSL_Process1;

FSL_Process2: process(clk, psig, gsig, matchin, match_inaddr) is
begin
if (falling_edge(clk)) then
putout <= psig;
getout <= gsig;
matchout <= matchin;
match_outaddr <= match_inaddr;
end if;
end process FSL_Process2;
end architecture FSL_beh;

--Second Stage Pipeline ESS


library IEEE;
use IEEE.std_logic_1164.all;

entity Second_Stage is
port(clk, clock, get, put, empin_fmTS, lexpd_fmTS, put_fmTS, matchin_fmTS: in std_logic;
    matchin_fmFS: in std_logic;
    mataddr_fmFS: in std_logic_vector(4 downto 0);
    matchout_sec, putout_sec, getout_sec: out std_logic;
    emptysig_out_sec, life_expd_out_sec, GF_out, PF_out: out std_logic;
    mux_outaddr_sec: out std_logic_vector(4 downto 0));
end entity Second_Stage;

architecture Second_Stage_beh of Second_Stage is
--Components
component SSPE is
port(clk, clock, get, put, empin_fmTS, lexpd_fmTS, put_fmTS, matchin_fmTS: in std_logic;
    matchin_fmFS: in std_logic;
    mataddr_fmFS: in std_logic_vector(4 downto 0);
    mux_addrout: out std_logic_vector(4 downto 0);
    empty_out, life_expd_out, GF, PF: out std_logic);
end component SSPE;

component SSL is
port(clk: in std_logic;
    matchin_sec, putin_sec, getin_sec: in std_logic;
    emptysig_in, life_expd_in, GF_in, PF_in: in std_logic;
    mux_inaddr_sec: in std_logic_vector(4 downto 0);
    matchout_sec, putout_sec, getout_sec: out std_logic;
```

```
   emptysig_out_sec, life_expd_out_sec, GF_out, PF_out: out std_logic;
   mux_outaddr_sec: out std_logic_vector(4 downto 0));
end component SSL;

--signals
signal muxoutsig: std_logic_vector(4 downto 0);
signal emptyoutsig, lifeexpdsig, GF_sig, PF_sig: std_logic;
begin

SSPE_comp: SSPE port map(clk=>clk, clock=>clock, get=>get, put=>put, empin_fmTS=>empin_fmTS,
lexpd_fmTS=>lexpd_fmTS, put_fmTS=>put_fmTS, matchin_fmTS=>matchin_fmTS,
matchin_fmFS=>matchin_fmFS, mataddr_fmFS=>mataddr_fmFS, mux_addrout=>muxoutsig,
empty_out=>emptyoutsig, life_expd_out=>lifeexpdsig, GF=>GF_sig, PF=>PF_sig);

SSL_comp: SSL port map(clk=>clk, matchin_sec=>matchin_fmFS, putin_sec=>put, getin_sec=>get,
emptysig_in=>emptyoutsig, life_expd_in=>lifeexpdsig, GF_in=>GF_sig, PF_in=>PF_sig,
mux_inaddr_sec=>muxoutsig, matchout_sec=>matchout_sec, putout_sec=>putout_sec,
getout_sec=>getout_sec, emptysig_out_sec=>emptysig_out_sec, life_expd_out_sec=>life_expd_out_sec,
GF_out=>GF_out, PF_out=>PF_out, mux_outaddr_sec=>mux_outaddr_sec);

end architecture Second_Stage_beh;

--Individual Components
-- Second Stage of Pipeline ESS

library IEEE;
use IEEE.std_logic_1164.all;

entity SSPE is
port(clk, clock, get, put, empin_fmTS, lexpd_fmTS, put_fmTS, matchin_fmTS: in std_logic;
   matchin_fmFS: in std_logic;
   mataddr_fmFS: in std_logic_vector(4 downto 0);
   mux_addrout: out std_logic_vector(4 downto 0);
   empty_out, life_expd_out, GF, PF: out std_logic);
end entity SSPE;

architecture SSPE_beh of SSPE is
--components
--empty ram
component empram0 is
port(addr: in std_logic_vector(4 downto 0);
    data_in_emp0: in std_logic;
    data_out_emp0: out std_logic;
    emp_loc_addr: out std_logic_vector(4 downto 0);
    empout: out std_logic_vector(31 downto 0);
    clk: in std_logic;
    we_emp0: in std_logic);
end component empram0;
--check empty
component empcount is
port(emptysig: in std_logic_vector(31 downto 0);
    chk_empty: out std_logic);
end component empcount;
--mux address
component mux1 is
port (a1: in STD_LOGIC_VECTOR (4 downto 0);
    b1: in STD_LOGIC_VECTOR (4 downto 0);
    s1: in STD_LOGIC;
    y1: out STD_LOGIC_VECTOR (4 downto 0) );
end component mux1;
--exp. time ram
component exptime_ram is
port(clk, we, en, rst: in std_logic;
    addr: in std_logic_vector(4 downto 0);
    din: in std_logic_vector(7 downto 0);
    dout: out std_logic_vector(7 downto 0));
end component exptime_ram;
--exp. time calc
component exp_calc is
port(exptime_in: in std_logic_vector(7 downto 0);  -- originally it has to be 10 bits bot now for checking 8
bits
    clock, chklife: in std_logic;
    life_expd: out std_logic;
    exptime_out: out std_logic_vector(7 downto 0));
end component exp_calc;

--signals
signal int_mux_addr, empty_addr: std_logic_vector(4 downto 0);
signal empout_full: std_logic_vector(31 downto 0);
signal expdatain, expdataout: std_logic_vector(7 downto 0);
signal empsig, data_outsig, we_exp_sig, we_emp_sig, life_expd_sig, ensig, rstsig, chksig: std_logic;
begin

mux_addrout <= int_mux_addr;
empty_out <= empsig;
life_expd_out <= life_expd_sig;
GF <= ( ((not(matchin_fmFS)) and get) or (matchin_fmFS and life_expd_sig and get) );
PF <= ( (not(matchin_fmFS)) and (not(empsig)) and put);
we_exp_sig <= ( (put_fmTS and (not(matchin_fmTS)) and empin_fmTS) or (put_fmTS and matchin_fmTS
and lexpd_fmTS) );
we_emp_sig <= ( (put_fmTS and matchin_fmTS) or (put_fmTS and (not(matchin_fmTS)) and
empin_fmTS) );
ensig <= '1';
rstsig <= '0';
chksig <= matchin_fmFS;

empram_comp: empram0 port map(addr=>int_mux_addr, data_in_emp0=>empin_fmTS,
data_out_emp0=>data_outsig, emp_loc_addr=>empty_addr, empout=>empout_full, clk=>clk,
we_emp0=>we_emp_sig);

empcnt_comp: empcount port map(emptysig=>empout_full, chk_empty=>empsig);

addrmux_comp: mux1 port map(a1=>empty_addr, b1=>mataddr_fmFS, s1=>matchin_fmFS,
y1=>int_mux_addr);

expram_comp: exptime_ram port map(clk=>clk, we=>we_exp_sig, en=>ensig, rst=>rstsig,
addr=>int_mux_addr, din=>expdatain, dout=>expdataout);

expcalc_comp: exp_calc port map(exptime_in=>expdataout, clock=>clock, chklife=>chksig,
life_expd=>life_expd_sig, exptime_out=>expdatain);

end architecture SSPE_beh;
```

```
--Individual components
-- Exp Mem Design using Block RAM

library IEEE;
use IEEE.std_logic_1164.all;

entity exptime_ram is
port(clk, we, en, rst: in std_logic;
     addr: in std_logic_vector(4 downto 0);
     din: in std_logic_vector(7 downto 0);
     dout: out std_logic_vector(7 downto 0));
end entity exptime_ram;
architecture behaviour of exptime_ram is

component RAMB4_S8 is
port(ADDR: in std_logic_vector(8 downto 0);
     CLK: in std_logic;
     DI: in std_logic_vector(7 downto 0);
     DO: out std_logic_vector(7 downto 0);
     EN, RST, WE: in std_logic);
end component RAMB4_S8;

signal msbaddr: std_logic_vector(3 downto 0);
signal addr_expram: std_logic_vector(8 downto 0);
begin
msbaddr <= "0000";
addr_expram <= msbaddr & addr;

ram0: RAMB4_S8 port map(ADDR=>addr_expram, CLK=>clk, DI=>din, DO=>dout, EN=>en,
RST=>rst, WE=>we);

end architecture behaviour;

-- EXPIRATION TIME CALCULATION MODULE
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity exp_calc is
port(exptime_in: in std_logic_vector(7 downto 0);  -- originally it has to be 10 bits bot now for checking 8
bits
     clock, chklife: in std_logic;
     life_expd: out std_logic;
     exptime_out: out std_logic_vector(7 downto 0));
end entity exp_calc;

architecture expcalc_beh of exp_calc is
signal gcrsig: std_logic_vector(7 downto 0);
begin

expcalcprocess:process(gcrsig, exptime_in, chklife) is
begin
if(chklife = '1') then
if(gcrsig <= exptime_in) then
```

```
life_expd <= '0'; -- life time is not expired
else
life_expd <= '1'; -- life time expired
end if;
else
life_expd <= '0'; -- default
end if;
end process expcalcprocess;

gcrprocess:process(clock, gcrsig) is
variable tau: std_logic_vector(7 downto 0);
begin
tau := "00001111";
exptime_out <= gcrsig + tau; -- new life time
if (rising_edge(clock)) then
gcrsig <= gcrsig + 1;
end if;
end process gcrprocess;
end architecture expcalc_beh;

-- EMPTY RAM Module

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity empram0 is
port(addr: in std_logic_vector(4 downto 0);
    data_in_emp0: in std_logic;
    data_out_emp0: out std_logic;
    emp_loc_addr: out std_logic_vector(4 downto 0);
    empout: out std_logic_vector(31 downto 0);
    clk: in std_logic;
    we_emp0: in std_logic);
end entity empram0;

architecture behavioural of empram0 is
-- function for getting integer
function getint(signal data: std_logic_vector) return integer is
variable count: integer range 0 to 32;
begin
for i in data'range loop
if(data(i) = '0') then
count := i;
end if;
end loop;
return (count);
end function getint;

type mem_array is array(0 to 31) of std_logic;
signal emptyout: std_logic_vector(31 downto 0);
signal empty_mem :mem_array;
signal address: integer;
signal emploc: integer range 0 to 32;
begin
```

```
address <= conv_integer(addr);
emploc <= getint(emptyout);
emp_loc_addr <= conv_std_logic_vector(emploc, 5);

mem_process:process(clk, addr, we_emp0, data_in_emp0, empty_mem, emptyout) is
begin
if (rising_edge(clk)) then
 if (we_emp0 = '1') then
   empty_mem(address) <= data_in_emp0;
 else
   data_out_emp0 <= empty_mem(address);
 end if;
end if;

for i in 31 downto 0 loop
emptyout(i) <= empty_mem(i);
end loop;
empout <= emptyout;
end process mem_process;
end architecture behavioural;

-- Count the number for zeros for empty location

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity empcount is
port(emptysig: in std_logic_vector(31 downto 0);
     chk_empty: out std_logic);
end entity empcount;

architecture empcnt_beh of empcount is
signal c: std_logic;
begin
process(emptysig, c) is
begin

c <= ((emptysig(31) and emptysig(30) and emptysig(29) and emptysig(28)) and (emptysig(27) and
emptysig(26) and emptysig(25) and emptysig(24)) and (emptysig(23) and emptysig(22) and emptysig(21)
and emptysig(20)) and (emptysig(19) and emptysig(18) and emptysig(17) and emptysig(16)) and
(emptysig(15) and emptysig(14) and emptysig(13) and emptysig(12)) and (emptysig(11) and emptysig(10)
and emptysig(9) and emptysig(8)) and (emptysig(7) and emptysig(6) and emptysig(5) and emptysig(4)) and
(emptysig(3) and emptysig(2) and emptysig(1) and emptysig(0)));
chk_empty <= not (c);

end process;
end architecture empcnt_beh;

--MUX for address
library IEEE;
use IEEE.std_logic_1164.all;

entity mux1 is
port (a1: in STD_LOGIC_VECTOR (4 downto 0);
    b1: in STD_LOGIC_VECTOR (4 downto 0);
    s1: in STD_LOGIC;
    y1: out STD_LOGIC_VECTOR (4 downto 0) );
end entity mux1;

architecture mux_arch1 of mux1 is
begin

process (a1, b1, s1)
begin
case s1 is
when '0' => y1 <= a1;
when '1' => y1 <= b1;
when others => y1 <= (others => '0');
end case;
end process;
end mux_arch1;

--Second Stage Latch of Pipeline ESS

library IEEE;
use IEEE.std_logic_1164.all;

entity SSL is
port(clk: in std_logic;
    matchin_sec, putin_sec, getin_sec: in std_logic;
    emptysig_in, life_expd_in, GF_in, PF_in: in std_logic;
    mux_inaddr_sec: in std_logic_vector(4 downto 0);
    matchout_sec, putout_sec, getout_sec: out std_logic;
    emptysig_out_sec, life_expd_out_sec, GF_out, PF_out: out std_logic;
    mux_outaddr_sec: out std_logic_vector(4 downto 0));
end entity SSL;

architecture SSL_beh of SSL is
signal m2sig, p2sig, g2sig, e2sig, l2sig, GF2sig, PF2sig: std_logic;
signal maddr2_sig: std_logic_vector(4 downto 0);
begin

SSL_Process1: process(clk, matchin_sec, putin_sec, getin_sec, emptysig_in, life_expd_in, GF_in, PF_in,
mux_inaddr_sec) is
begin
if (rising_edge(clk)) then
m2sig <= matchin_sec;
p2sig <= putin_sec;
g2sig <= getin_sec;
e2sig <= emptysig_in;
--l2sig <= life_expd_in;
GF2sig <= GF_in;
PF2sig <= PF_in;
maddr2_sig <= mux_inaddr_sec;
end if;
end process SSL_Process1;

SSL_Process2: process(clk, m2sig, p2sig, g2sig, e2sig, life_expd_in, GF2sig, PF2sig, maddr2_sig) is
begin
if (falling_edge(clk)) then
```

```
matchout_sec <= m2sig;
putout_sec <= p2sig;
getout_sec <= g2sig;
emptysig_out_sec <= e2sig;
life_expd_out_sec <= life_expd_in;
GF_out <= GF2sig;
PF_out <= PF2sig;
mux_outaddr_sec <= maddr2_sig;
end if;
end process SSL_Process2;
end architecture SSL_beh;

--Third Stage of Pipeline ESS

library IEEE;
use IEEE.std_logic_1164.all;

entity TSPE is
port(clk, get, put: in std_logic;
    GF_fmsec, PF_fmsec, empty_fmsec, lifeexpd_fmsec, match_fmsec: in std_logic;
    value_in: in std_logic_vector(63 downto 0);
    muxaddr_fmsec: in std_logic_vector(4 downto 0);
    GFOUT, PFOUT, ESSFULL, le: out std_logic;
    OUTVALUE: out std_logic_vector(63 downto 0));
end entity TSPE;

architecture TSPE_beh of TSPE is
--components
component ram_val is
port(clk, we_val, en, rst: in std_logic;
    addr: in std_logic_vector(4 downto 0);
    data_in_val: in std_logic_vector(63 downto 0);
    data_out_val: out std_logic_vector(63 downto 0));
end component ram_val;

component mux is
port(a: in STD_LOGIC_VECTOR (63 downto 0);
    b: in STD_LOGIC_VECTOR (63 downto 0);
    s: in STD_LOGIC;
    y: out STD_LOGIC_VECTOR (63 downto 0) );
end component mux;

--signals
signal zerosig, muxvalout, OUTsig: std_logic_vector(63 downto 0);
signal rst_zero, en_one, wesig, ggfsig: std_logic;
begin

zerosig <= (others => '0');
rst_zero <= '0';
en_one <= '1';
wesig <= ( (get and match_fmsec and lifeexpd_fmsec) or (get and (not(match_fmsec))) or (put and
match_fmsec) or (put and (not(match_fmsec)) and empty_fmsec) );

--outputs
GFOUT <= GF_fmsec;
PFOUT <= PF_fmsec;
```

```
le <= lifeexpd_fmsec;
ESSFULL <= (not(empty_fmsec));
ggfsig <= GF_fmsec or (not(get));

muxval_comp: mux port map(a=>value_in, b=>zerosig, s=>GF_fmsec, y=>muxvalout);
muxout_comp: mux port map(a=>OUTsig, b=>zerosig, s=>ggfsig, y=>OUTVALUE);
valram_comp: ram_val port map(clk=>clk, we_val=>wesig, en=>en_one, rst=>rst_zero,
addr=>muxaddr_fmsec, data_in_val=>muxvalout, data_out_val=>OUTsig);

end architecture TSPE_beh;

--Individual Components
-- For VALUE RAM

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ram_val is
port(clk, we_val, en, rst: in std_logic;
    addr: in std_logic_vector(4 downto 0);
    data_in_val: in std_logic_vector(63 downto 0);
    data_out_val: out std_logic_vector(63 downto 0));
end entity ram_val;

architecture ramval_behave of ram_val is
component valram is
port(clk, we, en, rst: in std_logic;
    addr: in std_logic_vector(4 downto 0);
    din: in std_logic_vector(15 downto 0);
    dout: out std_logic_vector(15 downto 0));
end component valram;
begin

valueram0: valram port map(clk=>clk, we=>we_val, en=>en, rst=>rst, addr=>addr, din=>data_in_val(15
downto 0), dout=>data_out_val(15 downto 0));
valueram1: valram port map(clk=>clk, we=>we_val, en=>en, rst=>rst, addr=>addr, din=>data_in_val(31
downto 16), dout=>data_out_val(31 downto 16));
valueram2: valram port map(clk=>clk, we=>we_val, en=>en, rst=>rst, addr=>addr, din=>data_in_val(47
downto 32), dout=>data_out_val(47 downto 32));
valueram3: valram port map(clk=>clk, we=>we_val, en=>en, rst=>rst, addr=>addr, din=>data_in_val(63
downto 48), dout=>data_out_val(63 downto 48));

end architecture ramval_behave;

-- Value Mem Design using Block RAM
library IEEE;
use IEEE.std_logic_1164.all;

entity valram is
port(clk, we, en, rst: in std_logic;
    addr: in std_logic_vector(4 downto 0);
    din: in std_logic_vector(15 downto 0);
    dout: out std_logic_vector(15 downto 0));
end entity valram;
```

```
architecture behave_valram of valram is

component RAMB4_S16 is
port(ADDR: in std_logic_vector(7 downto 0);
    CLK: in std_logic;
    DI: in std_logic_vector(15 downto 0);
    DO: out std_logic_vector(15 downto 0);
    EN, RST, WE: in std_logic);
end component RAMB4_S16;

signal msbvaladdr: std_logic_vector(2 downto 0);
signal addr_valram: std_logic_vector(7 downto 0);
begin

msbvaladdr <= "000";
addr_valram <= msbvaladdr & addr;

ram0: RAMB4_S16 port map(ADDR=>addr_valram, CLK=>clk, DI=>din, DO=>dout, EN=>en,
RST=>rst, WE=>we);

end architecture behave_valram;

--MUX for VALUE

library IEEE;
use IEEE.std_logic_1164.all;

entity mux is
port(a: in STD_LOGIC_VECTOR (63 downto 0);
    b: in STD_LOGIC_VECTOR (63 downto 0);
    s: in STD_LOGIC;
    y: out STD_LOGIC_VECTOR (63 downto 0) );
end entity mux;

architecture mux_arch of mux is
begin

process (a, b, s)
begin
if ( s = '0') then
y <= a;
else y <= b;
end if;
end process;
end architecture mux_arch;

-- ETM/LTC stage Regsiter
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ex3_ex4_reg is
port(clk, EX_Flush_in: in std_logic;
    braddrin: in std_logic_vector(15 downto 0);
    ctrlinEX: in std_logic_vector(24 downto 0);
    opinEX: in std_logic_vector(5 downto 0);
    WB_in_fm_ex: in std_logic_vector(3 downto 0);
    RS1_in_fm_ex, RS2_in_fm_ex, RD_in_fm_ex, TR_in_fm_ex, VR_in_fm_ex: in std_logic_vector(4
downto 0);
    aluout_fm_ex, pktout_fm_ex, GPR1in, GPr2in: in std_logic_vector(63 downto 0);
    braddrout: out std_logic_vector(15 downto 0);
    ctrloutEX: out std_logic_vector(24 downto 0);
    opoutEX: out std_logic_vector(5 downto 0);
    aluout_to_wb, pktout_to_wb, GPR1out, GPR2out: out std_logic_vector(63 downto 0);
    RS1_out_to_regs, RS2_out_to_regs, RD_out_to_regs, TR_out_to_regs, VR_out_to_regs: out
std_logic_vector(4 downto 0);
    WB_out_fm_wb: out std_logic_vector(3 downto 0));
end entity ex3_ex4_reg;

architecture ex34_beh of ex3_ex4_reg is
signal chkoutalu: std_logic_vector(63 downto 0);
begin

ex3process:process(clk, chkoutalu, braddrin, ctrlinEX, opinEX, WB_in_fm_ex, RD_in_fm_ex,
TR_in_fm_ex, VR_in_fm_ex, aluout_fm_ex, pktout_fm_ex, GPR1in, GPR2in, EX_Flush_in) is
begin

if(falling_edge(clk)) then
case EX_Flush_in is
when '0' =>
WB_out_fm_wb <= WB_in_fm_ex;
RD_out_to_regs <= RD_in_fm_ex;
TR_out_to_regs <= TR_in_fm_ex;
VR_out_to_regs <= VR_in_fm_ex;
aluout_to_wb <= chkoutalu;
pktout_to_wb <= pktout_fm_ex;
ctrloutEX <= ctrlinEX;
opoutEX <= opinEX;
RS1_out_to_regs <= RS1_in_fm_ex;
RS2_out_to_regs <= RS2_in_fm_ex;
GPR1out <= GPR1in;
GPR2out <= GPR2in;
braddrout <= braddrin;
when '1' =>
WB_out_fm_wb <= (others => '0');
RD_out_to_regs <= (others => '0');
TR_out_to_regs <= (others => '0');
VR_out_to_regs <= (others => '0');
aluout_to_wb <= (others => '0');
pktout_to_wb <= (others => '0');
ctrloutEX <= (others => '0');
opoutEX <= (others => '0');
RS1_out_to_regs <= (others => '0');
RS2_out_to_regs <= (others => '0');
GPR1out <= (others => '0');
GPR2out <= (others => '0');
when others => null;
end case;
end if;
end process ex3process;
```

```
chkprocess: process(chkoutalu, opinEX, aluout_fm_ex, pktout_fm_ex) is
begin
case opinEX is
when "010101" =>
chkoutalu <= pktout_fm_ex;
when others =>
chkoutalu <= aluout_fm_ex;
end case;
end process chkprocess;
end architecture ex34_beh;
```

5. LTC Stage

```
-- LTC stage Top

library IEEE;
use IEEE.std_logic_1164.all;

entity ex4top is
port(clk: in std_logic;
    WBctrlin: in std_logic_vector(3 downto 0);
    out_fm_alu: in std_logic_vector(63 downto 0);
    RS1in, RS2in, VRin, VSTRD, VSTVRD: in std_logic_vector(4 downto 0);
    RDin_fm4, VRDin_fm4, TRDin_fm4: in std_logic_vector(4 downto 0);
    op_in: in std_logic_vector(5 downto 0);
    GPRin1, GPRin2, PTin: in std_logic_vector(63 downto 0);
    brtype: in std_logic_vector(2 downto 0);
    ccr_inp, ccr_ing: in std_logic;
    branch: out std_logic;
    WBctout: out std_logic_vector(3 downto 0);
    WBdataout: out std_logic_vector(63 downto 0);
    WBRDout, WBVRDout, WBTRDout: out std_logic_vector(4 downto 0));
end entity ex4top;

architecture ex4top_beh of ex4top is
--components
component ex4stage is
port(op_in: in std_logic_vector(5 downto 0);
    RS1in, RS2in, VRin, VSTRD, VSTVRD: in std_logic_vector(4 downto 0);
    GPRin1, GPRin2, PTin: in std_logic_vector(63 downto 0);
    brtype: in std_logic_vector(2 downto 0);
    ccr_inp, ccr_ing: in std_logic;
    branch: out std_logic);
end component ex4stage;

component ex4_ex5_reg is
port(clk: in std_logic;
    WBctrlin: in std_logic_vector(3 downto 0);
    out_fm_alu: in std_logic_vector(63 downto 0);
    RDin_fm4, VRDin_fm4, TRDin_fm4: in std_logic_vector(4 downto 0);
    WBctout: out std_logic_vector(3 downto 0);
    WBdataout: out std_logic_vector(63 downto 0);
    WBRDout, WBVRDout, WBTRDout: out std_logic_vector(4 downto 0));
end component ex4_ex5_reg;

begin
```

```
ex4stcomp: ex4stage port map(op_in=>op_in, RS1in=>RS1in, RS2in=>RS2in, VRin=>VRin,
VSTRD=>VSTRD, VSTVRD=>VSTVRD, GPRin1=>GPRin1, GPRin2=>GPRin2, PTin=>PTin,
brtype=>brtype, ccr_inp=>ccr_inp, ccr_ing=>ccr_ing, branch=>branch);

ex4regcomp: ex4_ex5_reg port map(clk=>clk, WBctrlin=>WBctrlin, out_fm_alu=>out_fm_alu,
RDin_fm4=>RDin_fm4, VRDin_fm4=>VRDin_fm4, TRDin_fm4=>TRDin_fm4, WBctout=>WBctout,
WBdataout=>WBdataout, WBRDout=>WBRDout, WBVRDout=>WBVRDout,
WBTRDout=>WBTRDout);

end architecture ex4top_beh;

--Individual componenets
-- LTC Module

library IEEE;
use IEEE.std_logic_1164.all;

entity ex4stage is
port(op_in: in std_logic_vector(5 downto 0);
    RS1in, RS2in, VRin, VSTRD, VSTVRD: in std_logic_vector(4 downto 0);
    GPRin1, GPRin2, PTin: in std_logic_vector(63 downto 0);
    brtype: in std_logic_vector(2 downto 0);
    ccr_inp, ccr_ing: in std_logic;
    branch: out std_logic);
end entity ex4stage;

architecture ex4_beh of ex4stage is
--componenets
component bdetunit is
port(brtype: in std_logic_vector(2 downto 0);
    op_in: in std_logic_vector(5 downto 0);
    RS1, RS2: in std_logic_vector(63 downto 0);
    ccr_inp, ccr_ing: in std_logic;
    branch: out std_logic);
end component bdetunit;

component muxbr is
port(GPRin, PTin: in std_logic_vector(63 downto 0);
    Sbr: in std_logic;
    Brin: out std_logic_vector(63 downto 0));
end component muxbr;

component fwd_br is
port(opcode_in: in std_logic_vector(5 downto 0);
    RS1in, RS2in, VRin, VSTRD, VSTVRD: in std_logic_vector(4 downto 0);
    Sbr1_out, Sbr2_out: out std_logic);
end component fwd_br;

--signals
signal brRS1in, brRS2in: std_logic_vector(63 downto 0);
signal Sbr1sig, Sbr2sig: std_logic;
begin

bcomp: bdetunit port map(brtype=>brtype, op_in=>op_in, RS1=>brRS1in, RS2=>brRS2in,
ccr_inp=>ccr_inp, ccr_ing=>ccr_ing, branch=>branch);
```

```
mbcomp1: muxbr port map(GPRin=>GPRin1, PTin=>PTin, Sbr=>Sbr1sig, Brin=>brRS1in);
mbcomp2: muxbr port map(GPRin=>GPRin2, PTin=>PTin, Sbr=>Sbr2sig, Brin=>brRS2in);

fcomp: fwd_br port map(opcode_in=>op_in, RS1in=>RS1in, RS2in=>RS2in, VRin=>VRin,
VSTRD=>VSTRD, VSTVRD=>VSTVRD, Sbr1_out=>Sbr1sig, Sbr2_out=>Sbr2sig);

end architecture ex4_beh;

--Individual Componenets
-- Branch Detect Unit

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity bdetunit is
port(brtype: in std_logic_vector(2 downto 0);
     op_in: in std_logic_vector(5 downto 0);
     RS1, RS2: in std_logic_vector(63 downto 0);
     ccr_inp, ccr_ing: in std_logic;
--     ccr_inp, ccr_ing, overflow, jumpin, retin: in std_logic;
--     jumpout, retout: out std_logic;
--     branch, ID_Flush_br: out std_logic);
     branch: out std_logic);
end entity bdetunit;

architecture bdetunit_beh of bdetunit is
-- comparator function
function compare(signal a, b: std_logic_vector) return std_logic is
variable equal: std_logic;
variable res_or: std_logic;
variable res_xor: std_logic_vector(63 downto 0);
begin

res_or := '0';
res_xor := a xor b;

for i in 63 downto 0 loop
res_or := res_or or res_xor(i);
end loop;
equal := not (res_or);
return equal;
end function compare;

signal temp, br_sig: std_logic;
signal zerosig: std_logic_vector(63 downto 0);

begin
brprocess:process(brtype, RS1, RS2, temp, zerosig, br_sig, ccr_inp, ccr_ing, op_in) is
begin
zerosig <= (others => '0');
case brtype is
when "001" => --BRNE
temp <= compare(RS1, RS2);
br_sig <= not(temp);
```

```
when "010" => --BREQ
temp <= compare(RS1, RS2);
br_sig <= temp;

when "011" => --BGE
if(RS1 >= RS2) then
temp <= '1';
else
temp <= '0';
end if;
br_sig <= temp;

when "100" => -- BNEZ
temp <= compare(RS1, zerosig);
br_sig <= not(temp);

when "101" => --BEQZ
temp <= compare(RS1, zerosig);
br_sig <= temp;

when "110" => -- BGF
temp <= ccr_ing;
br_sig <= temp;

when "111" => -- BPF
temp <= ccr_inp;
br_sig <= temp;

when "000" => -- BLT
if(op_in = "100011") then
if(RS1 < RS2) then
temp <= '1';
else
temp <= '0';
end if;
br_sig <= temp;
else
temp <= '0';
br_sig <= temp;
end if;
when others =>
br_sig <= '0';
temp <= '0';
end case;

branch <= br_sig;
end process brprocess;

end architecture bdetunit_beh;

-- MUX used as BR. Det unit in mux

library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity muxbr is
port(GPRin, PTin: in std_logic_vector(63 downto 0);
    Sbr: in std_logic;
    Brin: out std_logic_vector(63 downto 0));
end entity muxbr;

architecture muxbr_beh of muxbr is
signal brsig: std_logic_vector(63 downto 0);
begin

process(GPRin, PTin, Sbr, brsig) is
begin
case Sbr is
when '0' => brsig <= GPRin;
when '1' => brsig <= PTin;
when others => brsig <= brsig;
end case;
Brin <= brsig;
end process;
end architecture muxbr_beh;

-- Simple FWD unit for Br.Det

library IEEE;
use IEEE.std_logic_1164.all;

entity fwd_br is
port(opcode_in: in std_logic_vector(5 downto 0);
    RS1in, RS2in, VRin, VSTRD, VSTVRD: in std_logic_vector(4 downto 0);
    Sbr1_out, Sbr2_out: out std_logic);
end entity fwd_br;

architecture fwd_br_beh of fwd_br is
begin

b1p:process(opcode_in, RS1in, VRin, VSTRD, VSTVRD) is
begin
if(opcode_in = "010111" or opcode_in = "011000" or opcode_in = "011001" or opcode_in = "011010" or
opcode_in = "011011" or opcode_in = "100011") then
if( (RS1in /= "00000" and RS1in = VSTRD) or (VRin /= "00000" and VRin = VSTVRD) ) then
Sbr1_out <= '1';
else
Sbr1_out <= '0';
end if;
else
Sbr1_out <= '0';
end if;
end process b1p;

b2p:process(opcode_in, RS2in, VRin, VSTRD, VSTVRD) is
begin
if(opcode_in = "010111" or opcode_in = "011000" or opcode_in = "011001" or opcode_in = "011010" or
opcode_in = "011011" or opcode_in = "100011") then
if( (RS2in /= "00000" and RS2in = VSTRD) or (VRin /= "00000" and VRin = VSTVRD) ) then
Sbr2_out <= '1';
else
```

```
Sbr2_out <= '0';
end if;
else
Sbr2_out <= '0';
end if;
end process b2p;
end architecture fwd_br_beh;

-- LTC/UD stage reg

library IEEE;
use IEEE.std_logic_1164.all;

entity ex4_ex5_reg is
port(clk: in std_logic;
    WBctrlin: in std_logic_vector(3 downto 0);
    out_fm_alu: in std_logic_vector(63 downto 0);
    RDin_fm4, VRDin_fm4, TRDin_fm4: in std_logic_vector(4 downto 0);
    WBctout: out std_logic_vector(3 downto 0);
    WBdataout: out std_logic_vector(63 downto 0);
    WBRDout, WBVRDout, WBTRDout: out std_logic_vector(4 downto 0));
end entity ex4_ex5_reg;

architecture ex45_beh of ex4_ex5_reg is
begin

process(clk, WBctrlin, out_fm_alu, RDin_fm4, VRDin_fm4, TRDin_fm4) is
begin
if(falling_edge(clk)) then
WBctout <= WBctrlin;
WBdataout <= out_fm_alu;
WBRDout <= RDin_fm4;
WBVRDout <= VRDin_fm4;
WBTRDout <= TRDin_fm4;
end if;
end process;
end architecture ex45_beh;
```

6. UD STAGE

```
-- UD Stage Top

library IEEE;
use IEEE.std_logic_1164.all;

entity stage5 is
port(WB_in1: in std_logic;
    aluout_fm_ex, essout_fm_st5: in std_logic_vector(63 downto 0);
    dataout: out std_logic_vector(63 downto 0));
end entity stage5;
architecture stage5_beh of stage5 is

component wbstage is
port(WB_in1_fm_exreg: in std_logic;
    aluout_fm_exreg, essout_fm_exreg: in std_logic_vector(63 downto 0);
    dataoutfmwb: out std_logic_vector(63 downto 0));
end component wbstage;
```

```
begin
st5comp: wbstage port map(WB_in1_fm_exreg=>WB_in1, aluout_fm_exreg=>aluout_fm_ex,
essout_fm_exreg=>essout_fm_st5, dataoutfmwb=>dataout);
end architecture stage5_beh;

-- UD STAGE MUX

library IEEE;
use IEEE.std_logic_1164.all;

entity wbstage is
port(WB_in1_fm_exreg: in std_logic;
     aluout_fm_exreg, essout_fm_exreg: in std_logic_vector(63 downto 0);
     dataoutfmwb: out std_logic_vector(63 downto 0));
end entity wbstage;

architecture wbstage_beh of wbstage is
signal s6wbmux: std_logic;
signal Write_data_out_fmwb: std_logic_vector(63 downto 0);
begin

s6wbmux <= WB_in1_fm_exreg;

s6process:process(s6wbmux, aluout_fm_exreg, essout_fm_exreg, Write_data_out_fmwb) is
begin
case s6wbmux is
when '0' => Write_data_out_fmwb <= essout_fm_exreg;
when '1' => Write_data_out_fmwb <= aluout_fm_exreg;
when others => null;
end case;
dataoutfmwb <= Write_data_out_fmwb;
end process s6process;
end architecture wbstage_beh;
```

7. MACRO CONTROLLER

```
library IEEE;
use IEEE.std_logic_1164.all;

entity topmac is
port(ESPR_on, EOP, crcchkin, essfullin, locchk, clk: in std_logic;
     macop: in std_logic_vector(7 downto 0); -- decoded 3to8 macro opcode
     fmmactrlrout: out std_logic_vector(15 downto 0);
     incr_pc, macctl: out std_logic);
end entity topmac;

architecture topmac_beh of topmac is
component macctrl is
port(ESPR_on, EOP, crcchkin, essfullin, locchk, clk: in std_logic;
     dec_macop: in std_logic_vector(7 downto 0); -- decoded 3to8 macro opcode
     fm0, fm1, fm2, fm3, fmO, fmT, fmF, fmC, fmA, fmRC: out std_logic; -- for "fmmacctrlr"
     macctl, incr_pc: out std_logic);
end component macctrl;

component fmm is
```

```
port(fm0in, fm1in, fm2in, fm3in, fmOin, fmTin, fmFin, fmCin, fmAin, fmRCin: in std_logic;
    fmmactrlrout: out std_logic_vector(15 downto 0));
end component fmm;

component dec_3to81 is
port(inp: in std_logic_vector(2 downto 0);
    outp: out std_logic_vector(7 downto 0));
end component dec_3to81;

signal insig0, insig1, asig, fsig, osig, Tsig, FMsig, Csig, ACsig, RCsig: std_logic;
signal macop1: std_logic_vector(7 downto 0);
begin

maccomp: macctrl port map(ESPR_on=>ESPR_on, EOP=>EOP, crcchkin=>crcchkin, essfullin=>essfullin,
locchk=>locchk, clk=>clk, dec_macop=>macop1, fm0=>insig0, fm1=>insig1, fm2=>asig, fm3=>fsig,
fmO=>osig, fmT=>Tsig, fmF=>FMsig, fmC=>Csig, fmA=>ACsig, fmRC=>RCsig, macctl=>macctl,
incr_pc=>incr_pc);
fmmcomp: fmm port map(fm0in=>insig0, fm1in=>insig1, fm2in=>asig, fm3in=>fsig, fmOin=>osig,
fmTin=>Tsig, fmFin=>FMsig, fmCin=>Csig, fmAin=>ACsig, fmRCin=>RCsig,
fmmactrlrout=>fmmactrlrout);
decodecomp: dec_3to81 port map(inp=>macop(2 downto 0), outp=>macop1);

end architecture topmac_beh;

-- MACRO CONTROLLER

library IEEE;
use IEEE.std_logic_1164.all;

entity macctrl is
port(ESPR_on, EOP, crcchkin, essfullin, locchk, clk: in std_logic;
    dec_macop: in std_logic_vector(7 downto 0); -- decoded 3to8 macro opcode
    fm0, fm1, fm2, fm3, fmO, fmT, fmF, fmC, fmA, fmRC: out std_logic; -- for "fmmacctrlr"
    macctl, incr_pc: out std_logic);
end entity macctrl;
architecture macctrl_beh of macctrl is

component FD is
port(D, C: in std_logic;
    Q: out std_logic);
end component FD;

signal startespr, st0, st0_bar, st1, st1_bar: std_logic;
signal md0, md1, md2, md3, md4, md5, md6, md7, md8, md9, mdA, mdB, mdC, mdD, mdE, mdF:
std_logic;
signal md10, md11, md12, md13, md14, md15, md16, md17, md18, md19, md1A, md1B, md1C, md1D,
md1E, md1F: std_logic;
signal md20, md21, md22, md23, md24, md25, md26, md27, md28, md29, md2A, md2B, md2C, md2D,
md2E, md2F: std_logic;
signal md30, md31, md32, md33, md34, md35, md36, md37, md38, md39, md3A, md3B, md3C, md3D,
md3E, md3F: std_logic;
signal md40, md41, md42, md43, md44, md45, md46, md47, md48, md49, md4A, md4B, md4C, md4D,
md4E, md4F: std_logic;
signal md50, md51, md52, md53, md54, md55, md56, md57, md58, md59, md5A, md5B, md5C, md5D,
md5E, md5F: std_logic;
```

```
signal md60, md61, md62, md63, md64, md65, md66, md67, md68, md69, md6A, md6B, md6C, md6D,
md6E, md6F: std_logic;
signal md70, md71, md72, md73, md74, md75, md76, md77, md78, md79, md7A, md7B, md7C, md7D,
md7E, md7F: std_logic;
signal md80, md81, md82, md83, md84, md85, md86, md87, md88, md89, md8A, md8B, md8C, md8D,
md8E, md8F: std_logic;
signal md90, md91, md92, md93, md94, md95, md96, md97, md98, md99, md9A, md9B, md9C, md9D,
md9E, md9F: std_logic;
signal mdA0, mdA1, mdA2, mdA3, mdA4, mdA5, mdA6, mdA7, mdA8, mdA9, mdAA, mdAB, mdAC,
mdAD, mdAE, mdAF: std_logic;
signal mdB0, mdB1, mdB2, mdB3, mdB4, mdB5, mdB6, mdB7, mdB8, mdB9, mdBA, mdBB, mdBC,
mdBD, mdBE, mdBF: std_logic;
signal mdC0, mdC1, mdC2, mdC3, mdC4, mdC5, mdC6, mdC7, mdC8, mdC9, mdCA, mdCB, mdCC,
mdCD, mdCE, mdCF: std_logic;
signal mdD0, mdD1, mdD2, mdD3, mdD4, mdD5, mdD6, mdD7, mdD8, mdD9, mdDA, mdDB, mdDC,
mdDD, mdDE, mdDF: std_logic;

signal mt0, mt1, mt2, mt3, mt4, mt5, mt6, mt7, mt8, mt9, mtA, mtB, mtC, mtD, mtE, mtF: std_logic;
signal mt10, mt11, mt12, mt13, mt14, mt15, mt16, mt17, mt18, mt19, mt1A, mt1B, mt1C, mt1D, mt1E,
mt1F: std_logic;
signal mt20, mt21, mt22, mt23, mt24, mt25, mt26, mt27, mt28, mt29, mt2A, mt2B, mt2C, mt2D, mt2E,
mt2F: std_logic;
signal mt30, mt31, mt32, mt33, mt34, mt35, mt36, mt37, mt38, mt39, mt3A, mt3B, mt3C, mt3D, mt3E,
mt3F: std_logic;
signal mt40, mt41, mt42, mt43, mt44, mt45, mt46, mt47, mt48, mt49, mt4A, mt4B, mt4C, mt4D, mt4E,
mt4F: std_logic;
signal mt50, mt51, mt52, mt53, mt54, mt55, mt56, mt57, mt58, mt59, mt5A, mt5B, mt5C, mt5D, mt5E,
mt5F: std_logic;
signal mt60, mt61, mt62, mt63, mt64, mt65, mt66, mt67, mt68, mt69, mt6A, mt6B, mt6C, mt6D, mt6E,
mt6F: std_logic;
signal mt70, mt71, mt72, mt73, mt74, mt75, mt76, mt77, mt78, mt79, mt7A, mt7B, mt7C, mt7D, mt7E,
mt7F: std_logic;
signal mt80, mt81, mt82, mt83, mt84, mt85, mt86, mt87, mt88, mt89, mt8A, mt8B, mt8C, mt8D, mt8E,
mt8F: std_logic;
signal mt90, mt91, mt92, mt93, mt94, mt95, mt96, mt97, mt98, mt99, mt9A, mt9B, mt9C, mt9D, mt9E,
mt9F: std_logic;
signal mtA0, mtA1, mtA2, mtA3, mtA4, mtA5, mtA6, mtA7, mtA8, mtA9, mtAA, mtAB, mtAC, mtAD,
mtAE, mtAF: std_logic;
signal mtB0, mtB1, mtB2, mtB3, mtB4, mtB5, mtB6, mtB7, mtB8, mtB9, mtBA, mtBB, mtBC, mtBD,
mtBE, mtBF: std_logic;
signal mtC0, mtC1, mtC2, mtC3, mtC4, mtC5, mtC6, mtC7, mtC8, mtC9, mtCA, mtCB, mtCC, mtCD,
mtCE, mtCF: std_logic;
signal mtD0, mtD1, mtD2, mtD3, mtD4, mtD5, mtD6, mtD7, mtD8, mtD9, mtDA, mtDB, mtDC, mtDD,
mtDE, mtDF: std_logic;

signal crc_bar, loc_bar, ess_bar, eopbar, incr_pc1, incr_pc2, incr_pc3: std_logic;
begin

dff_st0: FD port map(D=>ESPR_on, C=>clk, Q=>st0);
dff_st1: FD port map(D=>st0, C=>clk, Q=>st1);
st0_bar <= not (st0);
st1_bar <= not (st1);
startespr <= st0 and st1_bar;

dffm0: FD port map(D=>md0, C=>clk, Q=>mt0);
dffm1: FD port map(D=>md1, C=>clk, Q=>mt1);
```

```
dffm2: FD port map(D=>md2, C=>clk, Q=>mt2);
dffm3: FD port map(D=>md3, C=>clk, Q=>mt3);
dffm4: FD port map(D=>md4, C=>clk, Q=>mt4);
dffm5: FD port map(D=>md5, C=>clk, Q=>mt5);
dffm6: FD port map(D=>md6, C=>clk, Q=>mt6);
dffm7: FD port map(D=>md7, C=>clk, Q=>mt7);
dffm8: FD port map(D=>md8, C=>clk, Q=>mt8);
dffm9: FD port map(D=>md9, C=>clk, Q=>mt9);
dffmA: FD port map(D=>mdA, C=>clk, Q=>mtA);
dffmB: FD port map(D=>mdB, C=>clk, Q=>mtB);
dffmC: FD port map(D=>mdC, C=>clk, Q=>mtC);
dffmD: FD port map(D=>mdD, C=>clk, Q=>mtD);
dffmE: FD port map(D=>mdE, C=>clk, Q=>mtE);
dffmF: FD port map(D=>mdF, C=>clk, Q=>mtF);

dffm10: FD port map(D=>md10, C=>clk, Q=>mt10);
dffm11: FD port map(D=>md11, C=>clk, Q=>mt11);
dffm12: FD port map(D=>md12, C=>clk, Q=>mt12);
dffm13: FD port map(D=>md13, C=>clk, Q=>mt13);
dffm14: FD port map(D=>md14, C=>clk, Q=>mt14);
dffm15: FD port map(D=>md15, C=>clk, Q=>mt15);
dffm16: FD port map(D=>md16, C=>clk, Q=>mt16);
dffm17: FD port map(D=>md17, C=>clk, Q=>mt17);
dffm18: FD port map(D=>md18, C=>clk, Q=>mt18);
dffm19: FD port map(D=>md19, C=>clk, Q=>mt19);
dffm1A: FD port map(D=>md1A, C=>clk, Q=>mt1A);
dffm1B: FD port map(D=>md1B, C=>clk, Q=>mt1B);
dffm1C: FD port map(D=>md1C, C=>clk, Q=>mt1C);
dffm1D: FD port map(D=>md1D, C=>clk, Q=>mt1D);
dffm1E: FD port map(D=>md1E, C=>clk, Q=>mt1E);
dffm1F: FD port map(D=>md1F, C=>clk, Q=>mt1F);

dffm20: FD port map(D=>md20, C=>clk, Q=>mt20);
dffm21: FD port map(D=>md21, C=>clk, Q=>mt21);
dffm22: FD port map(D=>md22, C=>clk, Q=>mt22);
dffm23: FD port map(D=>md23, C=>clk, Q=>mt23);
dffm24: FD port map(D=>md24, C=>clk, Q=>mt24);
dffm25: FD port map(D=>md25, C=>clk, Q=>mt25);
dffm26: FD port map(D=>md26, C=>clk, Q=>mt26);
dffm27: FD port map(D=>md27, C=>clk, Q=>mt27);
dffm28: FD port map(D=>md28, C=>clk, Q=>mt28);
dffm29: FD port map(D=>md29, C=>clk, Q=>mt29);
dffm2A: FD port map(D=>md2A, C=>clk, Q=>mt2A);
dffm2B: FD port map(D=>md2B, C=>clk, Q=>mt2B);
dffm2C: FD port map(D=>md2C, C=>clk, Q=>mt2C);
dffm2D: FD port map(D=>md2D, C=>clk, Q=>mt2D);
dffm2E: FD port map(D=>md2E, C=>clk, Q=>mt2E);
dffm2F: FD port map(D=>md2F, C=>clk, Q=>mt2F);
dffm30: FD port map(D=>md30, C=>clk, Q=>mt30);
dffm31: FD port map(D=>md31, C=>clk, Q=>mt31);
dffm32: FD port map(D=>md32, C=>clk, Q=>mt32);
dffm33: FD port map(D=>md33, C=>clk, Q=>mt33);
dffm34: FD port map(D=>md34, C=>clk, Q=>mt34);
dffm35: FD port map(D=>md35, C=>clk, Q=>mt35);
dffm36: FD port map(D=>md36, C=>clk, Q=>mt36);
dffm37: FD port map(D=>md37, C=>clk, Q=>mt37);
```

```
dffm38: FD port map(D=>md38, C=>clk, Q=>mt38);
dffm39: FD port map(D=>md39, C=>clk, Q=>mt39);
dffm3A: FD port map(D=>md3A, C=>clk, Q=>mt3A);
dffm3B: FD port map(D=>md3B, C=>clk, Q=>mt3B);
dffm3C: FD port map(D=>md3C, C=>clk, Q=>mt3C);
dffm3D: FD port map(D=>md3D, C=>clk, Q=>mt3D);
dffm3E: FD port map(D=>md3E, C=>clk, Q=>mt3E);
dffm3F: FD port map(D=>md3F, C=>clk, Q=>mt3F);

dffm40: FD port map(D=>md40, C=>clk, Q=>mt40);
dffm41: FD port map(D=>md41, C=>clk, Q=>mt41);
dffm42: FD port map(D=>md42, C=>clk, Q=>mt42);
dffm43: FD port map(D=>md43, C=>clk, Q=>mt43);
dffm44: FD port map(D=>md44, C=>clk, Q=>mt44);
dffm45: FD port map(D=>md45, C=>clk, Q=>mt45);
dffm46: FD port map(D=>md46, C=>clk, Q=>mt46);
dffm47: FD port map(D=>md47, C=>clk, Q=>mt47);
dffm48: FD port map(D=>md48, C=>clk, Q=>mt48);
dffm49: FD port map(D=>md49, C=>clk, Q=>mt49);
dffm4A: FD port map(D=>md4A, C=>clk, Q=>mt4A);
dffm4B: FD port map(D=>md4B, C=>clk, Q=>mt4B);
dffm4C: FD port map(D=>md4C, C=>clk, Q=>mt4C);
dffm4D: FD port map(D=>md4D, C=>clk, Q=>mt4D);
dffm4E: FD port map(D=>md4E, C=>clk, Q=>mt4E);
dffm4F: FD port map(D=>md4F, C=>clk, Q=>mt4F);

dffm50: FD port map(D=>md50, C=>clk, Q=>mt50);
dffm51: FD port map(D=>md51, C=>clk, Q=>mt51);
dffm52: FD port map(D=>md52, C=>clk, Q=>mt52);
dffm53: FD port map(D=>md53, C=>clk, Q=>mt53);
dffm54: FD port map(D=>md54, C=>clk, Q=>mt54);
dffm55: FD port map(D=>md55, C=>clk, Q=>mt55);
dffm56: FD port map(D=>md56, C=>clk, Q=>mt56);
dffm57: FD port map(D=>md57, C=>clk, Q=>mt57);
dffm58: FD port map(D=>md58, C=>clk, Q=>mt58);
dffm59: FD port map(D=>md59, C=>clk, Q=>mt59);
dffm5A: FD port map(D=>md5A, C=>clk, Q=>mt5A);
dffm5B: FD port map(D=>md5B, C=>clk, Q=>mt5B);
dffm5C: FD port map(D=>md5C, C=>clk, Q=>mt5C);
dffm5D: FD port map(D=>md5D, C=>clk, Q=>mt5D);
dffm5E: FD port map(D=>md5E, C=>clk, Q=>mt5E);
dffm5F: FD port map(D=>md5F, C=>clk, Q=>mt5F);

dffm60: FD port map(D=>md60, C=>clk, Q=>mt60);
dffm61: FD port map(D=>md61, C=>clk, Q=>mt61);
dffm62: FD port map(D=>md62, C=>clk, Q=>mt62);
dffm63: FD port map(D=>md63, C=>clk, Q=>mt63);
dffm64: FD port map(D=>md64, C=>clk, Q=>mt64);
dffm65: FD port map(D=>md65, C=>clk, Q=>mt65);
dffm66: FD port map(D=>md66, C=>clk, Q=>mt66);
dffm67: FD port map(D=>md67, C=>clk, Q=>mt67);
dffm68: FD port map(D=>md68, C=>clk, Q=>mt68);
dffm69: FD port map(D=>md69, C=>clk, Q=>mt69);
dffm6A: FD port map(D=>md6A, C=>clk, Q=>mt6A);
dffm6B: FD port map(D=>md6B, C=>clk, Q=>mt6B);
dffm6C: FD port map(D=>md6C, C=>clk, Q=>mt6C);
```

```
dffm6D: FD port map(D=>md6D, C=>clk, Q=>mt6D);
dffm6E: FD port map(D=>md6E, C=>clk, Q=>mt6E);
dffm6F: FD port map(D=>md6F, C=>clk, Q=>mt6F);

dffm70: FD port map(D=>md70, C=>clk, Q=>mt70);
dffm71: FD port map(D=>md71, C=>clk, Q=>mt71);
dffm72: FD port map(D=>md72, C=>clk, Q=>mt72);
dffm73: FD port map(D=>md73, C=>clk, Q=>mt73);
dffm74: FD port map(D=>md74, C=>clk, Q=>mt74);
dffm75: FD port map(D=>md75, C=>clk, Q=>mt75);
dffm76: FD port map(D=>md76, C=>clk, Q=>mt76);
dffm77: FD port map(D=>md77, C=>clk, Q=>mt77);
dffm78: FD port map(D=>md78, C=>clk, Q=>mt78);
dffm79: FD port map(D=>md79, C=>clk, Q=>mt79);
dffm7A: FD port map(D=>md7A, C=>clk, Q=>mt7A);
dffm7B: FD port map(D=>md7B, C=>clk, Q=>mt7B);
dffm7C: FD port map(D=>md7C, C=>clk, Q=>mt7C);
dffm7D: FD port map(D=>md7D, C=>clk, Q=>mt7D);
dffm7E: FD port map(D=>md7E, C=>clk, Q=>mt7E);
dffm7F: FD port map(D=>md7F, C=>clk, Q=>mt7F);

dffm80: FD port map(D=>md80, C=>clk, Q=>mt80);
dffm81: FD port map(D=>md81, C=>clk, Q=>mt81);
dffm82: FD port map(D=>md82, C=>clk, Q=>mt82);
dffm83: FD port map(D=>md83, C=>clk, Q=>mt83);
dffm84: FD port map(D=>md84, C=>clk, Q=>mt84);
dffm85: FD port map(D=>md85, C=>clk, Q=>mt85);
dffm86: FD port map(D=>md86, C=>clk, Q=>mt86);
dffm87: FD port map(D=>md87, C=>clk, Q=>mt87);
dffm88: FD port map(D=>md88, C=>clk, Q=>mt88);
dffm89: FD port map(D=>md89, C=>clk, Q=>mt89);
dffm8A: FD port map(D=>md8A, C=>clk, Q=>mt8A);
dffm8B: FD port map(D=>md8B, C=>clk, Q=>mt8B);
dffm8C: FD port map(D=>md8C, C=>clk, Q=>mt8C);
dffm8D: FD port map(D=>md8D, C=>clk, Q=>mt8D);
dffm8E: FD port map(D=>md8E, C=>clk, Q=>mt8E);
dffm8F: FD port map(D=>md8F, C=>clk, Q=>mt8F);

dffm90: FD port map(D=>md90, C=>clk, Q=>mt90);
dffm91: FD port map(D=>md91, C=>clk, Q=>mt91);
dffm92: FD port map(D=>md92, C=>clk, Q=>mt92);
dffm93: FD port map(D=>md93, C=>clk, Q=>mt93);
dffm94: FD port map(D=>md94, C=>clk, Q=>mt94);
dffm95: FD port map(D=>md95, C=>clk, Q=>mt95);
dffm96: FD port map(D=>md96, C=>clk, Q=>mt96);
dffm97: FD port map(D=>md97, C=>clk, Q=>mt97);
dffm98: FD port map(D=>md98, C=>clk, Q=>mt98);
dffm99: FD port map(D=>md99, C=>clk, Q=>mt99);
dffm9A: FD port map(D=>md9A, C=>clk, Q=>mt9A);
dffm9B: FD port map(D=>md9B, C=>clk, Q=>mt9B);
dffm9C: FD port map(D=>md9C, C=>clk, Q=>mt9C);
dffm9D: FD port map(D=>md9D, C=>clk, Q=>mt9D);
dffm9E: FD port map(D=>md9E, C=>clk, Q=>mt9E);
dffm9F: FD port map(D=>md9F, C=>clk, Q=>mt9F);

dffmA0: FD port map(D=>mdA0, C=>clk, Q=>mtA0);
```

```
dffmA1: FD port map(D=>mdA1, C=>clk, Q=>mtA1);
dffmA2: FD port map(D=>mdA2, C=>clk, Q=>mtA2);
dffmA3: FD port map(D=>mdA3, C=>clk, Q=>mtA3);
dffmA4: FD port map(D=>mdA4, C=>clk, Q=>mtA4);
dffmA5: FD port map(D=>mdA5, C=>clk, Q=>mtA5);
dffmA6: FD port map(D=>mdA6, C=>clk, Q=>mtA6);
dffmA7: FD port map(D=>mdA7, C=>clk, Q=>mtA7);
dffmA8: FD port map(D=>mdA8, C=>clk, Q=>mtA8);
dffmA9: FD port map(D=>mdA9, C=>clk, Q=>mtA9);
dffmAA: FD port map(D=>mdAA, C=>clk, Q=>mtAA);
dffmAB: FD port map(D=>mdAB, C=>clk, Q=>mtAB);
dffmAC: FD port map(D=>mdAC, C=>clk, Q=>mtAC);
dffmAD: FD port map(D=>mdAD, C=>clk, Q=>mtAD);
dffmAE: FD port map(D=>mdAE, C=>clk, Q=>mtAE);
dffmAF: FD port map(D=>mdAF, C=>clk, Q=>mtAF);

dffmB0: FD port map(D=>mdB0, C=>clk, Q=>mtB0);
dffmB1: FD port map(D=>mdB1, C=>clk, Q=>mtB1);
dffmB2: FD port map(D=>mdB2, C=>clk, Q=>mtB2);
dffmB3: FD port map(D=>mdB3, C=>clk, Q=>mtB3);
dffmB4: FD port map(D=>mdB4, C=>clk, Q=>mtB4);
dffmB5: FD port map(D=>mdB5, C=>clk, Q=>mtB5);
dffmB6: FD port map(D=>mdB6, C=>clk, Q=>mtB6);
dffmB7: FD port map(D=>mdB7, C=>clk, Q=>mtB7);
dffmB8: FD port map(D=>mdB8, C=>clk, Q=>mtB8);
dffmB9: FD port map(D=>mdB9, C=>clk, Q=>mtB9);
dffmBA: FD port map(D=>mdBA, C=>clk, Q=>mtBA);
dffmBB: FD port map(D=>mdBB, C=>clk, Q=>mtBB);
dffmBC: FD port map(D=>mdBC, C=>clk, Q=>mtBC);
dffmBD: FD port map(D=>mdBD, C=>clk, Q=>mtBD);
dffmBE: FD port map(D=>mdBE, C=>clk, Q=>mtBE);
dffmBF: FD port map(D=>mdBF, C=>clk, Q=>mtBF);

dffmC0: FD port map(D=>mdC0, C=>clk, Q=>mtC0);
dffmC1: FD port map(D=>mdC1, C=>clk, Q=>mtC1);
dffmC2: FD port map(D=>mdC2, C=>clk, Q=>mtC2);
dffmC3: FD port map(D=>mdC3, C=>clk, Q=>mtC3);
dffmC4: FD port map(D=>mdC4, C=>clk, Q=>mtC4);
dffmC5: FD port map(D=>mdC5, C=>clk, Q=>mtC5);
dffmC6: FD port map(D=>mdC6, C=>clk, Q=>mtC6);
dffmC7: FD port map(D=>mdC7, C=>clk, Q=>mtC7);
dffmC8: FD port map(D=>mdC8, C=>clk, Q=>mtC8);
dffmC9: FD port map(D=>mdC9, C=>clk, Q=>mtC9);
dffmCA: FD port map(D=>mdCA, C=>clk, Q=>mtCA);
dffmCB: FD port map(D=>mdCB, C=>clk, Q=>mtCB);
dffmCC: FD port map(D=>mdCC, C=>clk, Q=>mtCC);
dffmCD: FD port map(D=>mdCD, C=>clk, Q=>mtCD);
dffmCE: FD port map(D=>mdCE, C=>clk, Q=>mtCE);
dffmCF: FD port map(D=>mdCF, C=>clk, Q=>mtCF);

dffmD0: FD port map(D=>mdD0, C=>clk, Q=>mtD0);
dffmD1: FD port map(D=>mdD1, C=>clk, Q=>mtD1);
dffmD2: FD port map(D=>mdD2, C=>clk, Q=>mtD2);
dffmD3: FD port map(D=>mdD3, C=>clk, Q=>mtD3);
dffmD4: FD port map(D=>mdD4, C=>clk, Q=>mtD4);
dffmD5: FD port map(D=>mdD5, C=>clk, Q=>mtD5);
```

```
dffmD6: FD port map(D=>mdD6, C=>clk, Q=>mtD6);
dffmD7: FD port map(D=>mdD7, C=>clk, Q=>mtD7);
dffmD8: FD port map(D=>mdD8, C=>clk, Q=>mtD8);
dffmD9: FD port map(D=>mdD9, C=>clk, Q=>mtD9);
dffmDA: FD port map(D=>mdDA, C=>clk, Q=>mtDA);
dffmDB: FD port map(D=>mdDB, C=>clk, Q=>mtDB);
dffmDC: FD port map(D=>mdDC, C=>clk, Q=>mtDC);
dffmDD: FD port map(D=>mdDD, C=>clk, Q=>mtDD);
dffmDE: FD port map(D=>mdDE, C=>clk, Q=>mtDE);
dffmDF: FD port map(D=>mdDF, C=>clk, Q=>mtDF);

crc_bar <= not (crcchkin);
loc_bar <= not (locchk);
ess_bar <= not (essfullin);
eopbar <= not (EOP);

-- state equations
md0 <= startespr;
md1 <= (mt0 or (mt2 and eopbar));
md2 <= mt1;
md3 <= mt2 and EOP;
md4 <= mt3 and crc_bar;
md5 <= mt4;
md6 <= mt3 and crcchkin;
md7 <= mt6 and locchk;
md8 <= mt7;
md9 <= mt6 and loc_bar;
mdA <= mt9 and essfullin;
mdB <= mtA;

-- COUNT
mdC <= mt9 and ess_bar and dec_macop(0);
mdD <= mtC;
mdE <= mtD;
mdF <= mtE;
md10 <= mtF;
md11 <= mt10;
md12 <= mt11;
md13 <= mt12;
md14 <= mt13;
md15 <= mt14;
md16 <= mt15;
md17 <= mt16;
md18 <= mt17;
md19 <= mt18;
md1A <= mt19;
md1B <= mt1A;
md1C <= mt1B;
md1D <= mt1C;
md1E <= mt1D;
md1F <= mt1E;
md20 <= mt1F;
md21 <= mt20;
md22 <= mt21;
md23 <= mt22;
md24 <= mt23;
```

```
md25 <= mt24;
md26 <= mt25;
md27 <= mt26;
md28 <= mt27;
md29 <= mt28;
md2A <= mt29;
md2B <= mt2A;
md2C <= mt2B;

--COMPARE
md2D <= mt9 and ess_bar and dec_macop(1);
md2E <= mt2D;
md2F <= mt2E;
md30 <= mt2F;
md31 <= mt30;
md32 <= mt31;
md33 <= mt32;
md34 <= mt33;
md35 <= mt34;
md36 <= mt35;
md37 <= mt36;
md38 <= mt37;
md39 <= mt38;
md3A <= mt39;
md3B <= mt3A;
md3C <= mt3B;
md3D <= mt3C;
md3E <= mt3D;
md3F <= mt3E;
md40 <= mt3F;
md41 <= mt40;
md42 <= mt41;
md43 <= mt42;
md44 <= mt43;
md45 <= mt44;
md46 <= mt45;

--COLLECT
md47 <= mt9 and ess_bar and dec_macop(2);
md48 <= mt47;
md49 <= mt48;
md4A <= mt49;
md4B <= mt4A;
md4C <= mt4B;
md4D <= mt4C;
md4E <= mt4D;
md4F <= mt4E;
md50 <= mt4F;
md51 <= mt50;
md52 <= mt51;
md53 <= mt52;
md54 <= mt53;
md55 <= mt54;
md56 <= mt55;
md57 <= mt56;
md58 <= mt57;
```

```
md59 <= mt58;
md5A <= mt59;
md5B <= mt5A;
md5C <= mt5B;
md5D <= mt5C;
md5E <= mt5D;
md5F <= mt5E;
md60 <= mt5F;
md61 <= mt60;
md62 <= mt61;
md63 <= mt62;
md64 <= mt63;
md65 <= mt64;
md66 <= mt65;
md67 <= mt66;
md68 <= mt67;
md69 <= mt68;

--RCHLD
md6A <= mt9 and ess_bar and dec_macop(3);
md6B <= mt6A;
md6C <= mt6B;
md6D <= mt6C;
md6E <= mt6D;
md6F <= mt6E;
md70 <= mt6F;
md71 <= mt70;
md72 <= mt71;
md73 <= mt72;
md74 <= mt73;
md75 <= mt74;
md76 <= mt75;
md77 <= mt76;
md78 <= mt77;
md79 <= mt78;
md7A <= mt79;
md7B <= mt7A;
md7C <= mt7B;
md7D <= mt7C;
md7E <= mt7D;
md7F <= mt7E;
md80 <= mt7F;
md81 <= mt80;
md82 <= mt81;
md83 <= mt82;
md84 <= mt83;
md85 <= mt84;
md86 <= mt85;
md87 <= mt86;
md88 <= mt87;
md89 <= mt88;
md8A <= mt89;
md8B <= mt8A;
md8C <= mt8B;
md8D <= mt8C;
md8E <= mt8D;
```

```
md8F <= mt8E;
md90 <= mt8F;
md91 <= mt90;
md92 <= mt91;
md93 <= mt92;
md94 <= mt93;
md95 <= mt94;
md96 <= mt95;

--RCOLLECT
md97 <= mt9 and ess_bar and dec_macop(4);
md98 <= mt97;
md99 <= mt98;
md9A <= mt99;
md9B <= mt9A;
md9C <= mt9B;
md9D <= mt9C;
md9E <= mt9D;
md9F <= mt9E;
mdA0 <= mt9F;
mdA1 <= mtA0;
mdA2 <= mtA1;
mdA3 <= mtA2;
mdA4 <= mtA3;
mdA5 <= mtA4;
mdA6 <= mtA5;
mdA7 <= mtA6;
mdA8 <= mtA7;
mdA9 <= mtA8;
mdAA <= mtA9;
mdAB <= mtAA;
mdAC <= mtAB;
mdAD <= mtAC;
mdAE <= mtAD;
mdAF <= mtAE;
mdB0 <= mtAF;
mdB1 <= mtB0;
mdB2 <= mtB1;
mdB3 <= mtB2;
mdB4 <= mtB3;
mdB5 <= mtB4;
mdB6 <= mtB5;
mdB7 <= mtB6;
mdB8 <= mtB7;
mdB9 <= mtB8;
mdBA <= mtB9;
mdBB <= mtBA;
mdBC <= mtBB;
mdBD <= mtBC;
mdBE <= mtBD;
mdBF <= mtBE;
mdC0 <= mtBF;
mdC1 <= mtC0;
mdC2 <= mtC1;
mdC3 <= mtC2;
mdC4 <= mtC3;
```

```
mdC5 <= mtC4;
mdC6 <= mtC5;
mdC7 <= mtC6;
mdC8 <= mtC7;
mdC9 <= mtC8;
mdCA <= mtC9;
mdCB <= mtCA;
mdCC <= mtCB;
mdCD <= mtCC;
mdCE <= mtCD;
mdCF <= mtCE;
mdD0 <= mtCF;
mdD1 <= mtD0;
mdD2 <= mtD1;
mdD3 <= mtD2;
mdD4 <= mtD3;
mdD5 <= mtD4;
mdD6 <= mtD5;
mdD7 <= mtD6;
mdD8 <= mtD7;
mdD9 <= mtD8;
mdDA <= mtD9;
mdDB <= mtDA;
mdDC <= mtDB;
mdDD <= mtDC;
mdDE <= mtDD;
mdDF <= mtDE;

-- Output equations
macctl <= startespr or mt0 or mt4 or mt5 or mt7 or mt8 or mtA or mtB or mtC or mt2D or mt47 or mt6A or
mt97;
fm0 <= startespr;
fm1 <= mt0; -- IN
fm2 <= mt4 or mtA; -- ABORT2 for example
fm3 <= mt7; -- FWD
fmO <= mt5 or mt8 or mtB; -- OUT
fmT <= mtC;
fmF <= mt2D;
fmC <= mt47;
fmA <= mt6A;
fmRC <= mt97;

incr_pc1 <= mtC or mtD or mtE or mtF or mt10 or mt11 or mt12 or mt13 or mt14 or mt15 or mt16 or mt17
or mt18 or mt19 or mt1A or mt1B or mt1C or mt1D or mt1E or mt1F or mt20 or mt21 or mt22 or mt23 or
mt24 or mt26 or mt27 or mt28 or mt29 or mt2A or mt2B or mt2C or mt2D or mt2E or mt2F or mt30 or
mt31 or mt32 or mt33 or mt34 or mt35 or mt36 or mt37 or mt38 or mt39 or mt3A or mt3B or mt3C or
mt3D or mt3E or mt40 or mt41 or mt42 or mt43 or mt44 or mt45 or mt46 or mt47 or mt48 or mt49 or
mt4A or mt4B or mt4C or mt4D or mt4E or mt4F or mt50 or mt51 or mt52 or mt53 or mt54 or mt55 or
mt56 or mt57 or mt58 or mt59 or mt5A or mt5B or mt5C or mt5D or mt5E or mt5F;

incr_pc2 <= mt60 or mt61 or mt63 or mt64 or mt65 or mt66 or mt67 or mt68 or mt69 or mt6A or mt6B or
mt6C or mt6D or mt6E or mt6F or mt70 or mt71 or mt72 or mt73 or mt74 or mt75 or mt76 or mt77 or
mt78 or mt79 or mt7A or mt7B or mt7C or mt7D or mt7E or mt7F or mt80 or mt81 or mt82 or mt83 or
mt84 or mt85 or mt86 or mt87 or mt88 or mt89 or mt8A or mt8B or mt8C or mt8D or mt8E or mt90 or
mt91 or mt92 or mt93 or mt94 or mt95 or mt96 or mt97 or mt98 or mt99 or mt9A or mt9B or mt9C or
```

```
mt9D or mt9E or mt9F or mtA0 or mtA1 or mtA2 or mtA3 or mtA4 or mtA5 or mtA6 or mtA7 or mtA8 or
mtA9 or mtAA or mtAB or mtAC or mtAD or mtAE or mtAF;

incr_pc3 <= mtB0 or mtB1 or mtB2 or mtB3 or mtB4 or mtB5 or mtB6 or mtB7 or mtB8 or mtB9 or mtBA
or mtBB or mtBC or mtBD or mtBE or mtBF or mtC0 or mtC1 or mtC2 or mtC3 or mtC4 or mtC5 or mtC6
or mtC7 or mtC8 or mtC9 or mtCA or mtCB or mtCC or mtCD or mtCE or mtCF or mtD0 or mtD1 or
mtD2 or mtD3 or mtD4 or mtD6 or mtD7 or mtD8 or mtD9 or mtDA or mtDB or mtDC or mtDD or mtDE
or mt25 or mt3F or mt62 or mt87 or mtD5;

incr_pc <= incr_pc1 or incr_pc2 or incr_pc3;
end architecture macctrl_beh;

-- For getting fmmactrlr address for specific macro and micro instructions

library IEEE;
use IEEE.std_logic_1164.all;

entity fmm is
port(fm0in, fm1in, fm2in, fm3in, fmOin, fmTin, fmFin, fmCin, fmAin, fmRCin: in std_logic;
    fmmactrlrout: out std_logic_vector(15 downto 0));
end entity fmm;

architecture fmm_beh of fmm is
signal fmsig: std_logic_vector(9 downto 0);
begin

fmsig <= fm0in & fm1in & fm2in & fm3in & fmOin & fmTin & fmFin & fmCin & fmAin & fmRCin;

process(fmsig) is
begin
case fmsig is
when "1000000000" => fmmactrlrout <= (others => '0'); -- address 0 for IN
when "0100000000" => fmmactrlrout <= "0000000000000001"; -- 1 for IN
when "0010000000" => fmmactrlrout <= "0000000000000010"; -- 2 for Abort2
when "0001000000" => fmmactrlrout <= "0000000000000011"; -- 3 for Fwd
when "0000100000" => fmmactrlrout <= "0000000000011100"; -- 1C for Out
when "0000010000" => fmmactrlrout <= "0000000000000101"; -- 5 for THRESH
when "0000001000" => fmmactrlrout <= "0000000000100110"; -- 26 for FINDM
when "0000000100" => fmmactrlrout <= "0000000001000000"; -- 40 for COLLECT
when "0000000010" => fmmactrlrout <= "0000000001100011"; -- 63 for RCHLD
when "0000000001" => fmmactrlrout <= "0000000010010000"; -- 90 for RCOLLECT
when others => fmmactrlrout <= "0000000000100010" ; -- address 22 for NOP
end case;
end process;
end architecture fmm_beh;

-- 3to8 Decoder
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity dec_3to81 is
port(inp: in std_logic_vector(2 downto 0);
    outp: out std_logic_vector(7 downto 0));
end entity dec_3to81;
```

```
architecture dec_3to81_beh of dec_3to81 is
begin

process(inp) is
begin
case inp is
when "000" => outp <= "00000001";
when "001" => outp <= "00000010";
when "010" => outp <= "00000100";
when "011" => outp <= "00001000";
when "100" => outp <= "00010000";
when "101" => outp <= "00100000";
when "110" => outp <= "01000000";
when "111" => outp <= "10000000";
when others => outp <= (others => '0');
end case;
end process;
end architecture dec_3to81_beh;
```

8. Instruction Memory Initialization

```
--Instruction Memory for 'COUNT' Macro Instruction

library IEEE;
use IEEE.std_logic_1164.all;

--synopsys translate_off;
library unisim;
use unisim.vcomponents.all;
--synopsys translate_on;

entity INSTMEM is
port(clk, we, en, rst: in std_logic;
    addr: in std_logic_vector(7 downto 0);
    inst_in: in std_logic_vector(63 downto 0);
    inst_out: out std_logic_vector(63 downto 0));
end entity INSTMEM;

architecture behavioural of INSTMEM is

component RAMB4_S16 is
port(ADDR: in std_logic_vector(7 downto 0);
    CLK: in std_logic;
    DI: in std_logic_vector(15 downto 0);
    DO: out std_logic_vector(15 downto 0);
    EN, RST, WE: in std_logic);
end component RAMB4_S16;

attribute INIT_00: string;
attribute INIT_01: string;
attribute INIT_02: string;
attribute INIT_03: string;
attribute INIT_04: string;
attribute INIT_05: string;
attribute INIT_06: string;
attribute INIT_07: string;
attribute INIT_08: string;
attribute INIT_09: string;
attribute INIT_0A: string;
attribute INIT_0B: string;
attribute INIT_0C: string;
attribute INIT_0D: string;
attribute INIT_0E: string;
attribute INIT_0F: string;

attribute INIT_00 of Instram0 : label is
"00000000000006C0000000C000000000104000000000000000000004000000000";
attribute INIT_01 of Instram0 : label is
"0000000005400800000000000000000000900000014000000000080000000000";
attribute INIT_02 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_00 of Instram1 : label is
"0100000000000000800000000000000000000000810001000100010000000000";
attribute INIT_01 of Instram1 : label is
"0000000000000000000000000000000000000100010000000000000000008000";
attribute INIT_02 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_08 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram2 : label is
"0001000000000000004100600000000000000041000100601800000000000000";
attribute INIT_01 of Instram2 : label is
"0000000000000000004100000000000000028000001000000000000000000410001";
attribute INIT_02 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram3 : label is
"2C8000000000080007800540000000008400 7C001C051C0524A4208000000400";
attribute INIT_01 of Instram3 : label is
"000000007000084007C0000000000140064041CA054800000000084007C001C04";
attribute INIT_02 of Instram3 : label is
"00000000000000000000000000000000000000008000C0000000000008008800";
```

```
attribute INIT_03 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

begin

Instram0: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"00000000000006C0000000C0000000001040000000000000000004000000000 0",
INIT_01 => X"0000000005400800000000000000000900000001400000000000800000000000",
INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(15 downto 0), DO=>inst_out(15 downto 0), EN=>en,
RST=>rst, WE=>we);

Instram1: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"0100000000000000800000000000000000000810001000100010000000000000",
INIT_01 => X"0000000000000000000000000000100010000000000000000000000000008000",
INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(31 downto 16), DO=>inst_out(31 downto 16), EN=>en,
RST=>rst, WE=>we);

Instram2: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"0001000000000000000410060000000000000004100010060180000000000000",
INIT_01 => X"0000000000000000000410000000000028000001000000000000000000410001",
INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(47 downto 32), DO=>inst_out(47 downto 32), EN=>en,
RST=>rst, WE=>we);

Instram3: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"2C80000000008000780054000000000084007C001C051C0524A4208000000400",
INIT_01 => X"0000000700084007C000000000014006404 1CA0548000000000000084007C001C04",
INIT_02 => X"00000000000000000000000000000000000000008000C0000000000008008800",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(63 downto 48), DO=>inst_out(63 downto 48), EN=>en,
RST=>rst, WE=>we);
end architecture behavioural;

--Instruction Memory for 'COMPARE' Macro Instruction

library IEEE;
use IEEE.std_logic_1164.all;

--synopsys translate_off;
library unisim;
use unisim.vcomponents.all;
--synopsys translate_on;

entity INSTMEM is
port(clk, we, en, rst: in std_logic;
    addr: in std_logic_vector(7 downto 0);
    inst_in: in std_logic_vector(63 downto 0);
    inst_out: out std_logic_vector(63 downto 0));
end entity INSTMEM;

architecture behavioural of INSTMEM is

component RAMB4_S16 is
port(ADDR: in std_logic_vector(7 downto 0);
    CLK: in std_logic;
    DI: in std_logic_vector(15 downto 0);
    DO: out std_logic_vector(15 downto 0);
    EN, RST, WE: in std_logic);
end component RAMB4_S16;

attribute INIT_00: string;
attribute INIT_01: string;
attribute INIT_02: string;
attribute INIT_03: string;
attribute INIT_04: string;
attribute INIT_05: string;
attribute INIT_06: string;
attribute INIT_07: string;
attribute INIT_08: string;
attribute INIT_09: string;
attribute INIT_0A: string;
attribute INIT_0B: string;
attribute INIT_0C: string;
attribute INIT_0D: string;
attribute INIT_0E: string;
attribute INIT_0F: string;

attribute INIT_00 of Instram0 : label is
"0000014000000540000000C00000000000000000000000000000000000000000";
```

```
attribute INIT_01 of Instram0 : label is
"0000000000000000000000000000078000000000014000000000000000054001C0";
attribute INIT_02 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram1 : label is
"0100010000000000800000000000000000000000000000000000000000000000";
attribute INIT_01 of Instram1 : label is
"0000000000000000000000000000000000008000010000000000000000000080";
attribute INIT_02 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_0C of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram2 : label is
"0001000000000000041006000000000000000000000000000000000000000000";
attribute INIT_01 of Instram2 : label is
"0000000000000000000000000000000004100010000000000000000042800000";
attribute INIT_02 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram3 : label is
"1C8054A000008000780054000000000000000000000000000000000000000400";
attribute INIT_01 of Instram3 : label is
"080088000000000008000C00000084007C001C0554A000000000140000005400";
attribute INIT_02 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_03 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_07 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
begin

Instram0: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"0000014000000540000000C00000000000000000000000000000000000000000",
INIT_01 => X"0000000000000000000000000000078000000000140000000000000000054001C0",
INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(15 downto 0), DO=>inst_out(15 downto 0), EN=>en,
RST=>rst, WE=>we);

Instram1: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"0100010000000000800000000000000000000000000000000000000000000000",
INIT_01 => X"0000000000000000000000000000000000000000800001000000000000000080",
INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(31 downto 16), DO=>inst_out(31 downto 16), EN=>en,
RST=>rst, WE=>we);

Instram2: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"0001000000000000000041006000000000000000000000000000000000000000",
INIT_01 => X"0000000000000000000000000000000000004100010000000000000042800000",
INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(47 downto 32), DO=>inst_out(47 downto 32), EN=>en,
RST=>rst, WE=>we);

Instram3: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"1C8054A000008000780054000000000000000000000000000000000000000400",
INIT_01 => X"080088000000000008000C00000084007C001C0554A000000000140000005400",
INIT_02 => X"000000000000000000000000000000000000000008000C000000000008008800",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(63 downto 48), DO=>inst_out(63 downto 48), EN=>en,
RST=>rst, WE=>we);
end architecture behavioural;
```

```
--Instruction Memory for 'COLLECT' Macro Instruction

library IEEE;
use IEEE.std_logic_1164.all;

--synopsys translate_off;
library unisim;
use unisim.vcomponents.all;
--synopsys translate_on;

entity INSTMEM is
port(clk, we, en, rst: in std_logic;
   addr: in std_logic_vector(7 downto 0);
   inst_in: in std_logic_vector(63 downto 0);
   inst_out: out std_logic_vector(63 downto 0));
end entity INSTMEM;

architecture behavioural of INSTMEM is

component RAMB4_S16 is
port(ADDR: in std_logic_vector(7 downto 0);
   CLK: in std_logic;
   DI: in std_logic_vector(15 downto 0);
   DO: out std_logic_vector(15 downto 0);
   EN, RST, WE: in std_logic);
end component RAMB4_S16;

attribute INIT_00: string;
attribute INIT_01: string;
attribute INIT_02: string;
attribute INIT_03: string;
attribute INIT_04: string;
attribute INIT_05: string;
attribute INIT_06: string;
attribute INIT_07: string;
attribute INIT_08: string;
attribute INIT_09: string;
attribute INIT_0A: string;
attribute INIT_0B: string;
attribute INIT_0C: string;
attribute INIT_0D: string;
attribute INIT_0E: string;
attribute INIT_0F: string;

attribute INIT_00 of Instram0 : label is
"01C0000000000640000000C000000000104000000000000000000004000000000";
attribute INIT_01 of Instram0 : label is
"0000000001400000000000000000000007C007C002400000001400000007400000";
attribute INIT_02 of Instram0 : label is
"000000000000000014000000000000000AC000000640000000000000000000640";
attribute INIT_03 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_06 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram1 : label is
"0000000000000000800000000000000000000000810001000100010000000000";
attribute INIT_01 of Instram1 : label is
"0000800001000000000000000000000000000000800000800100010000000000008000";
attribute INIT_02 of Instram1 : label is
"0000000000000000000000000000000000000000000000008000010000000000";
attribute INIT_03 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram2 : label is
"00A0000000000000004100600000000000000004100010060180000000000000";
```

```
attribute INIT_01 of Instram2 : label is
"0082000200000000000000000000000000020020000000020000000000000082";
attribute INIT_02 of Instram2 : label is
"0000000000000000000200000000000000010000000000410001000100000000";
attribute INIT_03 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram3 : label is
"5400000000008000780054000000000084007C001C051C0524A4208000000400";
attribute INIT_01 of Instram3 : label is
"7C001C045480000000000800880000007000000554001CA05480000080007800";
attribute INIT_02 of Instram3 : label is
"000000008000C005800000000000140006C00000084007C001C0630C000008400";
attribute INIT_03 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_04 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_0C of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

begin

Instram0: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"01C0000000000640000000C000000000104000000000000000004000000000000",
INIT_01 => X"000000000140000000000000000007C007C00240000000000140000007400000",
INIT_02 => X"000000000000000001400000000000000AC00000064000000000000000000640",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(15 downto 0), DO=>inst_out(15 downto 0), EN=>en,
RST=>rst, WE=>we);

Instram1: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"0000000000000000800000000000000000000000810001000100010000000000",
INIT_01 => X"0000800001000000000000000000000000008000008001000100000000008000",
INIT_02 => X"0000000000000000000000000000000000000000000000008000010000000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(31 downto 16), DO=>inst_out(31 downto 16), EN=>en,
RST=>rst, WE=>we);
```

```
Instram2: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"00A0000000000000000041006000000000000004100010060180000000000000",
INIT_01 => X"0082000200000000000000000000000000002002000000020000000000000082",
INIT_02 => X"0000000000000000000020000000000000001000000000410001000100000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(47 downto 32), DO=>inst_out(47 downto 32), EN=>en,
RST=>rst, WE=>we);

Instram3: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"5400000000008000780054000000000084007C001C051C0524A4208000000400",
INIT_01 => X"7C001C04548000000000800880000007000000554001CA054800000800007800",
INIT_02 => X"0000000008000C0058000000000014006C00000084007C001C0630C000008400",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(63 downto 48), DO=>inst_out(63 downto 48), EN=>en,
RST=>rst, WE=>we);
end architecture behavioural;

--Instruction Memory for 'RCHLD' Macro Instruction

library IEEE;
use IEEE.std_logic_1164.all;

--synopsys translate_off;
library unisim;
use unisim.vcomponents.all;
```

```
--synopsys translate_on;
entity INSTMEM is
port(clk, we, en, rst: in std_logic;
   addr: in std_logic_vector(7 downto 0);
   inst_in: in std_logic_vector(63 downto 0);
   inst_out: out std_logic_vector(63 downto 0));
end entity INSTMEM;

architecture behavioural of INSTMEM is

component RAMB4_S16 is
port(ADDR: in std_logic_vector(7 downto 0);
   CLK: in std_logic;
   DI: in std_logic_vector(15 downto 0);
   DO: out std_logic_vector(15 downto 0);
   EN, RST, WE: in std_logic);
end component RAMB4_S16;

attribute INIT_00: string;
attribute INIT_01: string;
attribute INIT_02: string;
attribute INIT_03: string;
attribute INIT_04: string;
attribute INIT_05: string;
attribute INIT_06: string;
attribute INIT_07: string;
attribute INIT_08: string;
attribute INIT_09: string;
attribute INIT_0A: string;
attribute INIT_0B: string;
attribute INIT_0C: string;
attribute INIT_0D: string;
attribute INIT_0E: string;
attribute INIT_0F: string;

attribute INIT_00 of Instram0 : label is
"000001C000000C00000000C00000000010400000000000000000004000000000";
attribute INIT_01 of Instram0 : label is
"0000024000000A4000000000000000008C000000014000000A40000000000000";
attribute INIT_02 of Instram0 : label is
"00000000000001C00000000000000000078000000A4000000000000000000B00";
attribute INIT_03 of Instram0 : label is
"000000000000000000000000000000000000000000000540000000A40000000000";
attribute INIT_04 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_0A of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram0 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram1 : label is
"0100010000000000800000000000000000000000810001000100010000000000";
attribute INIT_01 of Instram1 : label is
"0100010000000000000080000100000000008000000000000000000080000100";
attribute INIT_02 of Instram1 : label is
"0000000000000000000000000000000000000000000000008000000000000000";
attribute INIT_03 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000008000";
attribute INIT_04 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram1 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram2 : label is
"00020000000000008200A0000000000000000000410001006018000000000000";
attribute INIT_01 of Instram2 : label is
"0001000000000000410001000100000000004100600000000000008200024000";
attribute INIT_02 of Instram2 : label is
"0000000000000000000000000000000000000000000041000100000000002800";
attribute INIT_03 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000820002";
attribute INIT_04 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
attribute INIT_05 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram2 : label is
"0000000000000000000000000000000000000000000000000000000000000000";

attribute INIT_00 of Instram3 : label is
"1CC0550000008000780054000000000084007C001C051C0524A4208000000400";
attribute INIT_01 of Instram3 : label is
"1CA05480000084007C001C042C800000800078005400000084007C001C0734E6";
attribute INIT_02 of Instram3 : label is
"000008000C00580300000800880000007000000084007C001C00000014006404";
attribute INIT_03 of Instram3 : label is
"000000000000000000000000000000000000000000007000000084007C001C00";
attribute INIT_04 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_07 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_08 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_09 of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0A of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0B of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0C of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0D of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0E of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_0F of Instram3 : label is
"0000000000000000000000000000000000000000000000000000000000000000";
```

```
begin

Instram0: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"000001C000000C00000000C00000000010400000000000000000000400000000",
INIT_01 => X"0000024000000A40000000000000000008C00000014000000A40000000000000",
INIT_02 => X"00000000000001C00000000000000000078000000A4000000000000000000B00",
INIT_03 => X"0000000000000000000000000000000000000000000005400000A40000000000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(15 downto 0), DO=>inst_out(15 downto 0), EN=>en,
RST=>rst, WE=>we);

Instram1: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"0100010000000000080000000000000000000000081000100010001000000000",
INIT_01 => X"0100010000000000000080000100000000000800000000000000000080000100",
INIT_02 => X"0000000000000000000000000000000000000000000000008000000000000000",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000008000",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(31 downto 16), DO=>inst_out(31 downto 16), EN=>en,
RST=>rst, WE=>we);

Instram2: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"000200000000000008200A00000000000000000041000100601800000000000",
INIT_01 => X"0001000000000000410001000100000000004100600000000000008200024000",
INIT_02 => X"0000000000000000000000000000000000000000000000410001000000002800",
INIT_03 => X"0000000000000000000000000000000000000000000000000000000000820002",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(47 downto 32), DO=>inst_out(47 downto 32), EN=>en,
RST=>rst, WE=>we);

Instram3: RAMB4_S16
--synopsys translate_off
GENERIC MAP (
INIT_00 => X"1CC05500000008000780054000000000000084007C001C051C0524A4208000000400",
INIT_01 => X"1CA05480000084007C001C042C800000800078005400000084007C001C0734E6",
INIT_02 => X"000008000C005803000008008800000007000000084007C001C00000014006404",
INIT_03 => X"00000000000000000000000000000000000000070000000084007C001C00",
INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000")
--synopsys translate_on
port map(ADDR=>addr, CLK=>clk, DI=>inst_in(63 downto 48), DO=>inst_out(63 downto 48), EN=>en,
RST=>rst, WE=>we);
end architecture behavioural;
```

References

1. T. T. Speakman, D. Farinacci, S. Lin, and A. Tweedly. The PGM Reliable Transport Protocol, August 1998.RFC (draft-speakman-pgm-spec-02.txt).

2. H. W. Holbrook and D. R. Cheriton, IP Multicast Channels: EXPRESS Support for Large-Scale Single Source Applications. In *Proceedings of SIGCOMM'99, 1999.*

3. D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse, ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols, 1998.

4. M. Hicks, P. Kakkar, T. Moore, C. A. Gunter, and S. Nettles, PLAN: A Packet Language for Active Networks. 1998. International Conference on Functional Programming.

5. J. T. Moore, M. Hicks, and S. Nettles, Practical Programmable Packets. In *IEEE INFOCOM,* Anchorage, AK, April 2001.

6. S. Wen, J. Griffioen, and K. Calvert, Building Multicast Services from Unicast Forwarding and Ephemeral State, In *Proceedings of 2001 Open Architectures and Network Programming Workshop,* Anchorage, AK, April 27-28, 2001.

7. S. Wen, J. Griffioen, and K. Calvert, CALM: Congestion-Aware Layered Multicast, In *Proceedings of 2002 Open Architectures and Network Programming Workshop,* New York, NY, June, 2002

8. Kenneth L. Calvert, James Griffioen and Su Wen. Lightweight Network Support for Scalable End-to-End Services. In Proceedings of SIGCOMM 2002, Pittsburg, PA, August 19-23, 2002.

9. Burton Bloom, Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM,* 13(7):422-426, July 1970.

10. S. Pingali, D. Towsley, and J. Kurose, A Comparison of Sender-initiated and Receiver-initiated Reliable Multicast Protocols. In *Proceedings of the ACM SIGMETRICS'94 Conference,* Pages 221-230, 1994

11. I. Stoica, T. S. Eugene Ng, and H. Zhang. REUNITE: A Recursive Unicast Approach to Multicast. In *Proceedings of INFOCOM 2000,* 2000.

12. S. Savage, D. Wetherall, A. Karlin, and T. Anderson, Practical Network Support for IP Traceback. In *ACM SIGCOMM,* Stockholm, Sweden, August, 2000.

13. A. C. Snoren, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-Based IP Traceback. In *ACM SIGCOMM,* San Diego, CA, August, 2001.

14. K. Park and H. Lee, On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *ACM SIGCOMM*, San Diego, CA, August 2001.

15. K. Calvert, J. Griffioen, and S. Wen. Concast: Design and Implementation of a New Network Service. In *Proceedings of 1999 International Conference on Network Protocols,* Toronto, Ontario, November, 1999.

16. B. Schwartz, A. Jackson, W. Strayer, W. Zhou, R. Rockwell, and C. Partridge. Smart Packets for Active Networks. In *1999 IEEE Second Conference on Open Architectures and Network Programming,* Pages 90-97, March, 1999.

17. http://www.xilinx.com

18. T. Halfhill, "Intel Network Processor Targets Routers: IXP1200 Integrates Seven Cores for Multithreaded Packet Routing", *MICROPROCESSOR REPORT,* Vol. 13, No. 12, Sept. 13, 1999.

19. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, " Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Transactions on VLSI Systems*, Vol. 4, Issue 1, pp.56-69 , March, 1996 .

20. S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-638, April, 1998.

21. K. Bondalapati and V. K. Prasanna, "Reconfigurable Computing: Architectures, Models and Algorithms", *CURRENT SCIENCE: Special Section on Computational Science*, Vol. 78, No. 7, pp.828-837, April, 2000.

22. K. Compton, S. Hauck, "Reconfigurable Computing: Survey of Systems and Software", *ACM Computing Surveys (CSUR)*, Vol. 34, Issue 2, pp.171-210, June, 2002.

23. J.R. Heath, S. Ramamoorthy, C.E. Stroud, and A. Hurt, "Modeling, Design, and Performance Analysis of a Parallel Hybrid Data/Command Driven Architecture System and its Scalable Dynamic Load Balancing Circuit", *IEEE Trans. on Circuits and Systems, II: Analog and Digital Signal Processing*, Vol. 44, No. 1, pp. 22-40, January, 1997.

24. J.R. Heath and B. Sivanesa, "Development, Analysis, and Verification of a Parallel Hybrid Data-flow Computer Architectural Framework and Associated Load Balancing Strategies and Algorithms via Parallel Simulation", *SIMULATION*, Vol. 69, No. 1, pp. 7-25, July, 1997.

25. J.R. Heath and A. Tan, "Modeling, Design, Virtual and Physical Prototyping, Testing, and Verification of a Multifunctional Processor Queue for a Single-Chip Multiprocessor Architecture", *Proceedings of 2001 IEEE International Workshop on Rapid Systems Prototyping*, Monterey, California, 6 pps. June 25-27, 2001.

26. Su Wen, "Supporting Group Communication on a Lightweight Programmable Network", Ph.D. Thesis, Department of Computer Science, University of Kentucky, May, 2003.

27. John L. Hennessy and David A. Patterson, *Computer Organization and Design – The Hardware / Software Interface,* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

28. http://www.engr.uky.edu/~heath/M_Muthulakshmi_Thesis_ESPR.V1_VHDL_Code.pdf