1-1-2007

A Multi-Dimensional Logarithmic Number System based central processing unit.

Mahzad Azarmehr
University of Windsor

A Multi-Dimensional Logarithmic Number System based Central Processing Unit

by

Mahzad Azarmehr

A Thesis
Submitted to the Faculty of Graduate Studies and Research through
Electrical and Computer Engineering in Partial Fulfillment
of the Requirements for the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
2007

Your file Votre référence
ISBN: 978-0-494-42322-6
Our file Notre référence
ISBN: 978-0-494-42322-6

Abstract

The Multi-Dimensional Logarithmic Number System (MDLNS), provides a reduction in the size of the number representation and promises a lower cost realization of arithmetic operations. The orthogonal nature of the parallel base computations and the multi-digit extensions of the MDLNS representations reduce the complexity of computations. The reduced hardware complexity, simplified arithmetic operations, and the non-linear nature of the representation makes MDLNS suitable for some DSP applications.

The work presented in this thesis is the design and implementation of a 2DLNS based CPU. This CPU, in addition to traditional arithmetic operations, is able to perform some special 2DLNS based operations. The CPU takes advantage of a relatively simple architecture and a well designed organization which greatly simplifies the implementation of many DSP algorithms.
An assembly program is also written to implement a 2DLNS based filterbank architecture. This implementation demonstrates the efficiency and efficacy of 2DLNS CPU in real applications.

To whom made me believe that where there is a will, there is a way.

Acknowledgments

There are several people who deserve to be acknowledged for their generous contributions to this project. I would first like to express my sincere gratitude and appreciation to Dr. Roberto Muscedere, my supervisor, for his invaluable guidance throughout the course of this thesis work. Special thanks to Dr. Majid Ahmadi and Dr. Maher Sid-Ahmed for their expert guidance and constant support throughout my study. I would also like to thank Dr. Angela Sodan for reviewing this work.

I also sincerely appreciate my family for their endless support and my friends, Josee Jarry, Ashkan Hosseinzadeh, Mitra Mirhassani, Kevin Banovic, Amirali Yazdanshenas, Elham Shahinfard and Karl Leboeuf, for their help and friendship.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
EPROM	Erasable Programmable Read Only Memory
FIR	Finite Impulse Response
FSM	Finite State Machine
I/O	Input/Output
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IFIR	Interpolated Finite Impulse Response
ISA	Instruction Set Architecture
LNS	Logarithmic Number System
LUT	Look Up Table
MAC	Multiply and Accumulate
RALUT	Range Addressable Look Up Table
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTL	Register Transfer Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Introduction

Integrated Circuits (ICs), after only a half century of their initiation, are consistent parts of all microelectronic devices. ICs which are consisting of many interconnected transistors in a circuitry, are packed on chips. In accordance to *Moore's* law, during last decades the number of transistors per unit area of chips has been doubled every 18 months, which has decreased cost and/or increased functionality. Among the most advanced ICs are microprocessors, which in a variety of complexity, are dominant controllers of all digital appliances. Microprocessors are an example of Very Large Scale Integration (VLSI) devices. VLSI is the process of creating ICs by combining thousands of transistor-based circuits into a single chip.

The improvements in IC process technology has led to tremendous growth in the Digital Signal Processing (DSP) field. Today, DSP has permeated into almost every aspect of science and engineering.

The algorithms required for DSP are sometimes performed using specialized computers, which make use of specialized microprocessors. Digital signal processors are generally purpose-designed, Application Specific Integrated Circuits (ASICs), and process signals in real time.

The demands of low power consumption and small size processing have led to a number of advances in algorithms, semiconductor technologies and architectures of DSP systems.

Special purpose high performance DSP systems often take advantage of the properties of special number representations. The Logarithmic Number System (LNS) has been considered as a major alternative to the binary representation [10], [11], [18], [19], as it simplifies the difficult multiplication, division and exponentiation operations. It has been recognized that LNS architectures are perfectly suited for low-power, low-precision DSP problems. The major drawback of the LNS is the need to use very large ROM arrays in implementing addition and subtraction [6].

The Multi-Dimensional Logarithmic Number System (MDLNS), which has similar properties to the classical LNS, provides more degrees of freedom by the virtue of having multiple orthogonal bases, and the ability to gain from the use of multiple digits. The MDLNS has found initial applications in the implementation of special digital signal processing systems, where the parallel operations on independent bases greatly reduces both the hardware and the connectivity of the architecture [16].

In order to ease the implementation of MDLNS applications, and speed up any design and simulation process, the idea of having a CPU based on MDLNS has been raised. In such a case, every application is a microprogram running on this CPU. In order to maintain simplicity in our design, 2DLNS representations and operations have been considered, which it means just two bases are used. Since having two orthogonal bases is sufficient to provide desired precision in most applications, in this research work a 2DLNS based CPU has been designed and implemented. The design

concepts and algorithms are the same for dimensions more than two. Therefore, this work is scalable for any other dimension. The reasonably small number of instructions, limited instruction types, and simple instruction architecture, provides the CPU with a simple assembly language, and makes it applicable to other research work, as well as the realization of DSP algorithms.

1.2 Thesis Objectives

The work presented in this thesis conforms to the following objectives:

1. Develop a processor (CPU) with 2DLNS capabilities
2. Demonstrate the efficiency of the CPU by programming and implementing a filterbank application

1.3 Thesis Organization

This thesis is organized into five chapters and one appendix. Chapter 2 provides background material on MDLNS by covering the MDLNS representation, its properties and arithmetic, and conversions associated with it. Chapter 3 includes the details of a 2DLNS based CPU design, which is a Reduced Instruction Set Computer (RISC). This chapter contains the design flow of the CPU architecture and its organization. First of all, the Instruction Set Architecture and its operations are specified, the CPU interfaces are determined, and then the CPU organization, consisting of operational components, is designed. The functional behavior of these components is described at the Register Transfer Language (RTL) level. Finally, the CPU's test bench, and its ancillary HDL programs are explained. In this regard, the CPU's controller is described in detail. Chapter 4 explores a previous filterbank custom design and describes a microprogram which runs on the 2DLNS CPU. The chapter continues with

the simulation of the VHDL modules, and the final results are illustrated. Chapter 5 concludes this thesis and provides recommendations for future work. The HDL codes of all CPU components and auxiliary packages are attached in Appendix A.

Chapter 2

Multi-Dimensional Logarithmic Number System

2.1 Introduction

In the area of Digital Signal Processing (DSP) an increasing demand exists for compact, high speed, real time, and low power digital processing systems. DSP systems manipulate signals as a sequence of numbers, and usually require massive arithmetic computations to perform algorithmic processing such as modulation or filtering.

On the other hand, multipliers are fundamental units in most DSP applications, such as FIR filtering and in the Discrete Fourier Transform (DFT), and are also the most hardware consuming components. Multiplication is usually implemented in Multiply and Accumulate (MAC) units. Hence, specialized DSP hardwares heavily rely on optimized MAC operations.

In recent signal processing research, the use of special coding schemes or new num-

ber systems in the design of algorithms have been considered. Most new design approaches have been directed to provide a greater degree of modularity and parallelism in comparison with traditional algorithms. One such approach is a Multi-Dimensional Logarithmic Number System (MDLNS), whose arithmetic is broken up into independent sub-terms, providing a modular hardware implementation [8].

The MDLNS has a number of properties that can be advantageous to DSP applications, such as reduced hardware complexity for a DSP application mostly reliant on multiplication. The non-linear number representation of MDLNS which is mostly an error free mapping, may benefit certain special applications while the logarithmic-like representation of numbers promises a realization improvement on a non-uniform quantization mapping of data [12].

Since the MDLNS was introduced [7] in 1996, it has been increasingly used in some DSP and cryptography applications [4], [5], [3].

2.2 Representation

A representation of the real number, X, in the form:

$$X = \sum_{i=1}^{n} s_i \prod_{j=1}^{b} p_j^{e_j{}^{(i)}}$$

where $s_i \in \{-1, 0, 1\}$ and p_j, $e_j{}^{(i)}$ are integers, is called a multi-dimentional n-digit logarithmic (MDLNS) representation of X, where b is the number of bases used (at least two), the first one, that is p_1, will always be assumed to be 2 [13].

The logarithmic properties of the MDLNS allow for a reduced-complexity multiplication, and a larger dynamic range. On the other hand, error-free representations are special cases of the MDLNS, but the extra degree of freedom provided by the use of multiple digits can mitigate the non-uniform quantization properties of other representations [6]. The MDLNS orthogonal bases, and ability to gain from the use

of multiple digits, reduces both the hardware and the connectivity of the architecture [16].

MDLNS is a redundant number system, it is provable that every real value has a MDLNS representation, but realistically the majority will have at most 4 or 5 error-free representations. This redundancy can be useful in order to choose the best possible representation for each application. Particularly, the redundant values of 1 in MDLNS can be used as a coefficient to decrease the values of exponents, which is an easy way to prevent overflow in calculations.

2.3 Mathematical Operation

A 2DLNS representation provides a triple, $\{s_i, a_i, b_i\}$, for each digit, where s_i is the sign bit and a_i, b_i are the exponents of the binary and non-binary bases. Usually, the second base is shown with D. D is a suitably chosen real number (not necessarily an integer but not a multiple of 2). Thus a number, x, can be represented as given in:

$$x = \sum_{i=1}^{n} s_i.2^{a_i}.D^{b_i}$$

where $b_i = \{-2^{R-1}, ..., 2^R - 1\}$ and R is the number of bits needed to represent b in binary. The number of required bits to represent the second base exponent is usually shown by B. Although the range of a is self limiting, there is also some restriction based on B to specify the range of a in binary, which is $a_i = \{-2^{B-1}, ..., 2^B - 1\}$[13].

MDLNS multiplication and division are the simplest arithmetic operations. The corresponding equations, given a single-digit 2DLNS representation of $x = \{s_x, a_x, b_x\}$ and $y = \{s_y, a_y, b_y\}$, are [5]:

$$x.y = \{s_x.s_y, a_x + a_y, b_x + b_y\}$$

$$x/y = \{s_x.s_y, a_x - a_y, b_x - b_y\}$$

These equations show that single-digit 2DLNS multiplication / division can be implemented in hardware using two independent binary adders / subtracters and simple logic for the sign correction.

Unfortunately, addition and subtraction operations are not as simple as multiplication and division operations. They must be handled through a set of identities and look-up tables. The identities are:

$$\begin{aligned} 2^{a_x}.D^{b_x} + 2^{a_y}.D^{b_y} &= (2^{a_x}.D^{b_x}).(1 + 2^{a_y-a_x}.D^{b_y-b_x}) \\ &\simeq (2^{a_x}.D^{b_x}).\Phi(a_y - a_x, b_y - b_x) \end{aligned}$$

$$\begin{aligned} 2^{a_x}.D^{b_x} - 2^{a_y}.D^{b_y} &= (2^{a_x}.D^{b_x}).(1 - 2^{a_y-a_x}.D^{b_y-b_x}) \\ &\simeq (2^{a_x}.D^{b_x}).\Psi(a_y - a_x, b_y - b_x) \end{aligned}$$

The operators Φ and Ψ are look-up tables that store the precomputed 2DLNS values. Since the size of these tables may be very large, it is more practical to convert the 2DLNS numbers to binary, and perform the addition and subtraction using binary representation.

Multi-digit MDLNS arithmetic is simply an extension of the single-digit MDLNS arithmetic. In this case, each digit can be treated as an independent MDLNS number and the operations handled separately [13].

2.4 Conversion

Since the MDLNS was first introduced and considered for use in DSP applications, a method for converting data between binary and MDLNS representations was needed.

On the other hand, it was preferable to execute addition in binary for every application. Therefore, these converters should be considered as a requirement in almost every MDLNS application. Since there is no functional relationship between standard binary representation and MDLNS representation, the early methods proposed for binary to MDLNS conversion used simple look-up tables (LUTs) [5].
In order to explain how these tables work, a single-digit 2DLNS number is considered:

$$X = s.2^a.D^b$$

To find the equivalent binary representation, b is used as an index address to a LUT to find a floating-point representation for D^b:

$$D^b = \mu(b).2^{\epsilon(b)}$$

Here, $\mu(b)$ is the mantissa ($1 \leq \mu(b) < 2$) and $\epsilon(b)$ is the exponent (integer). The final floating-point representation of X is:

$$X = s.\mu(b).2^{(a+\epsilon(b))}$$

For the reverse conversion, the input to the LUT is the mantissa, $\mu(b)$, and the outputs are b and and $\epsilon(b)$. Since the mantissa is not influenced by the exponent, this exponent can remain as an output. The conversion of $|X|$ to a mantissa is easily achieved in hardware with a conditional feedback bit-shifter and counter, or a priority encoder [13]. In either case, the number of shifts performed, *shifts*, is used to generate the binary exponent:

$$a = shifts - \epsilon(mantissa)$$

Although a LUT offers a simple and fast binary to MDLNS conversion algorithm, the implementation of these LUTs can become very large and unrealistic as their size is exponentially dependent on the input binary dynamic range [8]. The LUT sizes further depend on the number of digits and bases in the MDLNS representation. Therefore, some other hardware realizable techniques for conversion have been

developed [13]. All these techniques use a special memory device named Range Addressable Look-up Tables (RALUTs).

In this approach, the address decode system is changed from exact matching to range matching. In a standard LUT architecture, shown in Fig. 2.1, an address decoder is used to match the address to a unique stored value.

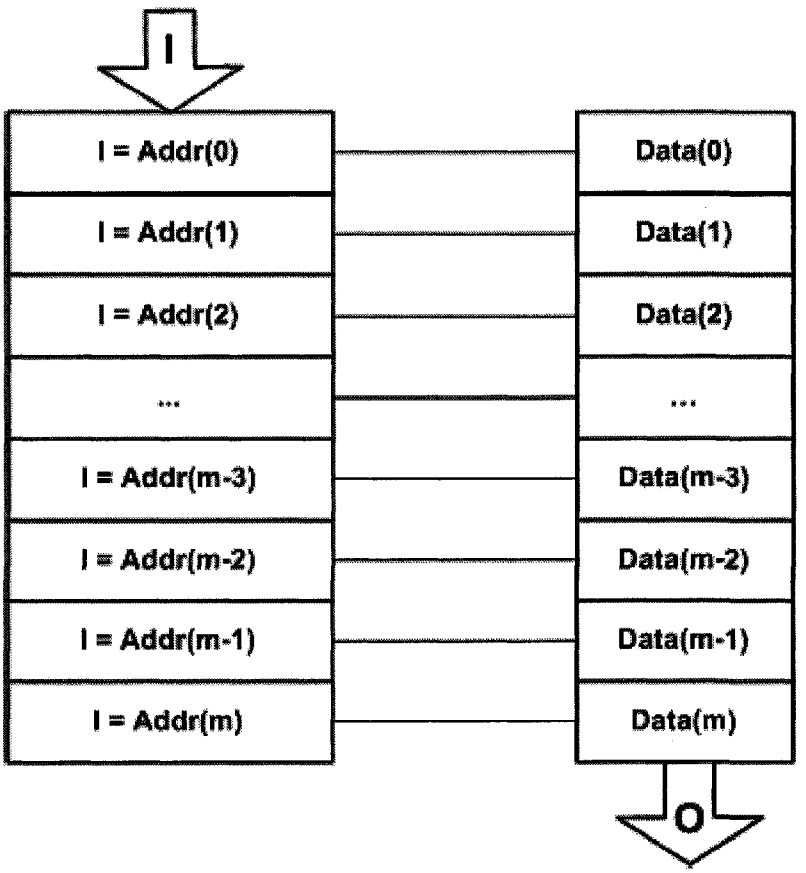

Figure 2.1: Standard LUT Structure

The RALUT architecture of Fig. 2.2, shows the new address decoder system that matches a stored value to a range of addresses.

The decoder compares the input address, **I**, to a range of two neighboring monotone addresses (e.g., **Addr(1)** and **Addr(2)**). Only one of these comparisons will match the input and activate a word-enable line, which drives the data patterns,

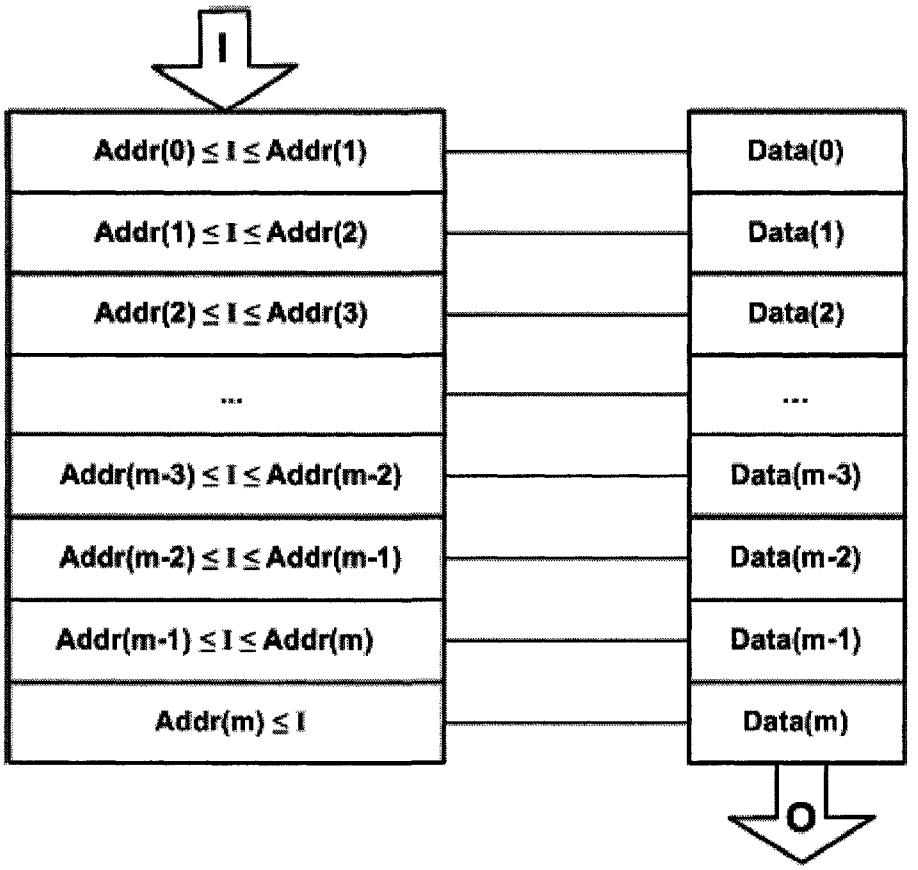

Figure 2.2: RALUT Structure

Data, to the output, **O**, of the RALUT. Half of the comparators in the range decoder can be removed:

$$(I < Addr(n)) = \overline{(Addr(n) \leq I)}$$

$$(I \geq Addr(n)) \oplus (I \geq Addr(n+1)) = Addr(n) \leq I < Addr(n+1)$$

Since the address has a monotone nature, if ($I \geq Addr(n+1)$) is true, then ($I \geq Addr(n)$) must also be true. Therefore, the XOR operator can be reduced as:

$$(I \geq Addr(n)).\overline{(I \geq Addr(n+1))} = Addr(n) \leq I < Addr(n+1)$$

The RALUT architecture is optimal since it only requires $2^R + 1$ rows [13].

Conversion from a two-digit 2DLNS to binary is a fairly simple process. Both 2DLNS digits are converted separately using the single-digit method, and their results are accumulated to produce the final binary representation. For the reverse conversion four methods have been developed [16].

1. The *Quick method* chooses the first-digit nearest to the target, and generates the second-digit to reduce the error, a simple greedy algorithm.

2. The *High/Low method* chooses the two nearest approximations to the target as the first-digits, generates two associated second-digits for the error, and selects the combination with the smaller error.

3. The *Brute-Force method* operates by selecting the combination with the smallest error, but it uses all possible mantissa of D^b as the first-digits instead of just one (*Quick*) or two (*High/Low*).

4. The *Extended-Brute-Force method* improved upon the *Brute-Force method* by using first-digit approximations above 2.0 and below 1.0 (shifted left or right L bits).

Each method ranges from simple implementations and fairly accurate approximations to difficult implementations and very accurate approximations. All of these methods have been implemented in fully parametrized Verilog HDL code, which can be found in [13].

The TLNS (as we name this CPU) CPU design makes use of the two-digit 2DLNS *High/Low* serial converter. Since the filterbank is intended for speech processing and low power operation, a serial implementation was selected to minimize both power and area.

Chapter 3

TLNS CPU Design

3.1 Overview of TLNS CPU Design

The first step in designing a CPU is to determine its applications. The 2DLNS CPU which we name TLNS hereafter, in addition to performing most traditional arithmetic and logical operations, should also be able to perform some particular tasks. These special tasks include 2DLNS / Binary conversions, 2DLNS multiplication and multiply and accumulation, as well as some other operations. Therefore this CPU is not an application specific design, and can be considered a relatively simple Reduced Instruction Set Computer (RISC) architecture. The original RISC design of *Hennessy and Patterson* [9] is considered as a conceptual basis in architecture design of this CPU. The Instruction Set Architecture (ISA) and CPU basic architecture are modified in order to achieve compatibility with design requirements [1]. We start by developing the processor's ISA based on its application. Then the state diagram of this CPU is designed. All micro-operations performed during each state, and the

conditions that cause the CPU to go from one state to another are shown. In the next step, micro-operations should be developed to fetch, decode, and execute each instruction. Then we determine the necessary components that need to be included within the CPU. Once this is done, we define the internal data paths and necessary control signals. Finally, we design the control unit, the logic that generates the control signals and causes the operations to occur. This is the entire design process to determine the CPU organization.

3.2 TLNS Instruction Set Architecture (ISA)

The Instruction Set Architecture (ISA) is essentially the microprocessor interface. A microprocessor's ISA includes information needed to interact with the microprocessor. The Instruction Set is the set of all assembly language instructions that the microprocessor can execute. In addition, the details of the programmer accessible registers within the microprocessor are also included in the ISA. These registers store and perform operations on data. The ISA must specify these registers, their sizes, and the instructions in the Instruction Set that can use each register. The ISA also includes information necessary to interact with memory [2]. Based on the conducted research on [15], considering ($B = 6$) for binary base exponent and ($R = 5$) for second base exponent and an optimal second base of $D = 0.92024380912663017$, in order to convert a 16-bit signed binary data to its equivalent 2DLNS representation, 19513 representations of 32768 possible representations, will be error-free (59.5% with $\epsilon < 0.5$) representations. The remaining 13255 representations have errors from 0.5 to 6. Therefore, in a 1-bit sign 2DLNS representation, considering 24 bits provides a reasonable mapping precision. In order to keep the TLNS CPU architecture consistent, instruction length, register size, data buses, and memory words are all designed in 24 bits. This section introduces the attributes of assembly language instructions

as well as processor register set specifications.

3.2.1 Registers

Registers have a large effect on the performance of a CPU. The CPU can retrieve data from its register set much more quickly than from memory. Having too few registers causes a program to refer to memory more often which reduces performance. TLNS has 16 general-purpose registers, as well as some special-purpose registers. Registers r0 to r15 are general-purpose registers that may be used to hold any 24-bit value, including data and instruction. Register r0 is special in that it always has the value 0. Any value written into this register is discarded. Register r14 is used to keep the equivalent 2DLNS representation for value 1. This value is necessary for using the MAC unit to perform 2DLNS / Binary conversion. In a 24-bit representation with two's-complement representations for both indexes, its Hexadecimal value is 414414. This value should be preloaded to data memory and written into r14 by the first instruction of every program. Register r15 also has a special application in every "link" instruction, as we will see later.

The remaining registers have special purposes and are not used to store data. The program counter (PC) holds the memory address of the next instruction to be read into the CPU. As mentioned above, each TLNS instruction and memory word is represented in one 24-bit word. Hence, the PC value is incremented by one after each instruction is fetched. The memory address register (MAR) is used in order to specify the next address in data memory. There are two other registers named A and B which will be discussed later as parts of CPU register file.

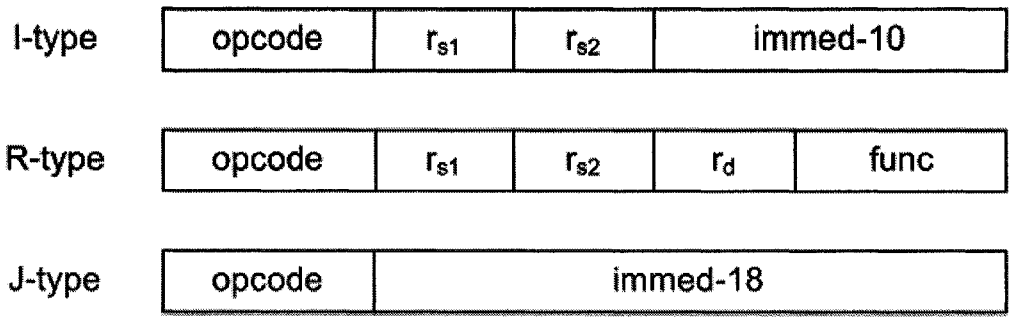

Figure 3.1: The three instruction types in TLNS

3.2.2 Instruction Types

As mentioned earlier, each TLNS instruction is encoded in a 24-bit word. There are three instruction formats, I-type, R-type and J-type, shown in Fig. 3.1. I-type format is generally used for arithmetic and logical instructions that have an immediate operand, and for branch instructions. R-type format is used for arithmetic and logical instructions that operate entirely on register contents. J-type instructions are used for unconditional jump instructions, allowing for a larger displacement.

3.2.3 Instruction Set

The TLNS includes instructions for transferring data to and from memory, for performing arithmetic and logical operations, for transferring control within a program, and some special instructions based on 2DLNS. These groups of instructions are described in the following sections.

It is also helpful to declare a package that represents the details of the TLNS instruction set. This package, which is named **tlns_instr**, also includes specified encodings for opcodes. The package first declares bit-vector types for the fields in an instruction. These are followed by some constants defining the opcode values for each

of the instructions in the TLNS instruction set. The opcode **op_special** represents a class of instructions that use R-type format. In this case, the **func** field is used to specify the instruction. Next, the type **reg_index** represents numeric values for register numbers. The constants **output_reg_1** and **output_reg_2** specify registers r12 and r13 as output registers for the filter instruction. The r14 register always contains the 2DLNS representation of 1. The constant **link_reg** is the number of the general-purpose register used in "jal" and "jalr" instructions.

Data Transfer Instructions

The TLNS instructions for transferring data are listed in Table 3.1. These instructions transfer data between the CPU and data memory. The memory address for both load and store instructions is determined by adding the immediate value to the contents of a base register r_{s1}, and r_d is the register whose content is loaded from memory or stored into memory.

Instruction	Operands	Descriptions
lw	r_{s1}, r_d, immed-10	Load word
sw	r_{s1}, r_d, immed-10	Store word

Table 3.1: TLNS Data Transfer Instructions

Arithmetic and Logical Instructions

Table 3.2 shows the TLNS instructions for performing arithmetic operations on binary integer data. Integers are either unsigned or two's-complement signed values. Instructions are included to add and subtract operands in source registers r_{s1} and

r_{s2}, with the result overwriting the destination register r_d. There are also immediate forms, in which the second operand is an immediate value. The next two groups of instructions shown in this table are relational instructions, which compare their source operands. The conditions that can be tested are "eq" (equal to), "ne" (not equal to), "lt" (less than), "le" (less than or equal to), "gt" (greater than) and "ge" (greater than or equal to). If the condition is met, the destination register is set to the integer 1, otherwise it is set to zero [1]. The "lhi" instruction is used to load a 10-bit immediate value into the most significant 10 bits of the destination register, clearing the rest 14 bits to zero. The "nop" instruction, as its name suggests, performs no operation.

The next group of rows in table 3.2 show the logical operations. The "and" , "or" and "xor" instructions perform the logical operation on corresponding bits from each of the operands to generate the 24 bits of the result. The immediate versions of these instructions extend the 10-bit immediate operand to 24 bits by adding zeros to the left, in order to perform the bitwise logical operation. The first group of shift instructions shift the value read from r_{s1} by the number of bits specified in r_{s2}, and store the results in r_d. The second group shifts the value by the number of bits specified by the immediate operand.

Control Transfer Instructions

The TLNS instructions to handle transfer of control within a program are listed in Table 3.3. The branch instruction transfer control to the data memory address calculated by adding a displacement to the PC. The "beqz" and "bnez" instructions compare a register r_{s1} with zero and branch if the condition is met. The next four instructions unconditionally transfer control. The "j" and "jal" instructions add the

Instruction	Operands	Descriptions
add, addu	r_{s1}, r_{s2}, r_d	Add signed or unsigned
sub, subu	r_{s1}, r_{s2}, r_d	Subtract signed or unsigned
addi, addui	r_{s1}, r_d, immed-10	Add signed or unsigned immediate
subi, subui	r_{s1}, r_d, immed-10	Subtract signed or unsigned immediate
sxx, sxxu	r_{s1}, r_{s2}, r_d	Set if condition signed or unsigned
sxxi, sxxui	r_{s1}, r_d, immed-10	Set if condition signed or unsigned immediate
lhi	r_d, immed-10	Load high immediate
nop		No operation
and, or, xor	r_{s1}, r_{s2}, r_d	Bitwise logical and, or, exclusive-or
andi, ori, xori	r_{s1}, r_d, immed-10	Bitwise logical and, or, exclusive-or immediate
sll, srl, sra	r_{s1}, r_{s2}, r_d	Shift left-logical, right-logical, right-arithmetic
slli, srli, srai	r_{s1}, r_d, immed-10	Shift left-logical, right-logical, right-arithmetic immediate

Table 3.2: TLNS Arithmetic and Logical Instructions

displacement to the PC to determine the target address. The "jr" and "jalr" instructions, on the other hand, use the contents of a register r_{s1} as the target address. The term "link" in "jal" and "jalr" means that these instructions copy the old value of PC into register r15 before overwriting it with the target address. The last instruction, "halt" is used for terminating the program.

Instruction	Operands	Descriptions
beqz	r_{s1}, immed-10	Branch if register equal to zero
bnez	r_{s1}, immed-10	Branch if register not equal to zero
j	immed-18	Jump unconditional
jal	immed-18	Jump and link unconditional
jr	r_{s1}	Jump register
jalr	r_{s1}	Jump and link register
halt		Halt execution

Table 3.3: TLNS Control Transfer Instructions

Special Instructions

The final set of instructions provided by the TLNS are special instructions. The first two instructions are used to provide data transfer between the CPU and the external world. The "inpt" instruction reads data from the external input register to the destination register r_d. The "oupt" instruction writes the contents of the r_{s1} register to the external output register. The "mult" instruction performs 2DLNS multiplications. The contents of r_{s1} and r_{s2} are operands, and the product is written to r_d. A 2DLNS multiply and accumulate is executed by the "filter" instruction. Since this operation is performed on two sequences of data in instruction and data memories, the start addresses of these sequences should be loaded to a register before the "filter" instruction is executed. This register is addressed by r_{s1}. These addresses are incremented with each iteration of the instruction execution. While the data sequence in instruction memory is supposed to have a fixed range specified by the least significant 7 bits of the immediate value, the range of the data sequence in the data memory is preloaded to register r_d. The accumulated results are written to registers r12 and r13. This instruction will be discussed in more detail in later sections.

Instruction	Operands	Descriptions
inpt	r_d	Read input data to r_d
oupt	r_{s1}	Write output data from r_{s1}
mult	r_{s1}, r_{s2}, r_d	2DLNS multiplication
filter	r_{s1}, r_d, immed-10	FIR filter (Multiply and Accumulate)
tbc	r_{s1}, r_d	2DLNS to binary conversion
btc	r_{s1}, r_d	Binary to 2DLNS conversion

Table 3.4: TLNS Special Instructions

The "tbc" instruction performs 2DLNS to binary conversion. This instruction converts the contents of r_{s1} to 2DLNS representation, and writes the result to r_d. The "btc" instruction performs the reverse conversion. Again, r_{s1} contains the source 2DLNS data, and the converted value is written to r_d.

3.3 TLNS External Interface

The TLNS CPU makes use of several ports. These are shown in Fig. 3.2. The signal **clk** is the master clock signal that drives the CPU. When **reset** changes to '1', the CPU aborts any activity in progress and returns all output signals to their inactive states. When **reset** returns to '0' , the CPU resumes fetching instructions from instruction memory address 0. The CPU uses the **halt** signal to indicate that it has stopped execution. The **ifetch** signal is a status signal that the CPU sets to '1' to distinguish a read to fetch an instruction from a read to fetch data. The **Input_data** port is used to enter data to the CPU, and the **Output_data** port transfers data outside of the CPU. The controller sets the **Output_enable** signal to indicate the contents of the output register outside of the CPU are valid for the next cycle.

The remaining ports of the CPU are its interface with the instruction and data memories. The signals **ir_mem_address(a)** and **mem_a** provide the addresses to access memories. Each address identifies a single word of memory which is equal to three bytes. The instruction memory words can only be read, and it is done via the **ir_mem_read_data(d)** bus. The controller sets the the **ir_mem_enable** signal to access instruction memory. Data memory has separate buses to read data from memory, **mem_d_in**, and to write data into memory, **mem_d_out**. The **mem_en** signal is used to enable the CPU to access data memory, and the signal **mem_write_en** is set by the controller whenever data memory is accessed to be written.

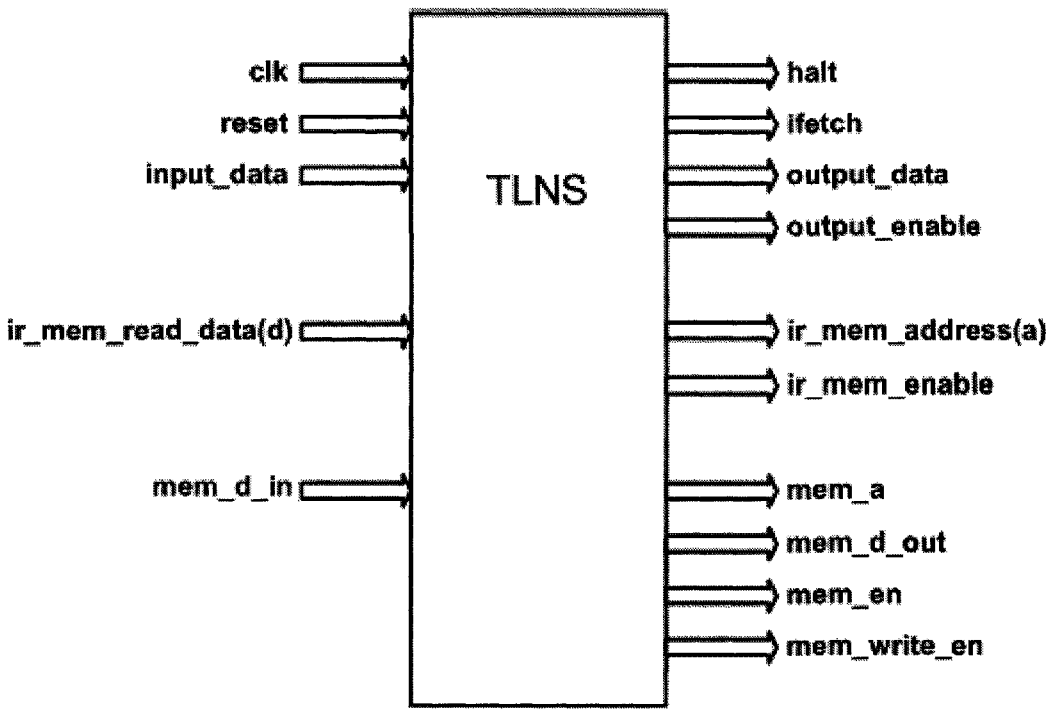

Figure 3.2: The external ports of the TLNS CPU

There are some data types used in the TLNS CPU design. These types are all defined in a package named **tlns_types**. Defined types are based on the predefined

bit and bit-vector types, and on the numeric bit-vector types defined in the IEEE **numeric_bit** package. The package also defines some other types that are used in the design. The type **tlns_address** represents logic vectors used for address values, and the type **tlns_word** represents logic vectors used for data values. **tlns_word_array** represent an array of data words. The type **tlns_bus_word** is a standard-logic vector used for the CPU tristate data bus. The constant **disabled_tlns_word** is the value driven by the CPU or memory onto the data bus when it is inactive.

3.4 TLNS Operation

In general, a CPU performs three sequences of operation in Fetch, Decode, and Execute cycles. In the Fetch cycle, the CPU fetches an instruction from memory, then goes to the Decode cycle. In the Decode cycle, based on the fetched instruction, a corresponding Execute cycle is determined. Finally, in the Execute cycle, the CPU executes the instruction and goes to the Fetch cycle to fetch the next instruction. The controller in the TLNS CPU, is a complex Finite State Machine (FSM) which realizes CPU operations. The controller is organized as a series of procedures. Each procedure includes all necessary micro-operations to perform each individual control task.

Fetching an Instruction

In the TLNS CPU, an instruction fetch is accomplished by performing two procedures. The first procedure, **bus_instruction_fetch_1**, sequences a bus read operation to fetch an instruction from instruction memory. The procedure sets the select control signal of the instruction memory address multiplexer, loading the PC value

onto the external memory address bus. It sets an instruction fetch (ifetch set to '1') and sets **ir_mem_enable** to '1' to start the memory operation. The controller then waits for the successive rising edge of **clk**, and then when the memory operation is done, executes the **bus_instruction_fetch_2** procedure. This procedure disables the instruction memory control signal.

Decoding the Instruction

In order to decode an instruction, two procedures are executed. After an instruction has been fetched, the CPU must increment the program counter. This can be done by the ALU addition function. The current value of the PC is read to **s1_bus** and the controller sends a constant value of 1 to **s2_bus**. These two values are sent to the ALU, and the result should be written to the PC as the next instruction memory address. This procedure, **instruction_decode_1**, also considers r_{s1} and r_{s2} as registers which should be read in the first step of each instruction execution, and sets enable signals for A and B registers, so they are ready to be read. There is an exception for 2DLNS/Binary conversion. In this case, register r14 is considered as register whose contents should be read to register B. Again, after one clock cycle, all control signals are reset through the **instruction_decode_2** procedure. Then, based on the decoded instruction, the execution state is specified.

Executing the Instruction

The most significant part of the controller, is the part that actually executes instructions. Execution starts immediately after the instruction is decoded. The controller uses a case statement to select which group of statements to execute, depending on the instruction opcode. As mentioned above, these statements are organized in pro-

cedures. Later, when the controller is discussed, these procedures will be described in detail.

3.5 TLNS CPU Organization

At the Register Transfer Language (RTL) level, the TLNS CPU is composed of registers, buses, multiplexers, an Arithmetic and Logical Unit (ALU), a Multiply and Accumulate (MAC) unit, a Binary / 2DLNS Converter (BTC), and a sequential control unit. Fig. 3.3 shows the RTL level organization of the CPU upon which we base our VHDL model.

It includes a register file for the 16 general purpose registers. Data is written directly to any of these registers, however the A and B registers are used to store values read from the register file. The input and output registers transfer data to and from the CPU. The Program Counter (PC) and Memory Address Register (MAR) are individual registers which are used to address memories. The two multiplexers allow memory addresses to be determined by the controller as well. The two modules X1 and X2 are extension modules which extend 10-bit or 18-bit immediate values of instructions to 24 bits for processing by ALU, BTC, or MAC. X2 is also used to directly pass the 24-bit data from instruction memory to **S2_bus**. All ALU and MAC operations are accomplished in one clock cycle, but the BTC operation needs a variable number of clock cycles to be performed. At the start of the operation cycle, the source operands are placed on the S1 and S2 buses, and the operation commences. At the end of operation, the result is placed on the destination bus, and is stored in the destination register.

The RTL level architecture body of the CPU is constructed using these compo-

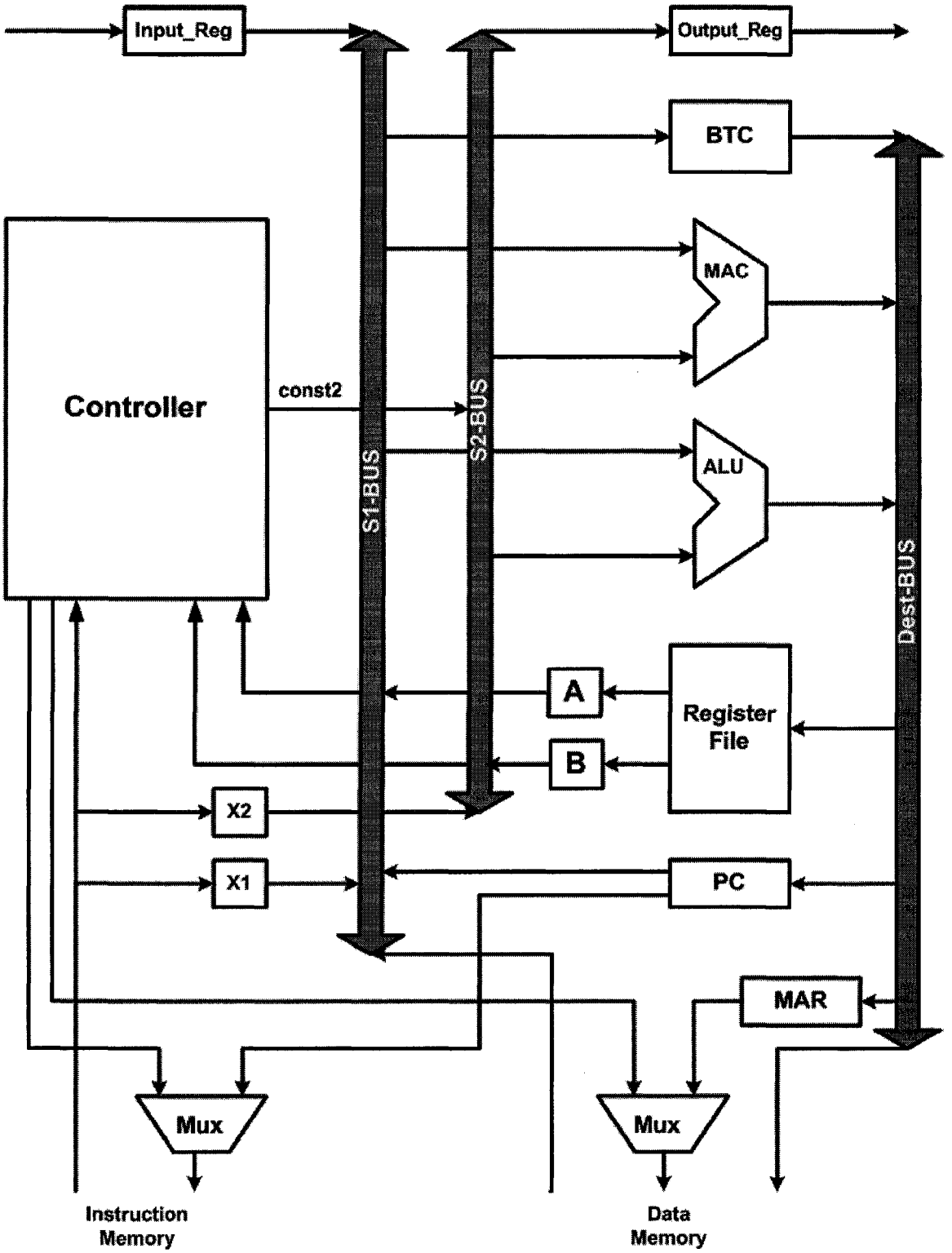

Figure 3.3: The TLNS CPU Organization

nents. The VHDL description of the "tlns" entity and its "rtl" architecture is coded in the "tlns.vhd" file. The declarative part of the architecture contains component declarations corresponding to each of the data path elements. The architecture body also declares signals corresponding to the connections shown in the data path diagram as well as additional control signals. The concurrent statement part of the architecture body consists of component instantiation statements that lay out the data path. The data input and output ports are connected using the declared signals, according to the CPU organization. The control ports of the data path component instances are connected to the declared control signals. Furthermore, the controller must be connected to the CPU's external control input and output ports so that it can sequence memory transfers.

We develop our design description of this implementation by a brief description of the data path entities and their behavioral architecture bodies. Finally, we describe the behavioral architecture of the controller that sequences data path operations. The VHDL code of all modules including their entity declaration and architecture body are found in Appendix A.

3.5.1 The Arithmetic and Logic Unit (ALU)

The ALU performs the operations on data needed to implement arithmetic and logic instructions. It is also used to perform address arithmetic for load and store instructions. The particular function to be performed by the ALU at any time is determined by the controller. The different function types and allowable values for each function are described in a separate package named **alu_types**. In this package, all ALU functions are defined as constant 4-bit encoded values. Two constants are also defined to represent identity operations on each of two ALU inputs.

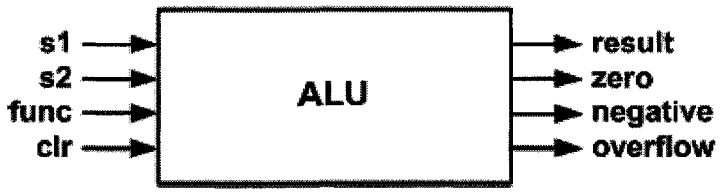

Figure 3.4: The TLNS ALU

The ALU block is shown in Fig. 3.4. The two ports, **s1** and **s2**, are the data inputs, and **result** is the data output. The **func** port selects the function to be performed, and the **zero**, **negative** and **overflow** ports are status outputs providing information regarding the **result** value. The **clr** port is used to control ALU access to the destination bus. If the ALU is not in use, it provides high impedance to its output, so that other components can use the destination bus.

The behavioral code of architecture body contains a single process, **alu_op**, that is sensitive to changes on any of the inputs. The process uses the function code as the selector expression in a case statement to select the function to perform. For the two identity functions, the result is simply a copy of the appropriate operand. For the logical and shift functions, the overloaded operators defined in the **numeric_bit** package, are used to determine the **result**. This package is an IEEE standard package which defines arithmetic operations on integers represented using vectors of **bit** elements. The ALU process include a local procedure, **add**, in order to provide a means of detecting overflow. The procedure includes a carry-in parameter and a Boolean parameter to indicate whether signed or unsigned addition should be performed. The process also generates status output values and sends them out through the ALU block ports.

3.5.2 The Registers

The CPU data path makes use of a number of different kinds of registers, most of which are rising-edge triggered. The registers A and B are similar, and both have an **enable** line to copy the value on the data input **d** to an internal variable. When the **out_en** signal is '1', this value is copied to output **q**. If **out_en** is '0', the output is disabled. The register block is shown in Fig. 3.5.

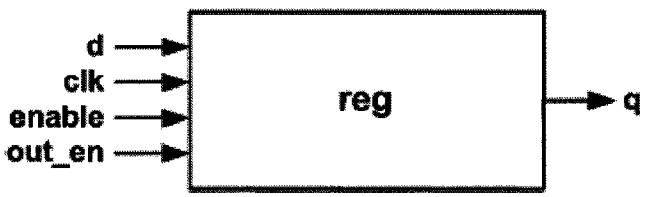

Figure 3.5: The TLNS Registers

The only difference between the Memory Address Register (MAR), shown in Fig. 3.6, and these registers is that MAR has no **out_en** port. Therefore, whenever the **enable** signal is '1', the data input **d** is directly copied to the output **q**.

Figure 3.6: The TLNS Memory Address Register (MAR)

The register used for the program counter has two outputs. The first one is connected to the **s1_bus**, and the second one goes to the multiplexer which addresses the instruction memory. The first output can be disabled from the bus when it is not necessary. This register, shown in Fig. 3.7, has an additional input port to reset the register to zero whenever the CPU is reset.

The register **input_reg** is not clocked, and there are just **d** and **enable** in its process sensitivity list. Since its input data, **d**, is a 16-bit binary word, it is resized to

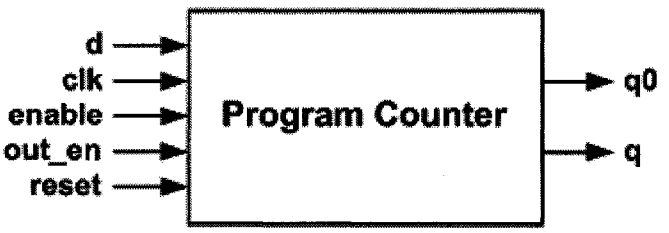

Figure 3.7: The TLNS Program Counter (PC)

24 bits by filling the most significant 8 bits with zeros. Fig. 3.8 shows the **input_reg** block.

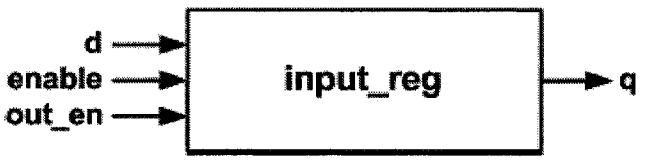

Figure 3.8: The TLNS Input Register

The block diagram of **output_reg** is shown in Fig. 3.9. Its behavioral architecture is different from the other registers. Here, the **out_enable** is an output signal and shows when the output of this register is ready to be read. There is another process in its architecture, to activate **out_enable**, based on the **enable** signal.

Figure 3.9: The TLNS Output Register

3.5.3 The Register File

The CPU data path includes a register file with two read ports and one write port. Ports **q1** and **q2** are the two read ports, and **d3** is the write port. Ports **a1**, **a2** and **a3**

are the corresponding register addresses, and **write_en** is a control input indicating when the write port should store a value into the register file. The register file block is shown in Fig. 3.10.

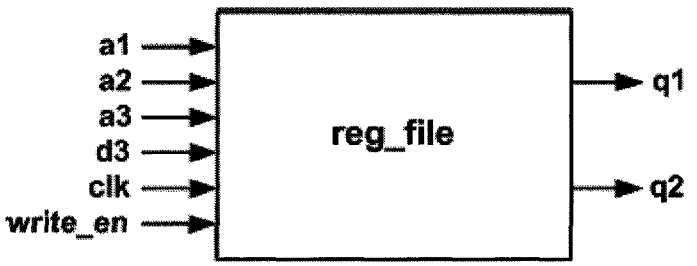

Figure 3.10: The TLNS Register File

Both register read and register write processes, **reg_write** and **reg_read**, in the architecture body contain an array of words to implement the register file storage. Since register r0 always represents zero, it never written to, and is excluded. When read, it returns the value zero, and when written, the data is discarded. The constant **all_zeros** represents the value returned when r0 is read. Having separate register read and register write processes ensure that a concurrent read and write of the same register returns the previously stored data word, not the recently written data word.

3.5.4 The Multiplexers

There are two multiplexers in the TLNS CPU data path. They have the same ports as shown in Fig. 3.11, but the different input data types of inputs causes a change in the selected signal assignment statement. By using these multiplexers, memories are also addressable by the controller. When the select input is '0', input **i0** is transmitted to the output, when it is '1', **i1** is transmitted to the output.

Figure 3.11: The TLNS Multiplexers

3.5.5 The Extenders

The CPU data path includes one extender and one extender / director block. The main use of these modules is to extend immediate values from the instruction to 24 bits. The extender block is shown in Fig. 3.12. The input port **d** is the data word containing the field to be extended, and **q** is the 24-bit output port. If the control input **immed_size_18** is '1', the rightmost 18 bits of **d** are selected for extension, otherwise the rightmost 10 bits are selected. If **immed_unsigned** is '1', the field is treated as an unsigned binary number and zero extended. Otherwise it is treated as a two's-complement signed number and sign extended. Finally, the **immed_en** controls when the extended value is used to drive the output, and when the output is disabled.

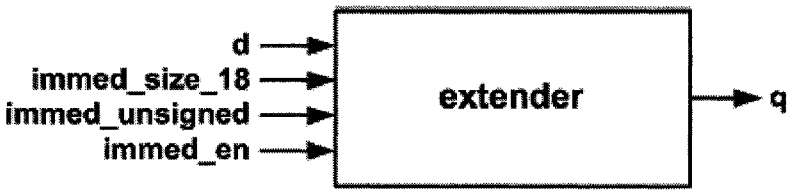

Figure 3.12: The TLNS Extender

The extender / director block as shown in Fig. 3.13 has an extra input port, **direct**. When data is read from the instruction memory, it should be directly transmitted to the **s2_bus**. In this case no extension is needed. Therefore, in this module, first the **direct** signal is checked, if it is '1', the **d** directed to the **q**, otherwise the same function as in the extender module is performed.

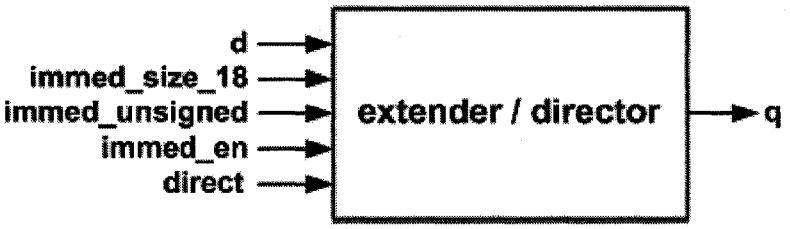

Figure 3.13: The TLNS Extender / Director

3.5.6 The Binary / 2DLNS Converter (BTC)

The BTC component of TLNS CPU data path consists of two blocks. The main block of this converter is a Verilog program, the **serial2digithighlow** module. This code obtained from [13], converts a binary value to its equivalent 2DLNS representation. Since there is no functional relationship between binary and 2DLNS representations, look-up tables should be used. Although a look-up table offers a simple and fast conversion scheme, the implementation of these tables can become very large and impractical as their size is exponentially dependent on the input binary dynamic range [8]. Therefore, a hardware realizable method of converting has been introduced in [14] using Range Addressable Look-Up Tables (RALUT). This converter is implemented using the serial method. The input port to this module as shown in Fig. 3.14, is **i**, which is the binary value. This converter uses two **reset** and **activate** control signals. The port **reset** should be set to '0' to start the conversion. The **activate** signal is also needed to set to '1' only for the first clock cycle of operation. The number of clock cycles for each conversion operation needed, varies based on the binary input value. When the conversion is accomplished, the **ready** signal is asserted.

The output of this module is the concatenation of signs, **output_sign**, first base exponents, **output_first**, and second base exponents, **output_second** of both 2DLNS digits. Therefore, another block is considered in order to split these outputs into separate digits. The inputs of this block are directly connected to outputs of the converter. This block is shown in Fig. 3.15. whenever its input **ready** signal is '1',

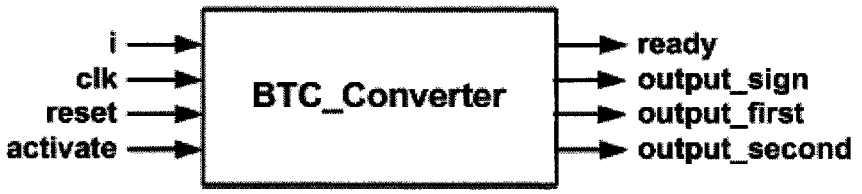

Figure 3.14: The TLNS Binary / 2DLNS Converter

all other inputs get reordered to form the appropriate 2-digit 2DLNS representation.

Figure 3.15: The TLNS Binary / 2DLNS Conversion Register

3.5.7 The Multiply and Accumulate unit (MAC)

Many Digital Signal Processing (DSP) algorithms, such as digital demodulation, filtering and equalization, make use of MACs. A MAC operates on two sequences of numbers, X_i and Y_i, multiplies corresponding elements of the sequences and accumulates the sum of the products. The result is

$$\sum_{i=1}^{N} X_i.Y_i$$

Where N is the length of the sequences. The MAC unit in the CPU data path is a fundamental feature and is composed of several components. Thus, the architecture of this unit is discussed in detail.

The **x** and **y** inputs to the MAC unit are 2-digit 2DLNS numbers. They are both

represented in 1-bit sign mode. Therefore, the sign bit is '0' for a positive number and is '1' for a negative number. Their multiplication is a summation of four 2DLNS digits. Two small parallel adders are used to add corresponding exponents of each base to form each partial product. Since addition in 2DLNS is not an easy operation, these partial products are converted to binary representations for addition. Two partial products are added in a single adder, and therefore three adders are used to add all partial products [15]. The final sum can be accumulated with the previous results and form the output signal **p**. The MAC unit is used to multiply two 2DLNS values, as well as multiply and accumulate the result in case of multiple sequences of data. The input signal **clr** clears the accumulator to zero whenever a new MAC operation commences. TLNS also uses MAC unit in order to perform a 2DLNS / Binary conversion. In this application, one operand is the 2DLNS value, which is read from a register, and the other one is 2DLNS representation of 1, which is the content of register r14. The converted binary value is written to the destination register. The external ports of the MAC unit are shown in Fig. 3.16.

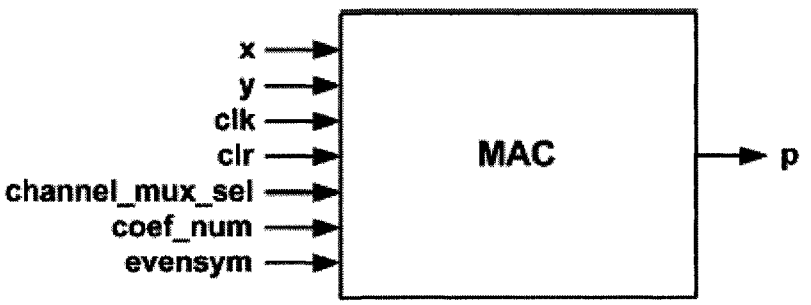

Figure 3.16: The TLNS Multiply and Accumulate unit (MAC)

The other input signals are considered in order to use MAC unit in a filter application. The MAC is the fundamental unit in Finite Impulse Response (FIR) filters. In symmetric filters, coefficients are duplicate in magnitude and based on even or odd symmetry, every other coefficient should be negated. Since only the sign of the coefficients may be different (depending on the symmetry of the filters) only the final

binary accumulator needs to be duplicated to generate output of each filter.

The implemented MAC unit multiplies each pair of data/coefficients as absolute values, and accumulates the results with the final sign separately. Therefore, it is capable to process dual filters at the same time. The **evensym** port is used to determine the type of symmetry. If the dual filter has been designed with even symmetry, **evensym** is set to '1', otherwise it is set to '0'. The input port **coefnum** is used to determine if the current coefficient is of an even order, which means it should be negated for the symmetric filter computations. Finally, the input signal **channel_mux_sel** is used to select the proper accumulator output to be sent to the destination bus. Since all the components, except for accumulator registers, are combinatorial in design, the MAC operation is performed in one clock cycle. This fact makes it appropriate for filter applications.

The organization of the MAC unit is shown in Fig. 3.17. All these components are instantiated in the RTL design of the MAC unit. When the **clr** signal is '0', the 24-bit accumulated value is placed on the output bus, otherwise the output is disabled. A brief description of its components, including entities and their behavioral architecture, are given in subsequent sections.

3.5.7.1 The Exclusive-or unit

In the first MAC stage, 2-digit 2DLNS numbers are multiplied and four partial products are generated. The sign of each partial product simply is determined by xoring the signs of the operands. Since these signs are inputs to the adder / subtracter units, a separate component is considered to compute them. Fig. 3.18 shows this unit.

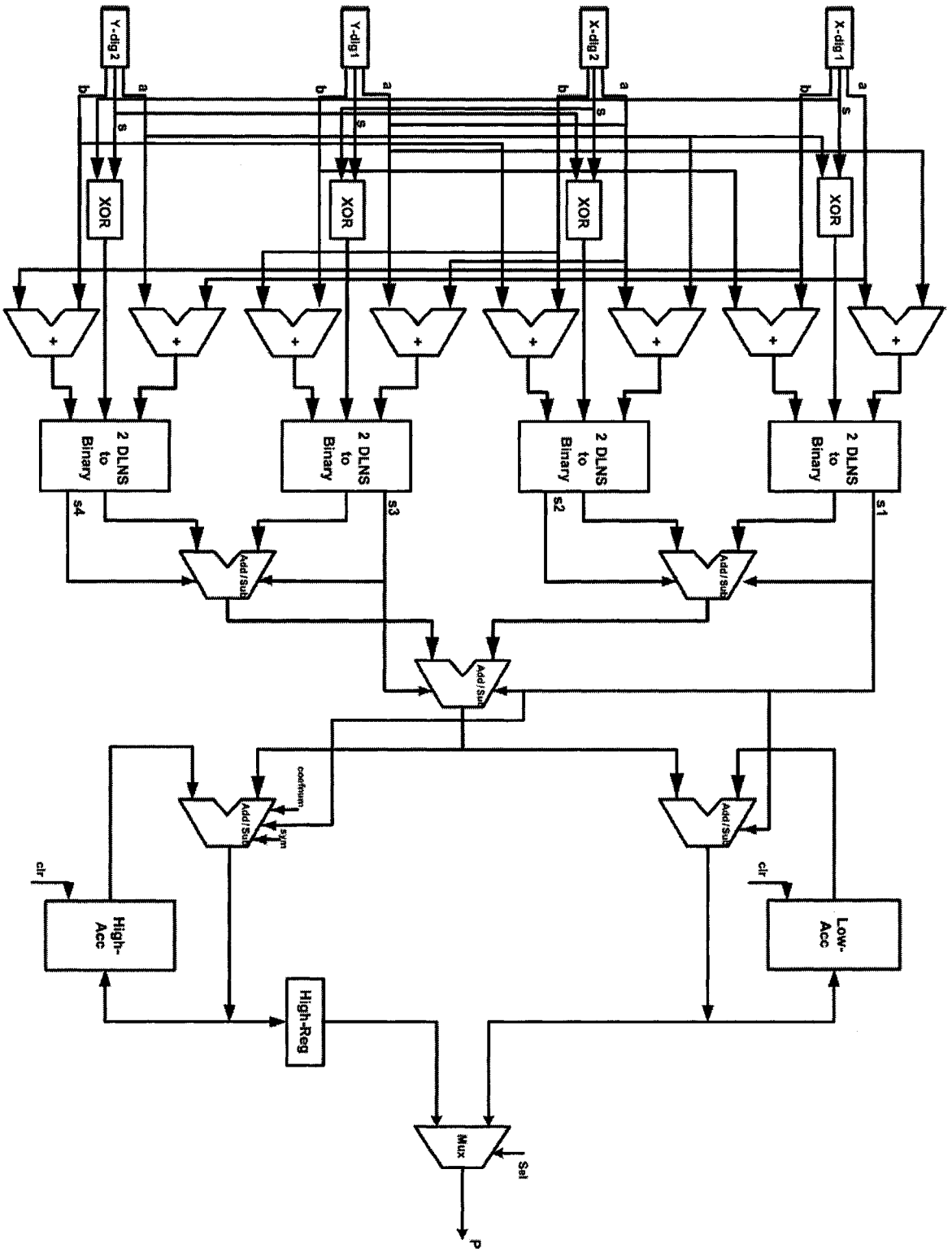

Figure 3.17: The MAC unit Organization

Figure 3.18: The MAC Exclusive-or unit

3.5.7.2 The First Exponent Adders

Since exponents are in two's-complement representations, this unit executes a behavioral code for a two's-complement adder. One extra bit is considered for the result, so an overflow is managed properly. Fig. 3.19 shows the adder ports.

Figure 3.19: The MAC unit First Exponent Adder

3.5.7.3 The Second Exponent Adders

Although the ports of these adders seem the same as the first exponents adders, as shown in Fig. 3.20, they are different in two aspects. First of all, the operands are not of the same size. Since in the 2DLNS computations, the size of the second exponent determines the size of RALUTs, it needs to be kept as small as possible. Thus, in addition to considering different sizes for operands, the range of values may also be limited. In this case, the limit is adjusted from -16 to 15 ($R = 5$) to -12 to 12, so that an overflow never occurs when the data is multiplied with the other operand with ($R = 3$). By limiting the indices in this way, the representation is used to its fullest [13].

On the other hand, the most negative second base exponent is considered as 2DLNS representation for zero. Therefore, if either of the second base exponents show the most negative two's-complement representation, the result of multiplication

Figure 3.20: The MAC unit Second Exponent Adder

should be zero as well, therefore the sum is also represented as the most negative second base exponent.

3.5.7.4 The 2DLNS / Binary Converter

This converter is Verilog code found in [13], which converts a 2DLNS value to its equivalent binary representation. As Fig. 3.21 shows, sign, first base exponent, and the second base exponent of the 2DLNS value are entered through separate ports. The converter parameters are set to generate the signed magnitude representation of a 16-bit binary value. Four extra bits are considered to increase the conversion precision, thus, the output is a 20-bit binary representation. The magnitude, **binaryout**, is treated as an absolute value in the next step of the MAC operation.

Figure 3.21: The MAC unit 2DLNS / Binary Converter

3.5.7.5 20-bit Adder / Subtracter

Two 20-bit adder / subtracters are used to add the outputs of converters. The implemented MAC unit adder / subtracter considers each pair of operands as absolute values, and the operands' signs are simply xored to determine if the operands should

be added or subtracted. One extra bit is considered for each Adder/Subtracter unit, so an overflow never occurs. The Fig. 3.22 shows the ports of this module.

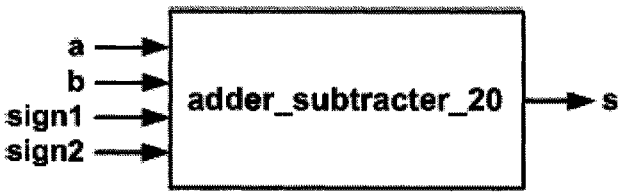

Figure 3.22: The MAC unit 20-bit Adder Subtracter

3.5.7.6 21-bit Adder / Subtracter

As it is shown in Fig. 3.23, this adder / subtracter is similar to the previous one with regards to its input and output ports. The structures are the same as well. The only difference is that this unit operates on 21-bit operands and the result is a 22-bit number. Thus, both operands are sign extended by one bit. The important point to notice is that the sign ports for this unit are the signs of the first and the third converter outputs.

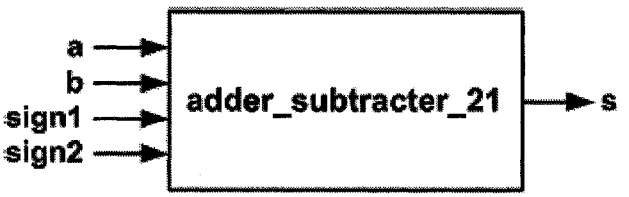

Figure 3.23: The MAC unit 21-bit Adder / Subtracter

3.5.7.7 23-bit Adder / Subtracter

There are two 23-bit Adder / Subtracter components in the MAC unit. One is used to accumulate the multiplication of the input pair of sequences, which is named channel low accumulator. The other one is only used in filter applications, specifically when

a dual symmetric filter exists. This accumulator is named channel high accumulator. Again these two have the same behavioral architecture as previous ones. Since they are the final accumulators in MAC unit, and one of the operands is the accumulated value of previous multiplications, two extra bits are considered for the result, in order to avoid an overflow. While the accumulated operand is a 24-bit value, the other one is signed extended by two bits and the adder / subtracter operates on 23 bits operands. This time for the channel low, only the sign of the first converter determines if the new value should be added to the accumulator, or subtracted from it. The Fig. 3.24 shows the block diagram of channel low adder / subtracter.

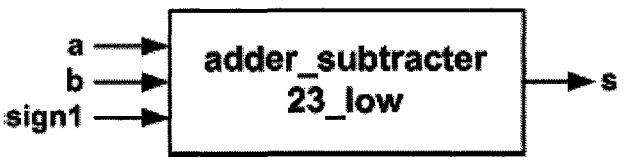

Figure 3.24: The MAC unit 23-bit Low-Channel Adder / Subtracter

For the channel high accumulator, there are two more input ports. The **sym** signal shows the type of symmetry. If this signal is set to '1', it means that an even symmetry exits, meaning that the products of sequences of an even order should be negated before being added to the accumulator. Therefore, another input port, **num**, is needed, which determines the order of current sequence. This signal is set by the CPU controller unit. This time, an exclusive-or of these two signals is xored with the sign of the first converter output, in order to determine whether the result should be added to the accumulator, or subtracted from it. Input and output ports of this unit are shown in Fig. 3.25.

3.5.7.8 Accumulator Register

There are two registers in feedback loops of low and high channels accumulators. These registers are completely similar. An input signal, **clr**, is considered to re-

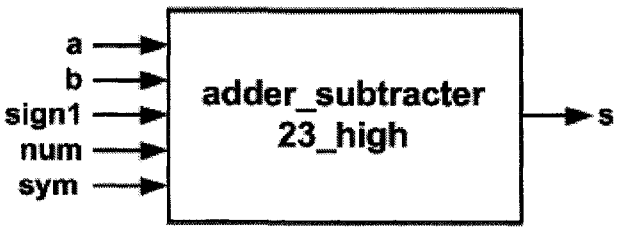

Figure 3.25: The MAC unit 23-bit High-Channel Adder / Subtracter

set the accumulators to zero, whenever a new multiply and accumulation operation commences. The block diagram of this register is shown in Fig. 3.26.

Figure 3.26: The MAC unit Accumulator Register

3.5.7.9 High Channel Register

This is an ordinary register which latches the output of the channel high accumulator for a single clock cycle. The output of this register is directly connected to a multiplexer input. Therefore, when the controller sets the **evensym**, input signal of the MAC unit to '1', the output of channel high is ready to sent out, exactly one clock cycle later than the output of channel low. The Fig. 3.27 shows the block diagram of this register.

Figure 3.27: The MAC unit High Channel Register

3.5.7.10 Channel Multiplexer

As above mentioned, A multiplexer is used in the last stage of the MAC unit to send the output to the destination bus of the CPU. By default the signal **sel** is set to '0'. So, it transfers the output of channel low to the outside. When MAC unit is used in a filter application with a symmetric filter, the controller switches the **sel** signal to '1' after the first output and gives the MAC unit another clock cycle to place the output of channel high on destination bus. When none of the outputs are ready to send out, the multiplexer output is disabled. The ports of the multiplexer are shown in Fig. 3.28.

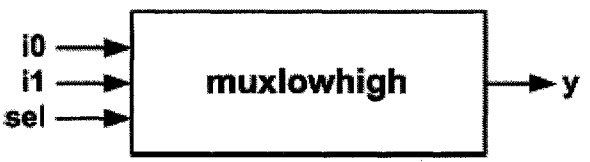

Figure 3.28: The MAC unit Channel Multiplexer

3.5.8 The Controller

The controller is a state machine which handles all CPU operations. Most of the input and output ports to this unit, as shown in Fig. 3.29, are control signals to other CPU components. The design of this unit is not exactly based on the classical microcoded method. The controller activates control signals to cause the data path components to act on the data, and manages these signals mostly by executing procedures. The controller behavioral architecture is implemented by the process **sequencer**. The input port **current_instruction** represents the current instruction read from the instruction memory. The aliases declared in the process represent the fields of this instruction. The variables **result_of_set_is_1** and **branch_taken** are used for the intermediate state when sequencing relational and branch instructions.

The remaining variables, are all defined for use in procedures which realize the "filter" instruction. Following these declarations are a number of local procedures that implement various stages of instruction sequencing.

The **sequencer** process begins by initializing the CPU's external control signals, and the internal data path control signals. All control signals are set to default values. The **sequencer** also disables its constant data output, and sets the starting state to s1. It then waits until **reset** is '0' on a rising **clk** edge before proceeding. Then **sequencer** uses a case statement to select among different alternatives, based on the instruction opcode. For each instruction or class of instructions, the **sequencer** calls local procedures to control the data path operations for the required stages, including execution, memory access, or register write back. The action for the case statement alternative corresponding to the **op_special** includes further case statement using the special opcode extension field to select alternative for the instruction subclass. The procedures which are used in Fetch and Decode stages, have been already described. The other procedures are briefly described in later sections.

Procedures to Execute Load and Store Instructions

The procedure **do_EX_load_store_1** sequences the calculation of the effective address for the load and store instructions during the Execute stage. The effective address is formed by adding the 10-bit displacement from the immediate value in the instruction to the source register operand. The procedure first enables the A register output onto the **s1_bus**, and the sign extended immediate value onto the **s2_bus**. The r_{s1} has already written to A register during the Decode stage. The procedure also sets the ALU function control signal to cause the ALU to add its operands, and enables the MAR register input to accept the result. After one clock

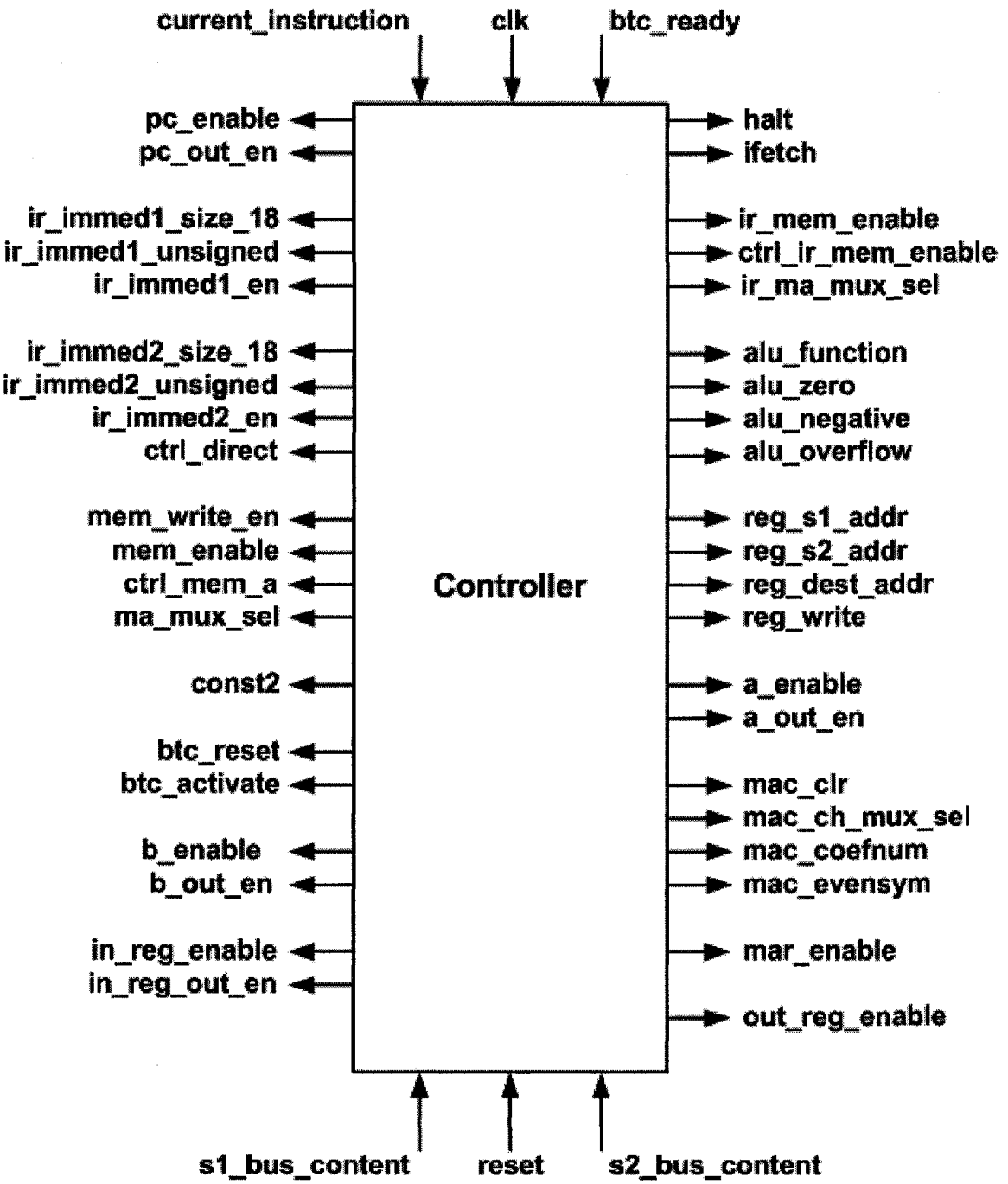

Figure 3.29: The TLNS Controller

cycle, when the effective address has been stored in MAR, in the next state, the procedure **do_EX_load_store_2** disables the A register and X2 extender outputs from the source buses, and disables the MAR input.

The memory reference for a load instruction is sequenced by the **do_MEM_load_1** procedure. This procedure accesses data memory to read a word. The read data is transferred onto the **s1_bus**. The procedure sets the ALU function to **alu_pass_s1**, so that the data will be available on **dest_bus** directly. One clock cycle later, in the next procedure, **do_MEM_load_2**, the data is written to destination register, r_d in the register file. This procedure also removes the operand from the **s1_bus** and disables the access signal to the memory.

The corresponding procedure that sequences the store memory reference is procedure **do_MEM_store_1**. This procedure accesses data memory to write a word. The r_d has already written to B register during the Decode stage. The contents of the B register are placed onto the **s2_bus**. The procedure sets the ALU function to **alu_pass_s2**, so that the data will be available on **dest_bus**, and sets the **mem_write_en** signal to '1'. Therefore, the data is written into the memory cell with the effective address which has been determined in the **do_EX_load_store_1** procedure.

Procedures to Execute Arithmetic and Logic Instructions

The procedure **do_EX_arith_logic_1** sequences the Execute stage of the arithmetic and logic instructions that draw both source operands from registers. The procedure first enables the two source operands from the A and B registers onto the s1 and s2 buses. It uses the opcode extension field to select which function the ALU should perform and sets the ALU function control signal accordingly. It also enables the destination register to accept the ALU result. In the next state, the procedure

do_EX_arith_logic_2 is called, which disables the A and B register outputs and the register file input.

The procedure **do_EX_arith_logic_immed_1** performs a similar process when one of the operands is an immediate value. For arithmetic operations, the 10-bit immediate value is sign extended. Then, in a case statement, the **alu_function** is determined. When the ALU result is written to the destination register, in the next state, the **do_EX_arith_logic_immed_2** procedure, disables the A register and the extender inputs and the register file input.

The other CPU instruction which uses ALU functions during execution is the "lhi" instruction. As mentioned above, this instruction is used to load a 10-bit immediate value into the most significant 10 bits of the destination register. The controller calls procedure **do_EX_lhi_1** to perform the "lhi" instruction. The immediate value is unsigned extended by the extender, and is transferred onto the **S1_bus**. The shift amount is determined by the **const2** signal, which is set to 14. The ALU function is specified in order to perform the shift left operation, and the result is written to the register file. In the next clock cycle, when the destination register is updated with the result, procedure **do_EX_lhi_2** disables all corresponding control signals.

Procedures to Execute Branch Instructions

The data path operations for the branch instructions are sequenced by four procedures. The **do_EX_branch_1** procedure enables the A register output and sets the ALU function control signal to cause the ALU to pass the source value through unchanged. The value itself is not used. Instead, the act of passing it through the ALU causes the **alu_zero** flag to be set, depending on whether the source value is zero or not. The flag is used to determine the branch outcome. After one clock cycle, when the value of the **alu_zero** flag has settled, procedure **do_EX_branch_2** is executed.

This procedure controls the testing of the source register to determine whether the branch is taken. The A register output is also disabled. The branch opcode is used to determine whether a zero or non-zero source value should cause the branch to be taken, and the Boolean variable **branch_taken** is set accordingly.
If the branch is taken, the procedure **do_MEM_branch_1** is called. The procedure controls the addition of the 10-bit immediate value from the instruction to the PC. It begins this sequence by enabling the PC register output and the sign-extended displacement onto the source buses. The procedure sets the ALU function to cause it to add its operands and enables the PC register input to accept the result. After one clock cycle, when the branch target address has been stored in the PC, the procedure **do_MEM_branch_2** is called. This procedure disables the PC and extender outputs, and the PC input.

Procedures to Execute Jump Instructions

The jump instructions are performed by adding the immediate value from the instruction to the PC's current value. Thus, corresponding procedures are similar to the procedures which execute branch instructions with a single difference. The jump instruction is done unconditionally. The procedure **do_MEM_jump_1**, enables the PC output onto the **S1_bus**, and the signed extended immediate value onto the **S1_bus**. The ALU function is set to add the data operands. The ALU result is once again written to the PC. In the next state, the procedure **do_MEM_jump_2** disables the extender and PC control signals.
In jump and link instructions, before procedure **do_MEM_jump_1**, the procedure **do_EX_link_1** is called. This procedure writes the current PC address into register r15. The ALU passes the PC contents to the register file, and the destination register is enabled. After one clock cycle, the **do_EX_link_2** procedure disables PC output

and register file input. In the next state the jump procedure commences.

There are two jump register instructions in the CPU Instruction Set. These instructions cause a jump to the address which is specified by a register. The procedure **do_MEM_jump_reg_1** enables the register A output onto **S1_bus**. The ALU function is set to pass this value to the destination bus, in order to be written to the PC register. When the PC is updated with this new address, in the next state procedure **do_MEM_jump_reg_2** disables both the register A output and the PC input. Again, the previous value of the PC can be kept in a link register, using a jump register and link instruction.

Procedures to Execute Relational Instructions

The controller includes two series of procedures for signed and unsigned operands that implement the Execute stage of relational instructions. These procedures are similar, only the setting of extender for signed and unsigned immediate values is different. The procedure **do_EX_set_unsigned_1** is called with an **immed** parameter which indicates whether the second operand of the instruction is an immediate value or a register operand. The procedure enables the A register output onto the **s1_bus** as the first source operand. If the second operand is an immediate value, it is extended and enabled onto the **s2_bus**, otherwise, the procedure enables the B register output as the second source operand. In either case, the procedure sets the ALU function code to cause the ALU to perform the subtraction function. The result of subtraction is not used. The status flags from the ALU are used to determine the relationship between the source operand values. It takes one clock cycle for ALU status signals to stabilize. Then, the procedure **do_EX_set_unsigned_2** is called. It disables the source operands from the source buses and uses either the instruction opcode or opcode extension to determine which relation is to be tested and sets the

Boolean variable **result_of_set_is_1** to the result of the test.
The relational instructions must set the destination register to the binary representation of the number 1 or 0, depending on the relation test. Then, it sets the ALU function code to pass the value on **s2_bus** to the destination bus. The register file input is enabled. When the result is written to the destination register, in the next state the procedure **do_EX_set_unsigned_3** removes the result value from the **s2_bus** and disables the register file input.

Procedures to Execute the Input Data Instruction

Reading the external data into the CPU is performed using an input register. The input data is latched to this register and when the register output signal is enabled, it is placed onto the **S1_bus**. The ALU passes the data onto the destination bus in order to be written to the register file. The procedure **do_EX_input_1** receives the destination register as an argument. The **in_reg_enable** and **in_reg_out_en** signals of the register are set to '1' at the same time. The **alu_function** signal is set to pass the data unchanged, and the destination register is enabled to be written.
In the next state, when the data is settled in the register file, procedure **do_EX_input_2** disables control signals to the input register and the register file.

Procedures to Execute the BTC Instruction

For a Binary to 2DLNS conversion, the binary data is read from a register. The first procedure, **do_EX_btconvert_1**, enables the A register output onto **S1_bus**. Meantime, **btc_reset** is set to '0' and **btc_activate** is asserted to '1', to start the conversion process. The **btc_activate** signal is needed to be '1' for a single clock cycle. Procedure **do_EX_btconvert_2** is called at the next rising edge of clock . This

procedure disables the **btc_activate** signal, and the A register output. This procedure also determines the destination register, and enables **reg_write** to be written. The conversion might take a variable number of clock cycles, so the controller waits for the **btc_ready** to be asserted to '1', and then executes the **do_EX_btconvert_3** procedure. This procedure simply disables the register write signal. In the next clock cycle, when the destination register is updated with the converted value, procedure **do_EX_btconvert_4** is executed, which resets the converter and releases the data buses.

Procedures to Execute the MAC Instruction

Both Multiplication and 2DLNS to Binary conversion use the MAC instruction. This instruction performs the **do_EX_mac_1** procedure to read data operands from registers by enabling A and B registers output onto data buses. It also activates the MAC unit to perform the multiply and accumulation operation. The result is written into the destination register. This procedure also enables the register file, by setting the **reg_write** signal to '1'. The multiply and accumulate is performed in a single clock cycle, thus in next state, procedure **do_EX_mac_2** is executed, which sets all corresponding control signals to their default values.

Procedures to Execute the Output Data Instruction

The TLNS CPU has an output register, in order to send out the processed data. The output instruction reads a register's contents and writes it to the output register. The procedure **do_EX_output_1** is called by the controller to execute this instruction. This procedure simply enables the B register output onto **S2_bus** and allows the output register to be written to. One clock cycle later, procedure **do_EX_output_2**

disables these signals.

Procedures to Execute the Filter Instruction

A brief discussion of the filter instruction seems necessary before the procedures to execute this instruction are explained. The filter instruction is mainly designed for FIR filters. FIR is a type of digital signal filter, in which every sample of output is the weighted sum of past and current samples of input, using only some finite number of past samples. In FIR filters:

$$y[n+1] = y[n] + h_c(n).h_d(n)$$

Here h_c represents the coefficient and h_d the data. In the TLNS CPU architecture, separate memories have been considered as instruction memory and data memory, both of which are addressable through the controller unit. In filter applications, coefficients and data are stored in separate memories, therefore they can be read into the MAC unit and processed in one clock cycle. The filter coefficients are preloaded into instruction memory and input data are stored in data memory, both in consecutive memory locations. In a real time application, input data samples are entered at specific time intervals. These samples are seated in a certain range of memory addresses. In order to maintain flexibility, these memory addresses are specified by the programmer. The TLNS controller should be aware of all of these addresses. The filter instruction transfers memory address information as well as other filter specifications to the controller.

As mentioned in previous sections, the filter instruction is an I-type instruction. In this case, instruction fields are particularly used in order to transfer some information with regards to the filter application. The filter instruction fields are shown in Fig. 3.30.

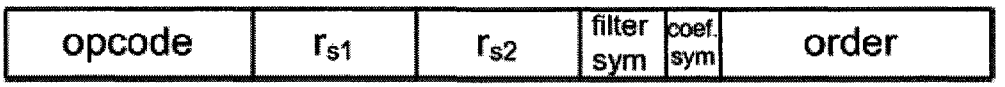

Figure 3.30: The TLNS Filter Instruction

r_{s1} is the register which contains the data and coefficient start addresses. The contents of r_{s2} is the data address range in the data memory. The 10-bit immediate value field is split into three fields. The 2-bit **filter_sym** field shows if the filter has a dual, which is a symmetric filter. The value of this field also shows the type of symmetry, if a dual filter exists. In symmetric filters, coefficients are duplicate in magnitude, and based on even or odd symmetry, every other coefficient should be negated. The 1-bit **coef_sym** field shows if the coefficients are symmetric, which means symmetric coefficients about a middle point have the same value. The remaining 7-bit field, determines the **order** of the filter. Therefore in a FIR filter application, a filter of maximum order 128 can be implemented.

When a filter instruction is decoded, first of all, **coef_sym** is checked. If the designed filter has an even symmetry, the signal **mac_evensym** is set to '1', and if it has an odd symmetry, the specified value for **mac_evensym** is '0'. IF the filter has no dual, this signal has no effect on operation, so is set to its default value. Then the procedure **do_EX_filt_start_1** is called. In this procedure, the contents of registers A and B are concurrently read to **S1_bus** and **S2_bus**. In the next clock cycle, when the **do_EX_filt_start_2** procedure is executed, the bus contents are read to some variables in the controller unit. The controller sets the addressing multiplexers to get the address from the controller, and using these variables specifies access addresses to the memories, and enables them both. Thus, two sequences of data and coefficients are ready to be read into the MAC unit. The MAC unit also needs to know the order of the coefficient which is being read. Therefore, the signal **mac_coefnum** is set to

'1' or '0', based on the type of symmetry. This signal is toggled whenever a coefficient is read. A counter is also initiated in the controller, to count filter iterations.

The next procedure is **do_EX_filt_mac**, which is called in the next state. In this procedure, the extender directs the coefficient read from instruction memory onto the **S2_bus**. The data memory also sends the corresponding data onto the **S1_bus**. Then, the MAC unit is enabled to read the data bus contents, and to commence the multiply and accumulation operation. The next coefficient address is determined based on the symmetry of filter. The next data address is also checked to be in the specified range of data memory. Finally, the next data and coefficient addresses are sent to the multiplexers, to address memories, and the iteration counter increments by one. As long as the iteration counter is less than order of the filter, this procedure is executed repetitively, and when the counter shows the order of filter, procedure **do_EX_filt_last** is called.

When **do_EX_filt_last** procedure is executed, the output data is ready to be written to the destination register. The register r12 has been specified as destination register by default. The memory control signals are also disabled in this procedure.

As previously explained in the MAC operation, the output for the dual filter is computed in a parallel accumulator. In the next state, firstly the filter symmetry is checked. If it is a single filter, the final procedure, which is **do_Ex_filt_out**, is called, otherwise, an extra clock cycle is provided for the MAC unit to write the output of the other accumulator into register r13, and then procedure **do_Ex_filt_out** is executed. In this procedure, all control signals are set to their default values, including extender control signals, multiplexers selectors, MAC control ports, and register file control signals.

3.6 TLNS CPU Test

As usual, testing the RTL model of a design is performed by providing a test bench model. Since the function performed by the CPU is to execute a machine language program stored in memory, the CPU can be tested by including a memory in the test bench. The instruction memory is preloaded with a program, which has been written using the instructions in the TLNS Instruction Set. The data memory is implemented in a similar way and loaded with proper data. It is also necessary to include a clock generator in the test bench to drive the **clk** and **reset** ports of the CPU. Another component is considered to read the external data from a file. By executing a program in instruction memory, CPU ports can be monitored to verify that it is fetching and executing the program instructions correctly.
The block diagram in Fig. 3.31 shows the test bench components and their connections.

The VHDL code of test bench and its components are all included in Appendix A. The test bench features are described in following sections.

3.6.1 The Test Bench Clock Generator

The clock generator entity has a generic constant, **Tclk**, that is used to specify the clock period. The architecture body contains two processes, one to generate the **reset** signal, and the other to generate the **clk** signal. The process **reset_driver** generates a single pulse on **reset**, starting at the beginning of a simulation and lasting three and a half clock periods. The process **clock_driver** initializes the **clk** signals to '0', then waits for a clock period. It then enters an infinite loop, in which it schedules the clock transitions for the next cycle, and then waits for a cycle.

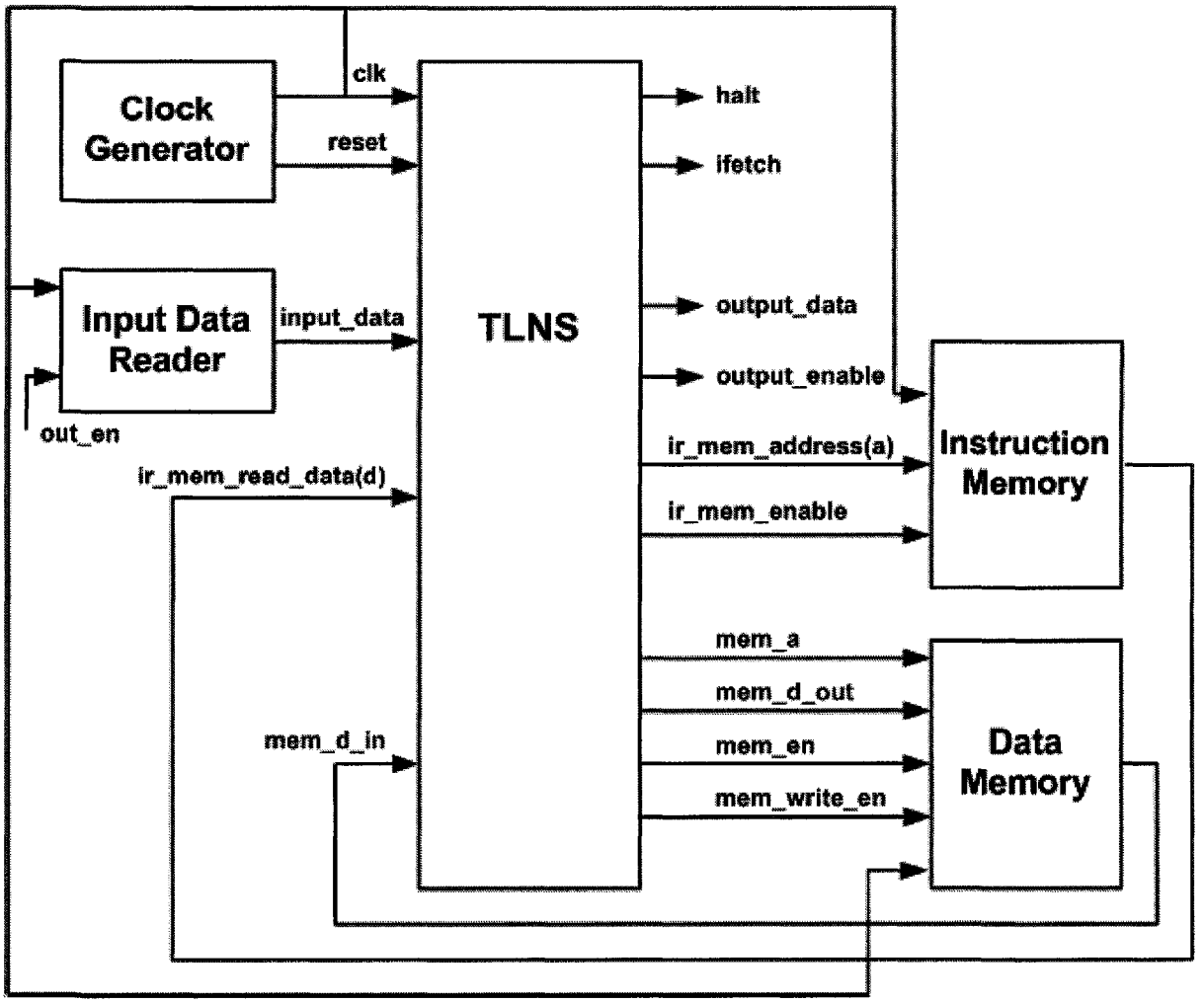

Figure 3.31: The TLNS Test Bench Organization

3.6.2 The Test Bench Instruction Memory

There are some generic constants in the entity declaration of instruction memory. The first constant, **mem_size**, is used to determine the amount of storage implemented within the memory. The remaining generic constants control the timing behavior. The **Tac_first** constant, is the access time for a single word in memory, and **Tpd_clk_out** is the propagation delay between a clock edge and a resulting output transition. The ports of the memory entity correspond to those of the CPU. They

include the clock signal **clk**, the address bus **a**, the data read bus **d**, and the control signal **ir_mem_enable**.

The behavioral architecture of the memory is implemented by the **mem_behavior** process. The constant **high_address** defines the range of addresses to which the memory responds, based on the size of the memory. Next, the process declares an array type **memory_array**, and a variable **mem** of the array type to represent the memory storage. Each element of the array is a one word bit vector. The procedure **load** is used to "preload" the memory. The array aggregate which is written in the declaration of the constant in **load** contains the binary representation of program. The procedure copies the program into the memory array, starting at address 0.

The process begins by calling the **load** procedure to preload the memory. The process then initializes the memory output port and enters a loop to handle memory access cycles. The start of a cycle is indicated by the rising edge of **clk**. Then, the process converts the byte address input to a numeric value and determines whether the address is within the memory address bounds. When this is detected, if the **ir_mem_enable** port is '1', the process performs a read access. It simply reads the memory word, places it on the data bus port, and lets the CPU to read it.

3.6.3 The Test Bench Data Memory

The only generic constant **data_memory_size** in the data memory entity declaration shows the amount of available storage locations. Again, the memory component ports, including the clock signal **clk**, memory address **mem_a**, data which is written to memory **mem_d_in**, data which is read from memory **mem_d_out**, and the control signals **mem_write_en** and **mem_enable** match with the CPU's.

The process **data_memory_behavior** implements the behavioral architecture. The constant **high_address**, array type **data_memory_array**, and variable **data_memory**

are defined similarly to those in instruction memory. The procedure **load** is defined to preload the data memory.

The process **data_memory_behavior** commences by calling the **load** procedure, then initializes its data port and enters a loop to handle memory read and write cycles. The start of a cycle is indicated by the **mem_enable** port being '1' on a rising edge of **clk**. When this is detected, the process converts the byte address to an integer value, and determines whether the requested access is a read or write. If the address is within the memory address bounds, the memory proceeds with the access cycle. In the case of a write, the process writes corresponding data into **data_memory**, and disables the data output port. The word address is used as an index to determine which memory array element to update. In a read cycle, the memory content of specified address, is placed on the data bus **mem_d_out**.

3.6.4 The Test Bench Input Data Reader

In some applications, the TLNS CPU needs to process external data. The input data reader component of test bench provides facility for reading input data from a file. The predefined package **textio** in the library **std** is used for reading from and writing to text files. These operations make it possible to read files which are produced by other sources, and write files that can be read by other software tools. The component has an input port **out_en** which specifies when data should be read, and an output port **data_in** which is used to transfer the read data. The file containing data is defined in the process **read_input** and is opened in read mode. On the rising edge of the clock, the **out_en** signal is checked. Whenever this signal is '1', data is read from the input file to the **data_in** port. The CPU instance in the test bench receives this data to perform further processes.

3.6.5 The TLNS CPU Test Bench

The TLNS CPU test bench is composed of several components. The test bench architecture includes component declarations corresponding to the clock generator, instruction memory, data memory, an input data reader, and the TLNS CPU entities previously described, as well as an instance of each of the components. In component declaration, the value of generic constants are also specified. For the clock generator, a period of 20 ns is considered. This corresponds to a clock frequency of 50 MHz. Both memory sizes are specified as 1024 bytes. The clock-to-output propagation delay for both memories and the CPU is 2 ns and access time is determined as 70 ns. A process **write_output** implemented in the architecture body of test bench entity writes the result of the CPU function to an output file. This facility is particularly used in the filter application of the TLNS CPU. Upon the rising edge of the clock signal **clk**, if the **out_en** port is '1', the value of **data_out** port is written into a file.

The description of TLNS CPU test bench components, ends this chapter. The designed CPU has been tested with a number of programs including all instructions in the Instruction Set. The test program, which is attached in Appendix A, is the program which has been written for the filterbank application. This program will be discussed in the next chapter.

Chapter 4

Filterbank Application

4.1 Filterbank Introduction

Digital Signal Processing (DSP) increasingly permeates most science and engineering fields. Digital signals can be generated, received, or analyzed through electronic components, test equipment, data acquisition devices, sensors, simulation hardware, and so on.

A fundamental operation in DSP is filtering. By filtering a signal, unwanted portions of the signal spectrum can be removed, or the signal can be modified and reshaped [8].

Having different frequency characteristics in some digital signals, may lead to different processes, such as amplification and attenuation, in separate frequency ranges. These processes needs to be carried out in multiple frequency bands.

A filterbank is an array of band-pass filters that separates the input signal into several components, each one carrying a single frequency sub band of the original

signal. These bands are then processed with custom algorithms. It also is desirable to design the filterbank in such a way that sub bands can be recombined to recover the original signal. Most binary implementation either use a modulated DFT or Interpolated FIR filter (IFIR) approach to perform the signal separation because they reduce the number of multiplications. When using MDLNS, every multiplication is converted to addition or subtraction, therefore, a simple FIR structure can be implemented to split the input signal [13].

One of the most basic structures used in DSP is the Finite Impulse Response (FIR) filter, or a non recursive filter. A FIR filter is a filter that does not depend on any past output values of the filter. If it is assumed that the FIR filter is linear, time-invariant, and causal, then the response of the filter, $y(nT)$, can be expressed as a difference equation:

$$y(nT) = \sum_{i=0}^{N} a_i.x(nT - iT)$$

where a_i represents constants, and N is the order of the filter. Non recursive filters are linear and guarantee a linear phase response. Therefore, for applications that require a linear phase response, such as video and audio applications, FIR filters are typically used. This is due to the guarantee of linear phase response, as well as ease of design and stability. As it can be seen, FIR filtering requires the use of inner product computations, which is based on MAC operations. Therefore, specialized DSP hardware, such as FIR filter microchips, heavily rely on optimized MAC operations. Through the use of MDLNS, a MAC architecture is built exclusively with adders, which provides a considerable reduction in hardware. MDLNS has a non-linear number representation and also provides a reduction in size of the data representation, that is advantageous in most systems. Thus, by using MDLNS a lower cost ($area \times power$) implementation in filterbank arithmetic operations is achieved. On the other hand, MDLNS provides a logarithmic type of representation with the

added advantage of allowing orthogonal implementations of the index computation in each base. Considering these all properties makes MDLNS an ideal candidate number system to implement a filterbank application [12].

4.2 Filterbank Design

Recently a research project has been conducted into next generation digital hearing instruments [17], and has been followed through further research on current digital filterbank algorithms, architecture, and hardware implementation [12].
In the latter, a custom filterbank architecture, based on 2DLNS, has been designed to be implemented in a hearing aid instrument. Later, in [13], this filterbank design has been considerably improved and implemented in a serial structure. In this research project, the same filterbank specifications are used. In this regard, some auxiliary programs which have already been written can be used as well.

The filterbank is one of the core parts of the digital hearing instrument. Its performance determines the frequency resolution of the instrument and the gain limits in each frequency band [17]. The main specifications for a filterbank design are its frequency range, number of bands, and stop band attenuation. Although its group delay, power consumption, and its size are also important.
The frequency range of human hearing is from 20 Hz to 20 kHz. Because of the octave band characteristic of the human hearing, good quality sound can still be achieved with half of the frequency range coverage. In this filterbank design, 16 kHz is taken as sampling frequency, assuming that the input is band limited to 8 kHz. Monitoring audio grams have shown that, considering 8 bands for filterbank provides an acceptable resolution for hearing instruments.
The stop band attenuation in each band determines the gain range of the hearing instrument, and the order of the filter is proportional to stop band attenuation and pass band ripple. These specifications of the filterbank require a 0.01 dB pass band

ripple and a 60 dB stop band attenuation for all the filters. In order to avoid any destruction in sound signal, the group delay is kept as small as possible, and 10 ms is determined as the border.

To minimize both power consumption and size, the minimum number of digital elements should be used in a filterbank architecture, and it can be realized by using MDLNS. It was explored that if equal bandwidths are considered for all filters with the same overlap margins, symmetrical filters could be designed. The advantage of generating symmetrical filters is that the overall magnitude response of the filters is perfectly flat (0 dB) across the entire frequency range, and there are duplicate coefficients (in magnitude) between the low and high bands. Since only the sign of the coefficients may be different, only the final binary accumulator needs to be duplicated to output each band. Using symmetric filters saves resources over non-symmetrical filters in a MDLNS FIR filterbank implementation. By using enough filter bands, filterbank custom-tailoring for the individual user will not be necessary. A general filterbank can be used for all users. Therefore, allowing the coefficients to be fixed which will also further improve the hardware implementation [13].

As previously mentioned, an improved design for a two-digit 2DLNS filterbank has been recently implemented [15]. In this research, 2DLNS was applied to the construction of a FIR filterbank. Using 8 separate and equal bands, filters were designed in MATLAB. Eight 75-tap filters were deemed acceptable with a 0.0128 dB pass band ripple and 58.9 dB stop band attenuation. Finding the optimal value for the second base is the next step. The optimal base is the value for the second base with the minimum number of bits for its corresponding exponent (R) that satisfies the filterbank specifications. In order to minimize the size of RALUTs, R needs to be kept as small as possible. Only 152 of 600 coefficients are unique in magnitude, which simplifies the search for an optimal base with a minimum value of R.

The 16-bit signed binary input data is converted to 2DLNS via a high/low serial implementation [16]. Considering a 6-bit first base index ($B = 6$) and 5-bit second base exponent ($R = 5$), provides a reasonably precise 2DLNS representations for input data. If the filter coefficients are represented with fewer bits for R, ($R = 3$), and the second base indices of input data limited to -12 to 12, an overflow never occurs when the input data is multiplied with the coefficients. With $R = 3$, the range of the coefficient's second base is from -3 to 3 which improves the filterbank responses to 0.0137 dB for the pass band ripple, and 58.2 dB for the stop band attenuation. With these specifications, the optimal base has been determined as $D = 0.92024380912663017$. Assuming that in this design the sampling frequency is 16 KHz and two of the 600 coefficients are processed each cycle, an operating clock of 16000 Hz $\times$ 600/2 = 4.8 MHz is required. The details of improved filterbank architecture design can be found in [13].

In the next part of this research work, the filterbank application is considered to show the processing capabilities of the TLNS CPU. In this regard, a filterbank architecture with the same specifications is implemented. Here, the hearing instrument processing tasks are microcodes running on the TLNS processor.

4.3 Filterbank TLNS Program

As mentioned in previous sections, in a filter application, precomputed coefficients are stored in the instruction memory. On the other hand, the input samples reside in consecutive addresses of data memory. Each pair of data and coefficient are read through data buses, placed onto the MAC unit input ports, and processed in one clock cycle. The accumulated results of MAC unit are written to the register file. When the output samples are tagged properly to identify the corresponding filter, they are transferred to the CPU output port.

The most important factor to be specified in the "filter" instruction is the order of

filter. This factor determines the number of instruction memory locations to store coefficients. The number of iterations for the multiply and accumulation operation is also specified by the order of filter. In order to achieve more flexibility, the programmer can manage both memories, and specify memory address ranges to store filter coefficients and input data. For a symmetric FIR filter design, symmetric coefficients are duplicated. Thus, storing half of the coefficients in memory is sufficient. In addition, the same "filter" instruction can be used to process a set of filters in a filterbank, as well as a single filter, where only corresponding fields should be set properly. In the case of processing dual filters, the type of symmetry should be specified, if there is any. Therefore, the program is dynamic, which will allow run time loading of the parameters such as filter order, symmetry of filters, symmetry of coefficients, and the addresses of data and coefficients in memory.

The TLNS program for the filterbank application consists of several parts. This code is preloaded to instruction memory and executed as a part of processor test bench. The first 8 instructions of this program write memory addresses to registers:

X″201404″,	–1		addi	r0, r5, dstart	registering addresses
X″2429FF″,	–2		addi	r0, r10, dend	
X″3C11FF″,	–3		lhi	r0, r4, dend	
X″0114E5″,	–4		or	r4 ,r5, r3	
X″20044E″,	–5		addi	r0, r1, data_address	
X″200840″,	–6	next	addi	r0, r2,coef_address	
X″50600E″,	–7		slli	r1, r8, E	
X″0208A5″,	–8		or	r8, r2, r2	

In the first instruction, **dstart** is added to register r0 content, which is zero, and the result is written to register r5. This is a more convenient way to write an im-

mediate value into a register. The value of **dstart** shows the start location of the memory range to store the input data. The last location is specified by **dend**, which is written to register r10 in the second instruction. These two addresses, **dstart** and **dend**, are concatenated into a single register for the "filter" instruction. Thus again, in the third instruction, **dend** is also written to the leftmost 10 bits of register r4. In the fourth instruction, the **or** operation is used to concatenate the contents of r4 and r5, and write the result to register r3.

Instruction 5 of the program writes **data_address** to register r1. **data_address** is the start address of the data in memory, and is an address in the range specified by **dstart** and **dend**. In the next instruction, the start address of the coefficients, **coef_address**, is placed in register r2. This instruction is labeled "next", just to show that, later, in the jump instruction of the program, the control is transferred to this instruction. Again, the content of r1 is shifted to the leftmost 10 bits of register r8, and the next instruction concatenates registers r2 and r8, and writes the result to r2. Now, registers r2 and r3 contain the address information which should be transferred to the CPU controller unit.

The next four instructions are codes for entering the quantized input sample to the CPU, converting it to 2DLNS representation, storing the converted value into the memory, and finally executing the filtering process:

X″401800″,	–9	inpt	r0, r6, 0	Entering Data
X″199*C*00″,	–a	btc	r6, r7	Converting Data
X″*AC*5*C*00″,	–b	sw	M[r1], r7	Storing Data
X″548*DCB*″,	–c	filter	r2, r3, coef sym, even, 75	Filters 0,7

In instruction 9 of the program, input data is written to register r6. The next instruction, "btc", reads the r6 contents to the binary / 2DLNS converter and the result

is written to register r7. The contents of r7 are stored to the data memory address which is determined by the contents of register r1. This register still contains the value of **data_address**. Then the "filter" instruction is executed. In the filterbank application, 8 even-symmetric filters are considered. The coefficients of dual filters are symmetric, and all filters are of 75th order. When this instruction is decoded by the CPU, information regarding the symmetry and order of filter are directed to the controller. Registers r2 and r3 are also read, and their contents placed onto data buses.

The next group of instructions tag the output sample in order to show the corresponding filter, and write the data to the CPU output port:

X″202400″,	–d	addi	r0, r9, band_tag	Writing output
X″533004″,	–e	slli	r12, r12, 4	
X″027325″,	–f	or	r9, r12, r12	
X″443000″,	–10	oupt	r0, r12, 0	
X″202C07″,	–11	addi	r0, r11, dual_band_tag	
X″537404″,	–12	slli	r13, r13, 4	
X″02F765″,	–13	or	r11, r13, r13	
X″443400″,	–14	oupt	r0, r13, 0	

band_tag is a 4-bit number, which shows what filter output is written. This value is added to the contents of register r0, which is zero, and written to register r9. As previously mentioned, the outputs of the "filter" instruction are registered in r12 and r13. Then, in next instruction, the 20-bit value of register r12 is shifted left for 4 bits. By oring registers r12 and r9, r12 contains a 24-bit value, including the output sample, and its corresponding filter number. The next instruction writes the contents of r12 to the output register. The same operations are done for the dual

filter. This time, **dual_band_tag** is written to register r11, then the contents of r11 are ored with the shifted contents of r13. Next, register 13 is read to output register.

X″208826″,	−15	addi	r2, r2, next_coef_address	Next Coefficients
X″548*DCB*″,	−16	filter	r2, r3, coef sym, even, 75	Filters 1,6
X″202401″,	−17	addi	r0, r9, band_tag	Writing output
X″533004″,	−18	slli	r12, r12, 4	
X″027325″,	−19	or	r9, r12, r12	
X″443000″,	−1a	oupt	r0, r12, 0	
X″202*C*06″,	−1b	addi	r0, r11, dual_band_tag	
X″537404″,	−1c	slli	r13, r13, 4	
X″02*F*765″,	−1d	or	r11, r13, r13	
X″443400″,	−1e	oupt	r0, r13, 0	

The TLNS test bench writes the output port of the processor to a file. The tag part of the entries to this file is used to split the corresponding output samples of each filter to a separate file. These files will be used later in a MATLAB program to plot the filterbank outputs.

It also worth mentioning that the result of filtering might be stored into memory for further processes, or might even be transferred to output ports in a different way. In this program, the outputs of each filter are merged with its corresponding tag to one register, and then this register's content has been transferred to the output register. Therefore, we are able to split the corresponding outputs for each filter and plot the result. In any other filterbank application, another appropriate way may be considered for output data.

When the output of filters 0 and 7 are computed, the contents of r2 are replaced with the memory address of next set of coefficients for filters 1 and 6. Since in filterbank

design symmetric coefficients of each filter are duplicated, only 38 unique coefficients reside in memory. Therefore, in order to specify the memory address of next set of coefficients, only the immediate value of 38 is added to the current value of register r2.

When the "filter" instruction with the new r2 contents is executed, the results for filters 1 and 6 are ready to output. Again, the same process for writing output data commences.

The next group of instructions executes the same procedure for filters 2 and 5:

X″208826″,	–1f	addi	r2, r2, next_coef_address	Next Coefficients
X″548DCB″,	–20	filter	r2, r3, coef sym, even, 75	Filters 2,5
X″202402″,	–21	addi	r0, r9, band_tag	Writing output
X″533004″,	–22	slli	r12, r12, 4	
X″027325″,	–23	or	r9, r12, r12	
X″443000″,	–24	oupt	r0, r12, 0	
X″202C05″,	–25	addi	r0, r11, dual_band_tag	
X″537404″,	–26	slli	r13, r13, 4	
X″02F765″,	–27	or	r11, r13, r13	
X″443400″,	–28	oupt	r0, r13, 0	

And finally, the next set of instructions, by computing the outputs of filters 3 and 4 and writing the results to CPU output port, complete the filtering operation for one input sample of data.

When the processor is done with an input sample, the program should specify the next memory address, **data_address**, for the input data. Since the memory addresses

X″208826″,	–29	addi	r2, r2, next_coef_address	Next Coefficients
X″548DCB″,	–2a	filter	r2, r3, coef sym, even, 75	Filters 3,4
X″202403″,	–2b	addi	r0, r9, band_tag	Writing output
X″533004″,	–2c	slli	r12, r12, 4	
X″027325″,	–2d	or	r9, r12, r12	
X″443000″,	–2e	oupt	r0, r12, 0	
X″202C04″,	–2f	addi	r0, r11, dual_band_tag	
X″537404″,	–30	slli	r13, r13, 4	
X″02F765″,	–31	or	r11, r13, r13	
X″443400″,	–32	oupt	r0, r13, 0	

are considered consecutively, register 1, which contains the data address, increments by 1. The new address should not exceed the upper range of data in memory, which is **dend**. Hence, it is compared to the contents of register r10. If **data_address** is still less than **dend**, the CPU branches to instruction 6 of program, and continues the operation. When **data_address** is equal to **dend**, the next data address in memory will be **dstart** . Thus, this value, which are the contents of register r5 is written to register r1. Then, the CPU jumps to instruction 6 of program and resumes execution:

X″204401″,	–33		addi	r1, r1, 1	Next Data address
X″0069AB″,	–34		sgt	r1, r10, r6	
X″118001″,	–35		beqz	r6, cont	
X″214400″,	–36		addi	r5, r1, 0	
X″0$BFFCE$″,	–37	cont	j	next	

This program is run for a prespecified simulation time, unless a "halt" instruction

terminate the program.

4.4 Filterbank Results

As it can be seen, this program has been written in 55 words and just with 12 different instructions from the CPU Instruction Set. While writing addresses including **dstart**, **dend**, **data_address** and **coef_address** into registers takes 41 clock cycles, this program needs 177 clock cycles for processing one input sample. Executing the "filter" instruction requires 82 clock cycles for 75th order filters. These numbers of clock cycles include 1 cycle for fetch the instruction, 4 cycles to access data and coefficients in memory, 76 cycles for multiply and accumulation, and another clock cycle to reset the control signals.

Operations	Number of Clock-Cycles
Registering Addresses	41
Entering Data	5
Converting Data	16
Storing Data	5
Filtering	82×4
Writing Output	40×4
Addressing Coefficients	5×3
Addressing Data	24
Total	594

Table 4.1: Filterbank Timing Results

The remaining 95 clock cycles out of 177, are used for entering data to the CPU,

performing Binary / 2DLNS conversion, storing data into data memory, writing the results, and finally specifying next addresses for data and coefficients.
The table 4.1 summarizes these results.

In order to keep the CPU generic, no special purpose commands are designed to generate the filterbank outputs. As a special instruction, all "inpt", "btc", "sw" instructions or a combination of them could be merged together, in order to significantly reduce the total number of clock cycles.

In order to test this program, filter coefficients are generated in MATLAB. By executing the **optimal base** software, in [13], the optimal base is specified. Once the optimal base is determined, the coefficients are mapped to that base. Then, a chirp signal is applied to input port of TLNS CPU. A chirp is a signal in which the frequency increases or decreases with time. In our sinusoidal chirp signal, which has been generated in MATLAB, the frequency increases with time. The same chirp generator in [8] has been used. Fig. 4.1 shows this signal.

Another MATLAB program has been used to display the filterbank results. Executing this program generates the output graphs. These graphs show that the filterbank operates properly. All codes used to generate the coefficients and input data, as well as the programs used to split the output data and convert it to decimal values and plot the graphs, have already been written and used in improved filterbank design [13].

Considering that TLNS requires 594 clock cycles to process one input sample, and sampling frequency is 16 KHz, the minimum clock frequency will be 16000 Hz $\times$ 594 = 9.5 MHz. The TLNS CPU has been successfully synthesized using 0.18 μm CMOS technology. In both filterbank custom design and improved design, just cell area and interconnects have been considered. Since, in our design, the total area consists of

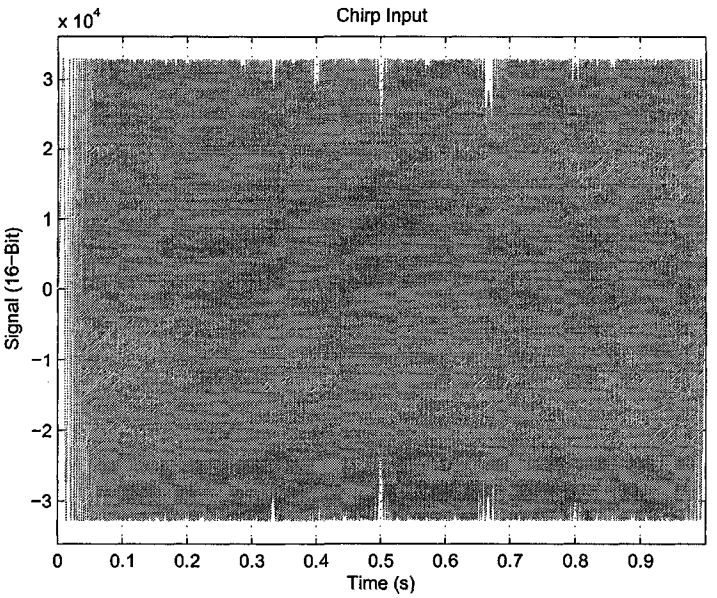

Figure 4.1: The Filterbank Input Signal

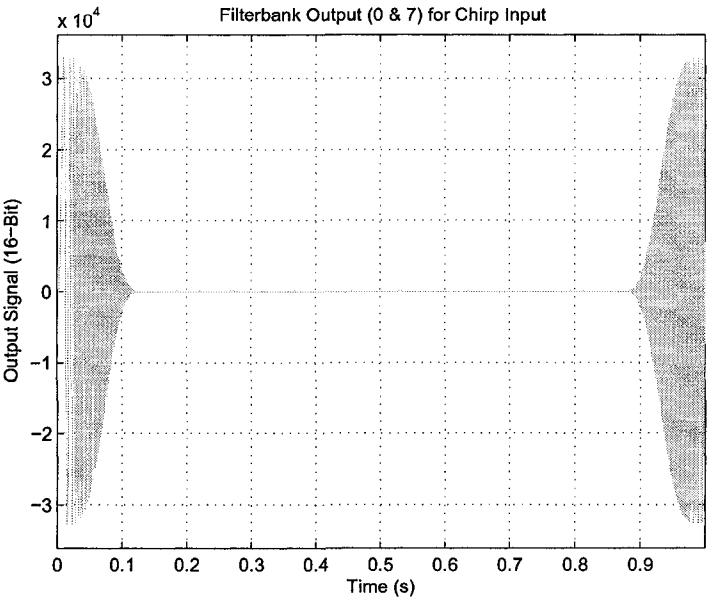

Figure 4.2: The Filterbank Output of Filters 0 and 7

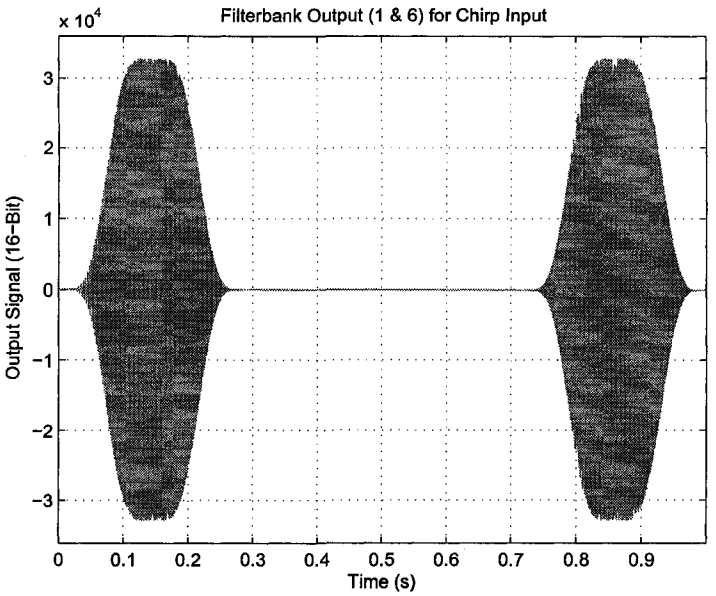

Figure 4.3: The Filterbank Output of Filters 1 and 6

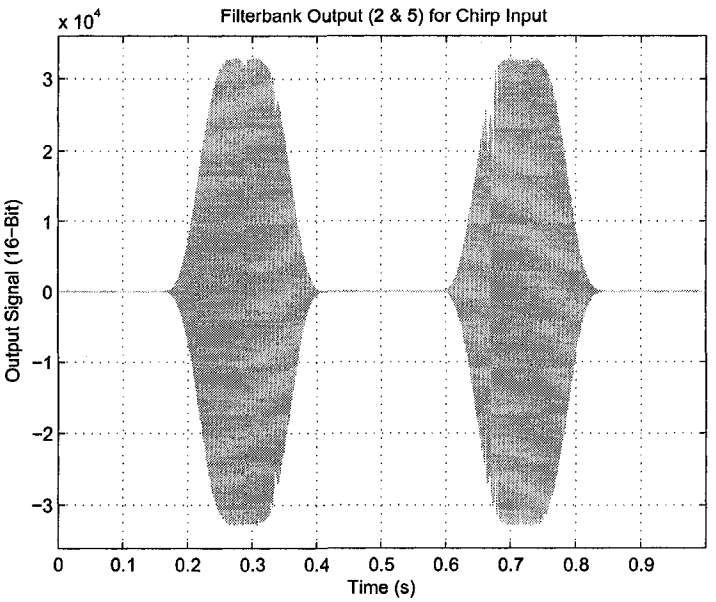

Figure 4.4: The Filterbank Output of Filters 2 and 5

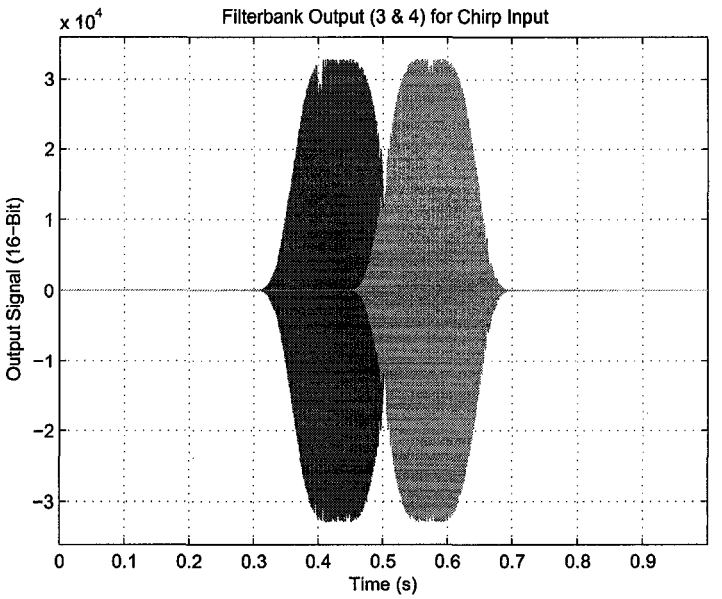

Figure 4.5: The Filterbank Output of Filters 3 and 4

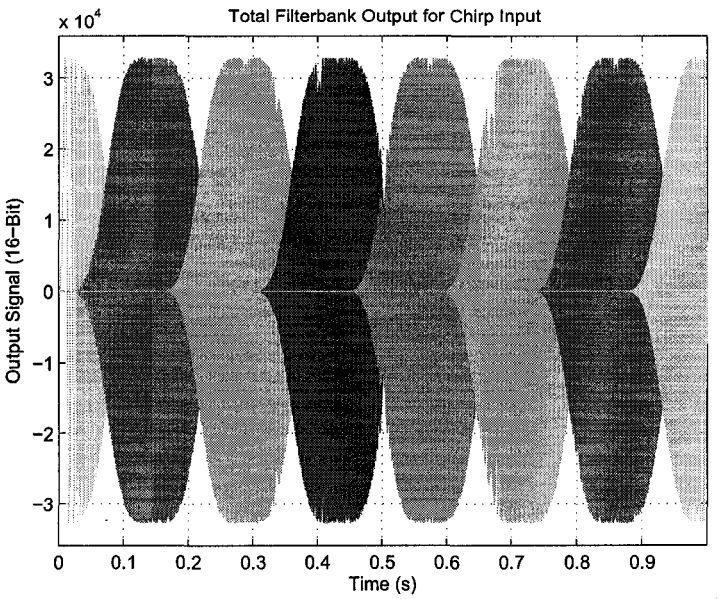

Figure 4.6: The Filterbank Output of all filters

all CPU components, particularly including conversion tables, the results can not be compared properly. Therefore, the table 4.2 only shows the synthesis results of TLNS CPU for two cases, optimization for area and when a 10 MHz clock-frequency has been specified. In both cases, the total number of ports is 154.

Specification	Optimized for area	Clock-frequency 10 MHz
Number of Nets	5406	5875
Number of Cells	4829	5146
Combinational Area $(\mu m)^2$	306686.47	497615.16
Noncombinational Area $(\mu m)^2$	79811.84	80425.82
Total Cell Area $(\mu m)^2$	386501.13	578064.94

Table 4.2: TLNS Synthesis Results

Nevertheless, a comparison between the number of necessary clock cycles just for filtering operation is possible. In filterbank custom design with the same design specifications [12], filtering is accomplished in 313 clock cycles and in improved filtarbank design [13], it has been reduced to 300 clock cycles. As it is been stated in table 4.1, filtering operation in TLNS filterbank application is performed in 328 clock cycles. It worth to mention that, the objective of this design is just to implement an appropriate potential application on the TLNS CPU. Therefore, this implementation has not been optimized and some recommendations for future work are suggested in next chapter to improve and speed up this application.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this research work a 2DLNS processor has been developed. This CPU will have some applications in the practical implementation of most of DSP algorithms. Another application of this CPU is in any other research work based on MDLNS. The CPU architecture has been designed based on its 2DLNS characteristics and its potential applications.

In order to maintain consistency, the CPU interface with external world has been considered through binary ports. The organization of CPU components has been developed based on the most profitable operation in 2DLNS, which is multiplication. Therefore, in addition to general features, some special components have been also implemented. Since TLNS performs addition and subtraction in binary representation, two special features for converting data between binary and 2DLNS have been included.

In the TLNS processor organization, two separate memories have been considered, both of which can also be addressed by the CPU controller. By the virtue of having two separate data buses, data can be read from both memories concurrently, which makes TLNS CPU suitable for some real time DSP applications. The TLNS CPU has been tested with several programs including different sets of instructions.

After reviewing past and improved MDLNS filterbank designs, a TLNS program has been coded to implement a filterbank application on the CPU. The same specifications for the filterbank design have been considered, therefore some of their ancillary codes could be used. The program has been written in considerably less time, without any need to know either HDL programming or details about MDLNS. Programming the TLNS CPU, like any other assembly language programming, requires a preliminary knowledge of the CPU Instruction Set. The functional results of this program are similar to those of the original filterbank design.

5.2 Suggestions for Future Work

The implemented filterbank uses FIR filters. There is a possibility to improve this design, in order to use Infinite Impulse Response (IIR) filters. An IIR filter uses past outputs to influence the current response of the filter. Recursive filters usually require a much lower order of filter to produce the same magnitude response of a FIR filter, but IIR filters are not guaranteed to be stable or have a linear phase. The output response, $y(nT)$, of a recursive filter can be described by the difference equation:

$$y(nT) = \sum_{i=0}^{N} a_i.x(nT - iT) - \sum_{i=1}^{M} b_i.y(nT - iT)$$

where a_i and b_i represent constants. Therefore, at any given moment, the response of the filter is dependent on the past N values of the input and the past M values of the output. The first term is the same as FIR filter equation, which can be realized using the TLNS MAC unit. The last M values of output can also be stored in memory. Thus, by considering a special subtracter, or MDLNS divider, in the CPU data path, an IIR filter could be implemented.

Some compound instructions can be added to the Instruction Set. At present, there are separate instructions for entering data to the CPU, converting it to 2DLNS representation, and storing a word in memory. In order to speed up applications which require these instructions consecutively, some instructions like "input and convert", or "convert and store", or even "input and convert and store" might be developed.

One of the computational advantages of MDLNS is calculating the square root of numbers. In order to calculate the square root of a 2DLNS number, all exponents are simply divided by 2. A comparison with the volume of work for the same calculation in binary based CPUs makes this advantage clear. As a suggestion, a special instruction can be added to the CPU Instruction Set in order to calculate the square root.

Another potential improvement is implementing some more special purpose instructions. This CPU may have some applications in modular exponentiation. Therefore, any instructions which ease corresponding operation may be added to the TLNS Instruction Set.

And Finally, the controller state machine may be improved. The controller procedures include micro codes to perform instructions. These microcodes may be opti-

mized to reduce the number of states for each instruction execution and as a result, increase the speed of processing.

References

[1] P. J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers, San Diego, 2002.

[2] J. D. Carpinelli. *Computer Systems Organization and Architecture.* Addison Wesley, 2001.

[3] V. S. Dimitrov and G. A. Jullien. A New Number Representation with Applications. *IEEE Circuits and Systems Magazine*, Second Quarter:6–23, 2003.

[4] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. An Algorithm for Modular Exponentiation. In *Information Processing Letters*, volume 36, pages 155–159, May 1998.

[5] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. Theory and Applications of the Double-Base Number System. *IEEE Transactions on Computers*, 48(10):1098–1106, October 1999.

[6] V. S. Dimitrov, G. A. Jullien, and K. Walus. Digital filtering using the multidimensional logarithmic number system. *Advanced Signal Processing Algorithms, Architectures, and Implementations XII, Proceedings of the SPIE*, 4791:412–423, December 2002.

[7] V. S. Dimitrov, S. Sadeghi-Emamchaie, G. A. Jullien, and W. C. Miller. A Near Canonical Double-Base Number System with Applications in DSP. In *SPIE Conference on Signal Processing Algorithms*, volume 2846, pages 14–25, 1996.

[8] S. J. Eskritt. *Inner Product Computational Architectures Using the Double Base Number System.* M.A.Sc. Thesis,, University of Windsor, 2001.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Fransisco, 1995.

[10] N. G. Kingsbury and P. J. Rayner. Digital Filtering Using Logarithmic Arithmetic. *Electronics Letters*, 7:56–58, 1971.

[11] D. M. Lewis. 114 MFLOPS Logarithmic Number System Arithmetic unit for DSP Applications. *IEEE J. Solid-State Circuits*, 30:1547–1553, 1995.

[12] H. Li. *A 2-digit Multi-dimensional Logarithmic Number System Filterbank Processor for a Digital Hearing Aid.* M.A.Sc. Thesis,, University of Windsor, 2003.

[13] R. Muscedere. *Difficult Operations in the Multi-Dimensional Logarithmic Number System.* Ph.D. Thesis,, University of Windsor, 2003.

[14] R. Muscedere, G. A. Jullien V. S. Dimitrov, and W. C. Miller. Efficient Conversion From Binary to Multi-Digit Multi-Dimensional Logarithmic Number Systems using Arrays of Range Addressable Look-Up Tables. *Proceedings of the 2002 IEEE conference on Application-Specific Systems, Architectures, and Processors*, pages 130–138, 2002.

[15] R. Muscedere, V. Dimitrov, G. Jullien, and W. Miller. A Low-Power Two-Digit Multi-dimensional Logarithmic Number System Filterbank Architecture for a Digital Hearing Aid. *EURASIP Journal on Applied Signal Processing*, 18:3015–3025, 2005.

[16] R. Muscedere, V. S. Dimitrov, G. A. Jullien, and W. C. Miller. Efficient Techniques for binary-to-multidigit multidimensional logarithmic number system conversion using range addressable look-up tables. *IEEE Trans. Comput. Special Issue on Computer Arithmetic*, 54(3):257–271, 2005.

[17] E. Onat. *DSP Algorithms for Digital Hearing Instruments.* M.A.Sc. Thesis,, University of Windsor, 2001.

[18] E. E. Swartzlander and A. G. Alexopoulos. The Sign/Logarithm Number System. *IEEE Transactions on Computers*, 42:1238–1242, 1975.

[19] F. J. Taylor, R. Gill, J. Joseph, and J. Radke. A 20 Bit Logarithmic Number System Processor. *IEEE Transactions on Computers*, 37:190–200, 1988.

Appendix A

Hardware Description Codes

A.1 TLNS Packages

A VHDL package contains subprograms, constant definitions, and/or type definitions to be used through one or more design units. TLNS makes use of some IEEE standard packages as well as a few special user defined packages which have been written particularly for this program.

VHDL standard packages include **STANDARD** package which contains basic type definitions and **TEXTIO** package which regards to ASCII input/output data types and subprograms. These packages are fully standard and not described here. In addition, another IEEE package, **numeric_bit** has been used in this program. This package defines numeric types and arithmetic functions for use with synthesis tools. In this research work, the original version of this package, **Standard VHDL Synthesis Package (1076.3, NUMERIC_BIT)**, has been used with some minor changes. Since the package code is too long to be included here, just the modified portions

are described. This appendix also contains VHDL codes for other packages which are specifically written for this CPU and used by all other modules.

A.1.1 The TLNS Types Package

This package defines the types for the CPU ports and some other types that are useful in this design.

```
library ieee;
use ieee.std_logic_1164.all;

use work.numeric_bit.all;

package tlns_types is

  -- little-endian addresses
  subtype tlns_address is unsigned(23 downto 0);

  -- big-endian data words
  subtype tlns_word is unsigned(23 downto 0);

  type tlns_word_array is array (natural range <>) of tlns_word;

  -- word for driving a bus
  subtype tlns_bus_word is std_logic_vector(23 downto 0);
  subtype mem_bus_addr is std_logic_vector(9 downto 0);

  subtype external_data is std_logic_vector(15 downto 0);

  type tlns_bus_word_array is array (natural range <>) of tlns_bus_word;

  -- tristate bus driving value
  constant disabled_tlns_word : tlns_bus_word := ( others => 'Z' );
  constant disabled_mem_addr : mem_bus_addr := ( others => 'Z' );

end package tlns_types;
```

A.1.2 The TLNS Instruction Set Package

This package represents the details of TLNS instruction set. These details include type definitions and constant encoding specifications.

```
use work.numeric_bit.all,
    work.tlns_types.all;

package tlns_instr is

  -- bit-vector types for the fields in an instruction
  subtype tlns_opcode is unsigned(5 downto 0);
  subtype tlns_sp_func is unsigned(5 downto 0);
  subtype tlns_reg_addr is unsigned(3 downto 0);
  subtype tlns_immed10 is unsigned(9 downto 0);
  subtype tlns_immed18 is unsigned(17 downto 0);
  subtype tlns_shamt is unsigned(3 downto 0);
  subtype tlns_addr is unsigned(9 downto 0);

  -- defining the opcode values for instructions
  constant op_special   : tlns_opcode := B"000000";
  constant op_tbc       : tlns_opcode := B"000001";
  constant op_j         : tlns_opcode := B"000010";
  constant op_jal       : tlns_opcode := B"000011";
  constant op_beqz      : tlns_opcode := B"000100";
  constant op_bnez      : tlns_opcode := B"000101";
  constant op_btc       : tlns_opcode := B"000110";
  constant op_filter    : tlns_opcode := B"000111";
  constant op_addi      : tlns_opcode := B"001000";
  constant op_addui     : tlns_opcode := B"001001";
  constant op_subi      : tlns_opcode := B"001010";
  constant op_subui     : tlns_opcode := B"001011";
  constant op_andi      : tlns_opcode := B"001100";
  constant op_ori       : tlns_opcode := B"001101";
  constant op_xori      : tlns_opcode := B"001110";
  constant op_lhi       : tlns_opcode := B"001111";

  constant op_inpt      : tlns_opcode := B"010000";
  constant op_oupt      : tlns_opcode := B"010001";
  constant op_jr        : tlns_opcode := B"010010";
  constant op_jalr      : tlns_opcode := B"010011";
  constant op_slli      : tlns_opcode := B"010100";
  constant op_filt      : tlns_opcode := B"010101";
  constant op_srli      : tlns_opcode := B"010110";
  constant op_srai      : tlns_opcode := B"010111";
  constant op_seqi      : tlns_opcode := B"011000";
  constant op_snei      : tlns_opcode := B"011001";
```

```
constant op_slti       : tlns_opcode := B"011010";
constant op_sgti       : tlns_opcode := B"011011";
constant op_slei       : tlns_opcode := B"011100";
constant op_sgei       : tlns_opcode := B"011101";
constant op_undef_1E   : tlns_opcode := B"011110";
constant op_undef_1F   : tlns_opcode := B"011111";

constant op_undef_20   : tlns_opcode := B"100000";
constant op_undef_21   : tlns_opcode := B"100001";
constant op_undef_22   : tlns_opcode := B"100010";
constant op_lw         : tlns_opcode := B"100011";
constant op_undef_24   : tlns_opcode := B"100100";
constant op_undef_25   : tlns_opcode := B"100101";
constant op_undef_26   : tlns_opcode := B"100110";
constant op_undef_27   : tlns_opcode := B"100111";
constant op_undef_28   : tlns_opcode := B"101000";
constant op_undef_29   : tlns_opcode := B"101001";
constant op_undef_2A   : tlns_opcode := B"101010";
constant op_sw         : tlns_opcode := B"101011";
constant op_undef_2C   : tlns_opcode := B"101100";
constant op_undef_2D   : tlns_opcode := B"101101";
constant op_undef_2E   : tlns_opcode := B"101110";
constant op_undef_2F   : tlns_opcode := B"101111";

constant op_sequi      : tlns_opcode := B"110000";
constant op_sneui      : tlns_opcode := B"110001";
constant op_sltui      : tlns_opcode := B"110010";
constant op_sgtui      : tlns_opcode := B"110011";
constant op_sleui      : tlns_opcode := B"110100";
constant op_sgeui      : tlns_opcode := B"110101";
constant op_undef_36   : tlns_opcode := B"110110";
constant op_undef_37   : tlns_opcode := B"110111";
constant op_undef_38   : tlns_opcode := B"111000";
constant op_undef_39   : tlns_opcode := B"111001";
constant op_undef_3A   : tlns_opcode := B"111010";
constant op_undef_3B   : tlns_opcode := B"111011";
constant op_undef_3C   : tlns_opcode := B"111100";
constant op_undef_3D   : tlns_opcode := B"111101";
constant op_multi      : tlns_opcode := B"111110";
constant op_divi       : tlns_opcode := B"111111";

constant sp_func_nop        : tlns_sp_func := B"000000";
constant sp_func_halt       : tlns_sp_func := B"000001";
constant sp_func_mult       : tlns_sp_func := B"000010";
constant sp_func_div        : tlns_sp_func := B"000011";
constant sp_func_sll        : tlns_sp_func := B"000100";
constant sp_func_undef_05   : tlns_sp_func := B"000101";
constant sp_func_srl        : tlns_sp_func := B"000110";
constant sp_func_sra        : tlns_sp_func := B"000111";
```

```
constant sp_func_undef_08  : tlns_sp_func := B"001000";
constant sp_func_undef_09  : tlns_sp_func := B"001001";
constant sp_func_undef_0A  : tlns_sp_func := B"001010";
constant sp_func_undef_0B  : tlns_sp_func := B"001011";
constant sp_func_undef_0C  : tlns_sp_func := B"001100";
constant sp_func_undef_0D  : tlns_sp_func := B"001101";
constant sp_func_undef_0E  : tlns_sp_func := B"001110";
constant sp_func_undef_0F  : tlns_sp_func := B"001111";

constant sp_func_sequ      : tlns_sp_func := B"010000";
constant sp_func_sneu      : tlns_sp_func := B"010001";
constant sp_func_sltu      : tlns_sp_func := B"010010";
constant sp_func_sgtu      : tlns_sp_func := B"010011";
constant sp_func_sleu      : tlns_sp_func := B"010100";
constant sp_func_sgeu      : tlns_sp_func := B"010101";
constant sp_func_undef_16  : tlns_sp_func := B"010110";
constant sp_func_undef_17  : tlns_sp_func := B"010111";
constant sp_func_undef_18  : tlns_sp_func := B"011000";
constant sp_func_undef_19  : tlns_sp_func := B"011001";
constant sp_func_undef_1A  : tlns_sp_func := B"011010";
constant sp_func_undef_1B  : tlns_sp_func := B"011011";
constant sp_func_undef_1C  : tlns_sp_func := B"011100";
constant sp_func_undef_1D  : tlns_sp_func := B"011101";
constant sp_func_undef_1E  : tlns_sp_func := B"011110";
constant sp_func_undef_1F  : tlns_sp_func := B"011111";

constant sp_func_add       : tlns_sp_func := B"100000";
constant sp_func_addu      : tlns_sp_func := B"100001";
constant sp_func_sub       : tlns_sp_func := B"100010";
constant sp_func_subu      : tlns_sp_func := B"100011";
constant sp_func_and       : tlns_sp_func := B"100100";
constant sp_func_or        : tlns_sp_func := B"100101";
constant sp_func_xor       : tlns_sp_func := B"100110";
constant sp_func_undef_27  : tlns_sp_func := B"100111";
constant sp_func_seq       : tlns_sp_func := B"101000";
constant sp_func_sne       : tlns_sp_func := B"101001";
constant sp_func_slt       : tlns_sp_func := B"101010";
constant sp_func_sgt       : tlns_sp_func := B"101011";
constant sp_func_sle       : tlns_sp_func := B"101100";
constant sp_func_sge       : tlns_sp_func := B"101101";
constant sp_func_undef_2E  : tlns_sp_func := B"101110";
constant sp_func_undef_2F  : tlns_sp_func := B"101111";

constant sp_func_undef_30  : tlns_sp_func := B"110000";
constant sp_func_undef_31  : tlns_sp_func := B"110001";
constant sp_func_undef_32  : tlns_sp_func := B"110010";
constant sp_func_undef_33  : tlns_sp_func := B"110011";
constant sp_func_undef_34  : tlns_sp_func := B"110100";
constant sp_func_undef_35  : tlns_sp_func := B"110101";
```

```
  constant sp_func_undef_36  : tlns_sp_func := B"110110";
  constant sp_func_undef_37  : tlns_sp_func := B"110111";
  constant sp_func_undef_38  : tlns_sp_func := B"111000";
  constant sp_func_undef_39  : tlns_sp_func := B"111001";
  constant sp_func_undef_3A  : tlns_sp_func := B"111010";
  constant sp_func_undef_3B  : tlns_sp_func := B"111011";
  constant sp_func_undef_3C  : tlns_sp_func := B"111100";
  constant sp_func_undef_3D  : tlns_sp_func := B"111101";
  constant sp_func_undef_3E  : tlns_sp_func := B"111110";
  constant sp_func_undef_3F  : tlns_sp_func := B"111111";

  -- numeric values for register numbers
  subtype reg_index is natural range 0 to 15;

  constant output_reg_1 : reg_index := 12;
  constant output_reg_2 : reg_index := 13;
  constant unity_reg : reg_index := 14;
  constant link_reg : reg_index := 15;

end package tlns_instr;
```

A.1.3 The TLNS ALU Types Package

The particular function to be performed by the ALU at any time is determined by the controller. Hence, the ALU has an input port to select the function. The type and allowable values for this port are described in a separate package, so they are accessible both in the ALU and the controller descriptions.

```
package alu_types is

  subtype alu_func is bit_vector(3 downto 0);

  -- encoded values for the ALU function
  constant alu_add :     alu_func := "0000";
  constant alu_addu :    alu_func := "0001";
  constant alu_sub :     alu_func := "0010";
  constant alu_subu :    alu_func := "0011";
  constant alu_and :     alu_func := "0100";
  constant alu_or :      alu_func := "0101";
  constant alu_xor :     alu_func := "0110";
  constant alu_sll :     alu_func := "1000";
  constant alu_srl :     alu_func := "1001";
  constant alu_sra :     alu_func := "1010";
```

```
  -- identity operations
  constant alu_pass_s1 : alu_func := "1100";
  constant alu_pass_s2 : alu_func := "1101";

end package alu_types;
```

A.1.4 The NUMERIC_BIT Package

The following VHDL file shows the differences between modified **numeric_bit** package and its original copy. This file clearly shows the lines which have been changed or added to the package. Most of these changes return back to initial values of variables. In whole package, initial values have been removed from declaration statements and have been assigned to the variables later. The definition of functions RISING_EDGE and FALLING_EDGE have had conflicts with other standard libraries. Therefore, the defined functions in this package have been ignored. Some other changes are just minor corrections regard to VHDL syntax. The core shift functions, XSLL and XSRL, have been modified based on the TLNS requirements. In this regard, the code for a barrel shifter has been entirely rewritten. Finally, the function RESIZE has been changed due to some errors.

```
801,805c801,805
<   -- Id: E.1
<   function RISING_EDGE (signal S: BIT) return BOOLEAN;
<   -- Result subtype: BOOLEAN
<   -- Result: Returns TRUE if an event is detected on signal S and the
<   --         value changed from a '0' to a '1'.
---
>   -- Id: E.1
>   function RISING_EDGE (signal S: BIT) return BOOLEAN;
>   -- Result subtype: BOOLEAN
>   -- Result: Returns TRUE if an event is detected on signal S and
    the
>   --         value changed from a '0' to a '1'.
807,811c807,811
<   -- Id: E.2
<   function FALLING_EDGE (signal S: BIT) return BOOLEAN;
```

```
<   -- Result subtype: BOOLEAN
<   -- Result: Returns TRUE if an event is detected on signal S and the
<   --         value changed from a '1' to a '0'.
---
> --  -- Id: E.2
> --  function FALLING_EDGE (signal S: BIT) return BOOLEAN;
> --  -- Result subtype: BOOLEAN
> --  -- Result: Returns TRUE if an event is detected on signal S and
    the
> --  --         value changed from a '1' to a '0'.
823,824c823,824
<   constant NAU: UNSIGNED(0 downto 1) := (others => '0');
<   constant NAS: SIGNED(0 downto 1) := (others => '0');
---
>   constant NAU: UNSIGNED(0 to 1) := (others => '0');
>   constant NAS: SIGNED(0 to 1) := (others => '0');
886c886
<     variable CBIT: BIT := C;
---
>     variable CBIT: BIT; --:= C;
887a888
>     CBIT := C;
904c905
<     variable CBIT: BIT := C;
---
>     variable CBIT: BIT; -- := C;
905a907
>     CBIT := C;
952,953c954,960
<     alias XARG: BIT_VECTOR(ARG_L downto 0) is ARG;
<     variable RESULT: BIT_VECTOR(ARG_L downto 0) := (others => '0');
---
>     variable RESULT: BIT_VECTOR(ARG_L downto 0);
>     variable temp: integer;
>     variable arg_1: bit_vector (ARG_L downto 0);
>     variable arg_12: bit_vector (ARG_L downto 0);
>     variable arg_124: bit_vector (ARG_L downto 0);
>     variable shift: bit_vector(3 downto 0);
>
955,956c962,975
<     if COUNT <= ARG_L then
<       RESULT(ARG_L downto COUNT) := XARG(ARG_L-COUNT downto 0);
---
>
>     temp := count;
>     for index in 0 to 3 loop
>       if  (temp rem 2) = 0 then shift(index) := '0';
>                            else shift(index) := '1';
>       end if;
```

```
>          temp := temp / 2 ;
>      end loop;
>
>      if shift(0) = '0' then arg_1 := arg ;
>                        else arg_1 := arg(ARG_L-1 downto 0) & "0" ;
>      end if;
>      if shift(1) = '0' then arg_12 := arg_1 ;
>                        else arg_12 := arg_1(ARG_L-2 downto 0) & "00" ;
957a977,983
>      if shift(2) = '0' then arg_124 := arg_12 ;
>                        else arg_124 := arg_12(ARG_L-4 downto 0) & "0000"
    ;
>      end if;
>      if shift(3) = '0' then RESULT := arg_124 ;
>                          else RESULT := arg_124(ARG_L-8 downto 0) &
                                           "00000000" ;
>      end if;
>
963,964c989,995
<      alias XARG: BIT_VECTOR(ARG_L downto 0) is ARG;
<      variable RESULT: BIT_VECTOR(ARG_L downto 0) := (others => '0');
---
>      variable RESULT: BIT_VECTOR(ARG_L downto 0);
>      variable temp: integer;
>      variable arg_1: bit_vector (ARG_L downto 0);
>      variable arg_12: bit_vector (ARG_L downto 0);
>      variable arg_124: bit_vector (ARG_L downto 0);
>      variable shift: bit_vector(3 downto 0);
>
966,967c997,1010
<      if COUNT <= ARG_L then
<        RESULT(ARG_L-COUNT downto 0) := XARG(ARG_L downto COUNT);
---
>
>      temp := count;
>      for index in 0 to 3 loop
>        if  (temp rem 2) = 0 then shift(index) := '0';
>                             else shift(index) := '1';
>        end if;
>          temp := temp / 2 ;
>      end loop;
>
>      if shift(0) = '0' then arg_1 := arg ;
>                        else arg_1 := "0" & arg(ARG_L downto 1) ;
>      end if;
>      if shift(1) = '0' then arg_12 := arg_1 ;
>                        else arg_12 := "00" & arg_1(ARG_L downto 2) ;
968a1012,1018
>      if shift(2) = '0' then arg_124 := arg_12 ;
```

```
>                       else arg_124 := "0000" & arg_12(ARG_L downto 4) ;
>     end if;
>     if shift(3) = '0' then RESULT := arg_124 ;
>                       else RESULT := "00000000" &
                                       arg_124(ARG_L downto 8) ;
>     end if;
>
976c1026
<     variable XCOUNT: NATURAL := COUNT;
---
>     variable XCOUNT: NATURAL; -- := COUNT;
977a1028
>     XCOUNT := COUNT;
991c1042
<     variable RESULT: BIT_VECTOR(ARG_L downto 0) := XARG;
---
>     variable RESULT: BIT_VECTOR(ARG_L downto 0); --:= XARG;
993a1045
>     RESULT := XARG;
1005c1057
<     variable RESULT: BIT_VECTOR(ARG_L downto 0) := XARG;
---
>     variable RESULT: BIT_VECTOR(ARG_L downto 0); -- := XARG;
1007a1060
>     RESULT := XARG;
1100c1153
<     variable CBIT: BIT := '1';
---
>     variable CBIT: BIT; --:= '1';
1101a1155
>     CBIT := '1';
1219c1273
<     variable RESULT: UNSIGNED((L'LENGTH+R'LENGTH-1) downto 0) :=
                       (others => '0');
---
>     variable RESULT: UNSIGNED((L'LENGTH+R'LENGTH-1) downto 0);
                       -- := (others => '0');
1221a1276
>     RESULT := (others => '0');
1239c1294
<     variable RESULT: SIGNED((L_LEFT+R_LEFT+1) downto 0) :=
                       (others => '0');
---
>     variable RESULT: SIGNED((L_LEFT+R_LEFT+1) downto 0);
                       -- := (others => '0');
1241a1297
>     RESULT := (others => '0');
1301c1357
<     variable QNEG: BOOLEAN := FALSE;
```

```
---
>       variable QNEG: BOOLEAN; --:= FALSE;
1302a1359
>       QNEG := FALSE;
1353a1411
>       null;
1387a1446
>       null;
1411c1470
<       variable RNEG: BOOLEAN := FALSE;
---
>       variable RNEG: BOOLEAN; --:= FALSE;
1412a1472
>       RNEG := FALSE;
1446a1507
>       null;
1482a1544
>       null;
1500a1563
>       null;
1524c1587
<       variable RNEG: BOOLEAN := FALSE;
---
>       variable RNEG: BOOLEAN; -- := FALSE;
1525a1589
>       RNEG := FALSE;
1581a1646
>       null;
1599a1665
>       null;
1617a1684
>       null;
1626c1693
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1627a1695
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1639c1707
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1640a1709
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1710c1779
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1711a1781
```

```
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1723c1793
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1724a1795
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1794c1865
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1795a1867
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1807c1879
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1808a1881
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1878c1951
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1879a1953
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1891c1965
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1892a1967
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1962c2037
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1963a2039
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1975c2051
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1976a2053
>       SIZE := MAX(L'LENGTH, R'LENGTH);
2046c2123
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
2047a2125
>       SIZE := MAX(L'LENGTH, R'LENGTH);
```

```
2059c2137
<      variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>      variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
2060a2139
>      SIZE := MAX(L'LENGTH, R'LENGTH);
2314c2393
<      variable RESULT: NATURAL := 0;
---
>      variable RESULT: NATURAL; -- := 0;
2315a2395
>      RESULT := 0;
2350c2430
<      variable I_VAL: NATURAL := ARG;
---
>      variable I_VAL: NATURAL; -- := ARG;
2351a2432
>      I_VAL := ARG;
2364a2446
>      null;
2372,2373c2454,2455
<      variable B_VAL: BIT := '0';
<      variable I_VAL: INTEGER := ARG;
---
>      variable B_VAL: BIT; -- := '0';
>      variable I_VAL: INTEGER; -- := ARG;
2374a2457,2458
>      B_VAL := '0';
>      I_VAL := ARG;
2392a2477
>      null;
2402c2487
<      variable RESULT: SIGNED(NEW_SIZE-1 downto 0) := (others => '0');
---
>      variable RESULT: SIGNED(NEW_SIZE-1 downto 0); -- := (others =>
    '0');
2404a2490
>      RESULT := (others => '0');
2418,2420c2504,2506
<      constant ARG_LEFT: INTEGER := ARG'LENGTH-1;
<      alias XARG: UNSIGNED(ARG_LEFT downto 0) is ARG;
<      variable RESULT: UNSIGNED(NEW_SIZE-1 downto 0) := (others => '0');
---
>      alias INVEC: UNSIGNED(ARG'LENGTH-1 downto 0) is ARG;
>      variable RESULT: UNSIGNED(NEW_SIZE-1 downto 0);
                        -- := (others => '0');
>      constant BOUND: INTEGER := MIN(ARG'LENGTH, RESULT'LENGTH)-1;
2421a2508
>      RESULT := (others => '0');
```

```
2424c2511
<      if XARG'LENGTH =0 then return RESULT;
---
>      if (ARG'LENGTH =0) then return RESULT;
2426,2430c2513,2514
<      if (RESULT'LENGTH < ARG'LENGTH) then
<        RESULT(RESULT'LEFT downto 0) := XARG(RESULT'LEFT downto 0);
<      else
<        RESULT(RESULT'LEFT downto XARG'LEFT+1) := (others => '0');
<        RESULT(XARG'LEFT downto 0) := XARG;
---
>      if BOUND >= 0 then
>        RESULT(BOUND downto 0) := INVEC(BOUND downto 0);
2434c2518
<
---
>
2559,2563c2643,2651
<    -- Id: E.1
<    function RISING_EDGE (signal S: BIT) return BOOLEAN is
<    begin
<      return S'EVENT and S = '1';
<    end RISING_EDGE;
---
> --  -- Id: E.1
> --  function RISING_EDGE (signal S: BIT) return BOOLEAN is
> --  begin
> --        return S'EVENT and S = '1';
> --    if (S'EVENT and S = '1')
> --    then  return true;
> --    else return  false;
> --    end if;
> --  end RISING_EDGE;
2565,2569c2653,2657
<    -- Id: E.2
<    function FALLING_EDGE (signal S: BIT) return BOOLEAN is
<    begin
<      return S'EVENT and S = '0';
<    end FALLING_EDGE;
---
> --  -- Id: E.2
> --  function FALLING_EDGE (signal S: BIT) return BOOLEAN is
> --  begin
> --    return S'EVENT and S = '0';
> --  end FALLING_EDGE;
```

A.2 The TLNS CPU Modules

The TLNS CPU data path consists of several features. All features are declared as components in the CPU top level file. The RTL architecture body of the CPU is constructed using these all data path components.

A.2.1 The TLNS CPU

The VHDL description of "tlns" entity and its RTL level architecture body is shown in this section. This is the top level file in CPU organization. All CPU components are declared and instantiated in this file.

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.tlns_instr.all,
    work.alu_types.all,
    work.numeric_bit.all,
    work.tlns_types.all;

entity tlns is
  port ( clk : in std_logic;
         reset : in std_logic;
         halt : out std_logic;
         input_data : in external_data;
         output_data : out tlns_bus_word;
         a : out tlns_bus_word;
         d : in tlns_bus_word;
         ifetch : out std_logic;
         ir_mem_enable : out std_logic;
         mem_a : out mem_bus_addr;
         mem_d_out : in tlns_bus_word;
         mem_d_in : out tlns_bus_word;
         mem_write_en : out std_logic;
         output_enable : out std_logic;
         mem_enable : out std_logic);
end entity tlns;


architecture rtl of tlns is
```

```
-- The TLNS Output Register
component output_reg is
  port (clk : in std_logic;
        d : in tlns_bus_word;
        q : out tlns_bus_word;
        out_enable : out std_logic;
        enable : in std_logic);
end component output_reg;

-- The TLNS Input Register
component input_reg is
  port (d : in external_data;
        q : out tlns_bus_word;
        enable : in std_logic;
        out_en : in std_logic);
end component input_reg;

-- The TLNS Binary / 2DLNS Converter
component  serial2digithighlow is
  port (CK : in std_logic;
        reset : in std_logic;
        activate : in std_logic;
        i : in std_logic_vector(15 downto 0);
        ready : out std_logic;
        output_sign : out std_logic_vector(1 downto 0);
        output_first : out std_logic_vector(11 downto 0);
        output_second : out std_logic_vector(9 downto 0));
end component;

-- The TLNS Binary / 2DLNS Conversion Register
component conv_out_reg is
  port ( ready : in std_logic;
         output_sign : in std_logic_vector(1 downto 0);
         output_first : in std_logic_vector(11 downto 0);
         output_second : in std_logic_vector(9 downto 0);
         tlns_output : out tlns_bus_word);
end component;

-- The TLNS Multiply and Accumulate unit
component mac is
  port (clk,clr : in std_logic;
        channel_mux_sel : in std_logic;
        coefnum : in std_logic;
        evensym : in std_logic;
        x : in tlns_bus_word;
        y : in tlns_bus_word;
        p : out tlns_bus_word);
end component mac;
```

```
-- The TLNS Arithmetic and Logic Unit
component alu is
  port ( s1 : in tlns_word;
         s2 : in tlns_word;
         result : out tlns_bus_word;
         func : in alu_func;
         zero, negative, overflow : out bit;
         clr : in  std_logic );
end component alu;

-- The TLNS Register File
component reg_file is
  port ( clk : in std_ulogic;
         a1 : in tlns_reg_addr;
         q1 : out tlns_word;
         a2 : in tlns_reg_addr;
         q2 : out tlns_word;
         a3 : in tlns_reg_addr;
         d3 : in tlns_bus_word;
         write_en : in std_logic );
end component reg_file;

-- The TLNS Extender
component ir_extender is
  port ( d : in tlns_word;
         q : out tlns_bus_word;
         immed_size_18 : in bit;
         immed_unsigned : in bit;
         immed_en : in bit );
end component ir_extender;

-- The TLNS Director / Extender
component ir_extender_director is
  port ( d : in tlns_word;
         q : out tlns_bus_word;
         immed_size_18 : in bit;
         immed_unsigned : in bit;
         immed_en : in bit;
         direct : in bit );
end component ir_extender_director;

-- The TLNS A and B registers
component reg is
  port ( d : in tlns_word;
         q : out tlns_bus_word;
         clk, enable : in std_logic;
         out_en : in std_logic );
end component reg;
```

```
-- The TLNS Program Counter
component reg_multiple_plus_one_out_reset is
  port ( d : in tlns_bus_word;
         q0 : out tlns_word;
         q : out tlns_bus_word;
         clk, enable : in std_logic;
         out_en : in std_logic;
         reset : in std_logic );
end component reg_multiple_plus_one_out_reset;

-- The TLNS Memory Address Register
component mar_reg is
  port ( d : in tlns_bus_word;
         q : out tlns_bus_word;
         clk, enable : in std_logic );
end component mar_reg;

-- The TLNS Instruction Memory Multiplexer
component mux2 is
  port ( i0 : in tlns_bus_word;
         i1 : in tlns_bus_word;
         y : out tlns_bus_word;
         sel : in bit);
end component mux2;

-- The TLNS Data Memory Multiplexer
component mux2_ma is
  port ( i0 : in tlns_bus_word;
         i1 : in tlns_address;
         y : out tlns_bus_word;
         sel : in bit);
end component mux2_ma;

-- The TLNS Controller
component controller is
  port ( clk : in std_logic;
         reset : in std_logic;
         halt : out std_logic;
         ir_mem_enable : out std_logic;
         ifetch : out std_logic;
         alu_function : out alu_func;
         alu_zero, alu_negative, alu_overflow : in bit;
         reg_s1_addr, reg_s2_addr, reg_dest_addr : out tlns_reg_addr;
         reg_write : out std_logic;
         a_enable : out std_logic;
         a_out_en : out std_logic;
         b_enable : out std_logic;
         b_out_en : out std_logic;
         pc_enable : out std_logic;
```

```
            pc_out_en : out std_logic;
            mar_enable : out std_logic;
            ir_immed1_size_18, ir_immed2_size_18 : out bit;
            ir_immed1_unsigned, ir_immed2_unsigned : out bit;
            ir_immed1_en, ir_immed2_en : out bit;
            current_instruction : in tlns_word;
            const2 : out tlns_bus_word;
            mem_write_en : out std_logic;
            mem_enable : out std_logic;
            mac_clr : out std_logic;
            alu_clr : out std_logic;
            btc_reset : out std_logic;
            btc_ready : in std_logic;
            btc_activate : out  std_logic;
            in_reg_enable : out std_logic;
            in_reg_out_en : out std_logic;
            out_reg_enable : out std_logic;
            ctrl_mem_a : out tlns_bus_word;
            ctrl_ir_mem_a : out tlns_bus_word;
            ma_mux_sel : out bit;
            ir_ma_mux_sel : out bit;
            ctrl_direct : out bit;
            mac_ch_mux_sel : out std_logic;
            mac_coefnum : out std_logic;
            mac_evensym : out std_logic;
            s1_bus_content : in tlns_bus_word;
            s2_bus_content : in tlns_bus_word );

    end component controller;

    signal s1_bus, s2_bus : tlns_bus_word;
    signal dest_bus : tlns_bus_word;
    signal s1_in, s2_in, ir_in : tlns_word;
    signal reg_file_out1, reg_file_out2 : tlns_word;
    signal current_instruction : tlns_word;
    signal alu_function : alu_func;
    signal alu_zero, alu_negative, alu_overflow : bit;
    signal reg_s1_addr, reg_s2_addr, reg_dest_addr : tlns_reg_addr;
    signal reg_write : std_logic;
    signal a_out_en : std_logic;
    signal a_enable : std_logic;
    signal b_out_en : std_logic;
    signal b_enable : std_logic;
    signal pc_out_en : std_logic;
    signal pc_enable : std_logic;
    signal mar_enable : std_logic;
    signal ir_immed1_size_18, ir_immed2_size_18 : bit;
    signal ir_immed1_unsigned, ir_immed2_unsigned : bit;
    signal ir_immed1_en, ir_immed2_en : bit;
```

```
  signal a_out : tlns_bus_word;
  signal pc_out : tlns_address;
  signal mem_addr : tlns_bus_word;
  signal mac_clr : std_logic;
  signal alu_clr : std_logic;
  signal btc_reset : std_logic;
  signal btc_ready : std_logic;
  signal btc_activate : std_logic;
  signal in_reg_enable : std_logic;
  signal in_reg_out_en : std_logic;
  signal out_reg_enable : std_logic;
  signal ctrl_mem_a : tlns_bus_word;
  signal din_demux_sel : bit;
  signal ma_mux_sel : bit;
  signal mem_data_in : tlns_bus_word;
  signal mem_data_out : tlns_bus_word;
  signal mar_a : tlns_bus_word;
  signal ctrl_ir_mem_a : tlns_bus_word;
  signal ir_ma_mux_sel : bit;
  signal ctrl_direct : bit;
  signal mac_ch_mux_sel : std_logic;
  signal mac_coefnum : std_logic;
  signal mac_evensym : std_logic;
  signal s1_bus_content : tlns_bus_word;
  signal s2_bus_content : tlns_bus_word;
  signal output_sign : std_logic_vector(1 downto 0);
  signal output_first : std_logic_vector(11 downto 0);
  signal output_second : std_logic_vector(9 downto 0);

begin

  -- Input / Output ports connections
  s1_in <= tlns_word(To_bitvector(s1_bus));
  s2_in <= tlns_word(To_bitvector(s2_bus));
  current_instruction <= tlns_word(To_bitvector(d));
  a <= a_out;
  mem_a <=  mem_addr(9 downto 0);
  mem_d_in <= dest_bus;
  s1_bus <= mem_d_out;

  -- The component instantiations

  output_register : output_reg
    port map ( clk => clk, d => s2_bus, q => output_data,
               out_enable => output_enable, enable => out_reg_enable );

  input_register : input_reg
    port map ( d => input_data, q => s1_bus, enable => in_reg_enable,
               out_en => in_reg_out_en );
```

```
BTC_converter : serial2digithighlow
  port map ( CK => clk, reset => btc_reset, activate => btc_activate,
             i => s1_bus(15 downto 0), ready => btc_ready,
             output_sign => output_sign,
             output_first => output_first,
             output_second => output_second );

BTC_reg : conv_out_reg
  port map ( ready => btc_ready,
             output_sign => output_sign,
             output_first => output_first,
             output_second => output_second,
             tlns_output => dest_bus);

multiplier_accumulater : mac
  port map ( clk => clk, clr => mac_clr,
             channel_mux_sel => mac_ch_mux_sel, coefnum => mac_coefnum
                 ,
             evensym => mac_evensym, x => s1_bus, y => s2_bus,
             p => dest_bus );

the_alu : alu
  port map ( s1 => s1_in, s2 => s2_in, result => dest_bus,
             func => alu_function, zero => alu_zero,
             negative => alu_negative, overflow => alu_overflow,
             clr => alu_clr);

the_reg_file : reg_file
  port map ( clk => clk, a1 => reg_s1_addr, q1 => reg_file_out1,
             a2 => reg_s2_addr, q2 => reg_file_out2,
             a3 => reg_dest_addr, d3 => dest_bus,
             write_en => reg_write );

a_reg : reg
  port map ( d => reg_file_out1, q => s1_bus, clk => clk,
             enable => a_enable, out_en => a_out_en );

b_reg : reg
  port map ( d => reg_file_out2, q => s2_bus, clk => clk,
             enable => b_enable, out_en => b_out_en );

pc_reg : reg_multiple_plus_one_out_reset
  port map ( d => dest_bus, q => s1_bus, q0 => pc_out, clk => clk,
             enable => pc_enable, out_en => pc_out_en, reset => reset
                 );

mar_register : mar_reg
  port map ( d => dest_bus, q => mar_a, clk => clk,
```

```
            enable => mar_enable );

ma_mux : mux2
  port map ( i0 => ctrl_mem_a, i1 => mar_a, y => mem_addr,
             sel => ma_mux_sel );

ir_ma_mux : mux2_ma
  port map ( i0 => ctrl_ir_mem_a, i1 => pc_out,  y => a_out,
             sel => ir_ma_mux_sel );

ir_extender1 : ir_extender
  port map ( d => current_instruction, q => s1_bus,
             immed_size_18 => ir_immed1_size_18,
             immed_unsigned => ir_immed1_unsigned,
             immed_en => ir_immed1_en );

ir_extender2 : ir_extender_director
  port map ( d => current_instruction, q => s2_bus,
             immed_size_18 => ir_immed2_size_18,
             immed_unsigned => ir_immed2_unsigned,
             immed_en => ir_immed2_en, direct => ctrl_direct);

the_controller : controller
  port map ( clk => clk, reset => reset, halt => halt,
             ir_mem_enable => ir_mem_enable,
             ifetch => ifetch, alu_function => alu_function,
             alu_zero => alu_zero, alu_negative => alu_negative,
             alu_overflow => alu_overflow, reg_s1_addr => reg_s1_addr,
             reg_s2_addr => reg_s2_addr,
             reg_dest_addr => reg_dest_addr, reg_write => reg_write,
             a_enable => a_enable, a_out_en => a_out_en,
             b_enable => b_enable, b_out_en => b_out_en,
             pc_enable => pc_enable, pc_out_en => pc_out_en,
             mar_enable => mar_enable,
             ir_immed1_size_18 => ir_immed1_size_18,
             ir_immed2_size_18 => ir_immed2_size_18,
             ir_immed1_unsigned => ir_immed1_unsigned,
             ir_immed2_unsigned => ir_immed2_unsigned,
             ir_immed1_en => ir_immed1_en, ir_immed2_en =>
                ir_immed2_en,
             current_instruction => current_instruction,
             const2 => s2_bus, mem_write_en => mem_write_en,
             mem_enable => mem_enable, mac_clr => mac_clr,
             alu_clr => alu_clr,
             btc_reset => btc_reset,
             btc_ready => btc_ready,
             btc_activate => btc_activate,
             in_reg_enable => in_reg_enable,
             in_reg_out_en => in_reg_out_en,
```

```
                out_reg_enable => out_reg_enable,
                ctrl_mem_a => ctrl_mem_a, ctrl_ir_mem_a => ctrl_ir_mem_a,
                ma_mux_sel => ma_mux_sel,
                ir_ma_mux_sel => ir_ma_mux_sel,
                ctrl_direct => ctrl_direct,
                mac_ch_mux_sel => mac_ch_mux_sel,
                mac_coefnum => mac_coefnum,
                mac_evensym => mac_evensym,
                s1_bus_content => s1_bus,
                s2_bus_content => s2_bus);

end architecture rtl;
```

A.2.2 The Arithmetic and Logic Unit (ALU)

This VHDL code describes the ALU in behavioral level.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.alu_types.all,
    work.numeric_bit.all;

entity alu is
  port ( s1 : in tlns_word;
         s2 : in tlns_word;
         result : out tlns_bus_word;
         func : in alu_func;
         zero, negative, overflow : out bit;
         clr : in  std_logic);
end entity alu;


architecture behavior of alu is
begin

  alu_op: process ( s1, s2, func ) is

    -- bit by bit adder
    procedure add ( L, R : in tlns_word;
                    result : out tlns_word;
                    overflow : out bit;
                    carry_in : in bit;
                    signed : in boolean) is
```

```
    variable carry : bit ;
    variable carry_prev : bit;

  begin
    carry := carry_in;
    for index in result'reverse_range loop
      carry_prev := carry;  -- of previous bit
      result(index) := L(index) xor R(index) xor carry;
      carry := (L(index) and R(index))
               or (carry and (L(index) xor R(index)));
    end loop;
    if signed then
      overflow := carry xor carry_prev;
    else
      overflow := carry;
    end if;
  end procedure add;

  variable temp_result : tlns_word;
  variable temp_overflow : bit;

begin
  temp_overflow := '0';

  -- determines the ALU result based on the ALU function
  case func is
    when alu_pass_s1 =>
      temp_result := s1;
    when alu_pass_s2 =>
      temp_result := s2;
    when alu_and =>
      temp_result := s1 and s2;
    when alu_or =>
      temp_result := s1 or s2;
    when alu_xor =>
      temp_result := s1 xor s2;
    when alu_sll =>
      temp_result := s1 sll to_integer(s2(3 downto 0));
    when alu_srl =>
      temp_result := s1 srl to_integer(s2(3 downto 0));
    when alu_sra =>
      temp_result := s1 sra to_integer(s2(3 downto 0));
    when alu_add =>
      add(s1, s2, temp_result, temp_overflow,
          carry_in => '0', signed => true);
    when alu_addu =>
      add(s1, s2, temp_result, temp_overflow,
          carry_in => '0', signed => false);
    when alu_sub =>
```

```
        add(s1, not s2, temp_result, temp_overflow,
            carry_in => '1', signed => true);
      when alu_subu =>
        add(s1, not s2, temp_result, temp_overflow,
            carry_in => '1', signed => false);
      when others =>
        null;
    end case;

    -- disables output, if ALU is not in use
    if (clr = '0') then
        result <= to_X01(bit_vector(temp_result));
      else
        result <= disabled_tlns_word;
    end if;

    -- sets the ALU zero flag
    if (temp_result = X"000_000") then
        zero <= '1';
      else
        zero <= '0';
    end if;

    -- sets the ALU sign flag
    negative <= temp_result(23) ;

    -- sets the ALU overflow flag
    overflow <= temp_overflow ;
  end process alu_op;

end architecture behavior;
```

A.2.3 The A and B registers

This is the VHDL code for registers which transfer data from register file to the data buses.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all;

entity reg is

  port ( d : in tlns_word;
```

```
        q : out tlns_bus_word;
        clk, enable : in std_logic;
        out_en : in std_logic );
end entity reg;


architecture behavior of reg is

  signal stored_value : tlns_word;

begin

  -- stores the input data
  reg: process ( clk ) is

  begin
    if rising_edge(clk) then
      if enable = '1' then
        stored_value <= d;
      end if;
    end if;
  end process reg;

    -- disables the output, when register is not out enabled
    q <= to_X01(bit_vector(stored_value)) when out_en = '1'
                                          else disabled_tlns_word;

end architecture behavior;
```

A.2.4 The Memory Address Register (MAR)

Whenever the data memory address is determined, MAR copies the input address directly to its output.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity mar_reg is
  port ( d : in tlns_bus_word;
         q : out tlns_bus_word;
         clk, enable : in std_logic );
end entity mar_reg;
```

```
architecture behavior of mar_reg is
begin

  -- directs input data to the output when is enable
  reg: process ( clk ) is
  begin
    if rising_edge(clk) then
      if enable = '1' then
        q <= d;
      end if;
    end if;
  end process reg;

end architecture behavior;
```

A.2.5 The Program Counter (PC)

The reset port of this register is connected to the reset signal of CPU. Therefore, program counter starts from zero, when CPU is reset. This register has two outputs, the first one always is connected to the data bus and is incremented in ALU, and the second one, when is enabled, transfers the address to the instruction memory multiplexer.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity reg_multiple_plus_one_out_reset is

  port ( d : in tlns_bus_word;
         q0 : out tlns_word;
         q : out tlns_bus_word;
         clk, enable : in std_logic;
         out_en : in std_logic;
         reset : in std_logic );
end entity reg_multiple_plus_one_out_reset;
```

```
architecture behavior of reg_multiple_plus_one_out_reset is

  signal stored_value : tlns_bus_word;

begin

  -- the register reset to zero based on CPU reset signal
  reg: process ( clk, reset ) is
  begin
    if reset = '1' then
      stored_value <= X"000_000";
    elsif rising_edge(clk) then
      if enable = '1' then
        stored_value <= d;
      end if;
    end if;
  end process reg;

  -- this output always goes to s1_bus
  q0 <= unsigned(to_bitvector(stored_value)) ;

  -- places the address onto memory multiplexer, when is enabled
  q <= stored_value when out_en = '1'
                    else disabled_tlns_word ;

end architecture behavior;
```

A.2.6 The Input Register

When the enable port of register is '1', data is stored and when the output is enabled the stored value is written to its output port.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity input_reg is
  port ( d : in external_data;
         q : out tlns_bus_word;
         enable : in std_logic;
         out_en : in std_logic);
end entity input_reg;
```

```
architecture behavior of input_reg is

  signal stored_value : tlns_bus_word;

begin

  -- data is stored when input is enabled
  input : process ( d, enable) is

  begin
    if enable = '1' then
        stored_value <= "00000000" & d;
    end if;

  end process input;

  -- stored data is written to output when output is enabled
  q <= stored_value when out_en = '1' else
        disabled_tlns_word ;

end architecture behavior;
```

A.2.7 The Output Register

While the enable port of register controls its output, the *out_enable* signal is set to be used by the controller. The controller makes use of this signal to read the output, one clock cycle after it has been written.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity output_reg is
  port ( clk : in std_logic;
         d : in tlns_bus_word;
         q : out tlns_bus_word;
         out_enable : out std_logic;
         enable : in std_logic);


end entity output_reg;
```

```
architecture behavior of output_reg is

begin

  output_data : process ( d, enable) is

  begin

    -- enable signal controls the output
    if enable = '1' then
        q <= d ;
    else
        q <= disabled_tlns_word;
    end if;
  end process output_data;

  -- the out_enable port is set to be used by the controller
  output_en : process ( clk ) is

  begin
    out_enable <= enable;
  end process output_en;

end architecture behavior;
```

A.2.8 The Register File

This VHDL file includes two processes for writing to and reading from register file. Register r0 is an exception, it never be written and returns zero, when it is read.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.tlns_instr.all,
    work.numeric_bit.all;

entity reg_file is
  port ( clk : in std_logic;
         a1 : in tlns_reg_addr;
         q1 : out tlns_word;
         a2 : in tlns_reg_addr;
         q2 : out tlns_word;
```

```
        a3 : in tlns_reg_addr;
        d3 : in tlns_bus_word;
        write_en : in std_logic);
end entity reg_file;


architecture behavior of reg_file is
    subtype register_array is tlns_word_array(1 to 15);
    constant all_zeros : tlns_word := X"000_000";
    signal register_file : register_array;

begin

  -- r0 is an exception for both read and write
  reg_write: process ( a3, d3, write_en, clk ) is

    variable reg_index3 : reg_index;

  begin
    -- does write if enabled
    if rising_edge(clk) then
      if write_en = '1' then
        reg_index3 := to_integer(a3);
        if reg_index3 /= 0 then
          register_file(reg_index3) <= unsigned(to_bitvector(d3));
        end if;
      end if;
    end if;

  end process reg_write;

  reg_read: process ( a1, a2 ) is

    variable reg_index1, reg_index2 : reg_index;

  begin
    -- read port 1
    reg_index1 := to_integer(a1);
    if reg_index1 /= 0 then
      q1 <= register_file(reg_index1) ;
    else
      q1 <= all_zeros ;
    end if;

    -- read port 2
    reg_index2 := to_integer(a2);
    if reg_index2 /= 0 then
      q2 <= register_file(reg_index2) ;
    else
```

```
      q2 <= all_zeros ;
    end if;
  end process reg_read;

end architecture behavior;
```

A.2.9 The Multiplexers

There are two multiplexers to address the memories. The only difference between these two is the different data types of inputs. Here is the code which describes the instruction memory multiplexer.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity mux2 is
  port ( i0 : in tlns_bus_word;
         i1 : in tlns_bus_word;
         y : out tlns_bus_word;
         sel : in bit);
end mux2;


architecture behavior of mux2 is
begin

  -- selects the input based on sel
  with sel select
    y <=  i0  when '0',
          i1  when '1',
          disabled_tlns_word when others;

end architecture behavior;
```

And, this file shows the multiplexer description for the data memory.

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use work.tlns_types.all,
    work.numeric_bit.all;

entity mux2_ma is
  port ( i0 : in tlns_bus_word;
         i1 : in tlns_address;
         y : out tlns_bus_word;
         sel : in bit);
end mux2_ma;


architecture behavior of mux2_ma is
begin

  -- selects the input based on sel
  with sel select
    y <=  i0  when '0',
          To_X01(bit_vector(i1))  when '1',
          disabled_tlns_word when others;

end architecture behavior;
```

A.2.10 The Extender

If the *immed_en* port is asserted to high, this module extends the immediate value in instruction to 24 bits.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity ir_extender is
  port ( d : in tlns_word;
         q : out tlns_bus_word;
         immed_size_18 : in bit;
         immed_unsigned : in bit;
         immed_en : in bit );
end entity ir_extender;


architecture behavior of ir_extender is
begin
```

```
  extender : process ( d, immed_en, immed_size_18, immed_unsigned ) is

  begin
    -- extends the input value when is enabled
    if immed_en = '1' then
      if immed_size_18 = '1' then
        -- 18-bit immediate value
        if immed_unsigned = '1' then
          -- resize and signed are functions in numeric_bit package
          q <= to_X01(bit_vector(resize(d(17 downto 0), 24))) ;
        else
          q <= to_X01(bit_vector(resize(signed(d(17 downto 0)), 24))) ;
        end if;
      else
        -- 10-bit immediate value
        if immed_unsigned = '1' then
          q <= to_X01(bit_vector(resize(d(9 downto 0), 24))) ;
        else
          q <= to_X01(bit_vector(resize(signed(d(9 downto 0)), 24))) ;
        end if;
      end if;
    -- disables the output if the extender is not enabled
    else
      q <= disabled_tlns_word ;
    end if;

  end process extender;

end architecture behavior;
```

A.2.11 The Extender / Director

This module also directs the immediate value to the output port when an extension is not necessary. It happens when a data is read from the instruction memory.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity ir_extender_director is
  port ( d : in tlns_word;
         q : out tlns_bus_word;
         immed_size_18 : in bit;
```

```
        immed_unsigned : in bit;
        immed_en : in bit;
        direct : in bit );
end entity ir_extender_director;


architecture behavior of ir_extender_director is
begin

  ext_dir : process ( d, immed_en, immed_size_18, immed_unsigned ) is

  begin
    -- first checks for a direction request
    if direct = '1' then
      q <= to_X01(bit_vector(d));
    else
     -- extends the input value when is enabled
     if immed_en = '1' then
        if immed_size_18 = '1' then
          -- 18-bit immediate value
          if immed_unsigned = '1' then
            -- resize and signed are functions in numeric_bit package
            q <= to_X01(bit_vector(resize(d(17 downto 0), 24))) ;
          else
            q <= to_X01(bit_vector(resize(signed(d(17 downto 0)), 24)))
                ;
          end if;
        else
          -- 10-bit immediate value
          if immed_unsigned = '1' then
            q <= to_X01(bit_vector(resize(d(9 downto 0), 24))) ;
          else
            q <= to_X01(bit_vector(resize(signed(d(9 downto 0)), 24))) ;
          end if;
        end if;
      -- disables the output if the extender is not enabled
      else
        q <= disabled_tlns_word ;
      end if;
    end if;

  end process ext_dir;

end architecture behavior;
```

A.2.12 The Binary / 2DLNS Converter (BTC)

For the binary / 2DLNS conversion, the Verilog code in [13] is used. The HDL code has been written fully parametrized and the parameters should be set when the shell script which generates Verilog module is run. The definition of these parameters were also included in [13]. Before running this script, the optimal base has been computed and stored in an ASCII file, 32768-13mn2unz.out. The binary / 2DLNS converter module in TLNS CPU has been generated by setting the parameters as:

makeserial2digithighlow.sh 32768-13mn2unz.out 16 6 5 3 2 1 -nz > myconverter.v

The name of generated module is **serial2digithighlow** which is instantiated, as an component, in the TLNS top module. This Verilog file is shown here.

```
'ifdef DC

'else

'timescale 1ns/10ps

// Define serial simulation module
module simulate_serial;

parameter inputbits = 16;
parameter firstbasebits = 6;
parameter secondbasebits = 5;
parameter twobitmode = 0;
parameter digits = 2;
parameter starter = -32768;
parameter stopper = 32767;

// Define systm clock
reg CK;

// Test bench registers, just one
reg [31:0] i;

// Interfacing registers
reg reset;
```

```
reg activate;
reg [ inputbits - 1 : 0 ] i0;
wire ready;
wire [ ( digits * ( twobitmode + 1 ) ) - 1 : 0 ] output_sign;
wire [ ( firstbasebits * digits ) - 1 : 0 ] output_first;
wire [ ( secondbasebits * digits ) - 1 : 0 ] output_second;

// Reordered converter results
reg [ twobitmode : 0 ] final_sign[ 0 : digits - 1];
reg [ firstbasebits - 1 : 0 ] final_first[ 0 : digits - 1 ];
reg [ secondbasebits - 1 : 0 ] final_second[ 0 : digits - 1 ];

// Temporary registers
reg [ ( digits * ( twobitmode + 1 ) ) - 1 : 0 ] output_sign2;
reg [ ( firstbasebits * digits ) - 1 : 0 ] output_first2;
reg [ ( secondbasebits * digits ) - 1 : 0 ] output_second2;

// Conversion operates on rising edge
serial2digithighlow
  serial(
    CK,
    reset,
    activate,
    i0,
    ready,
    output_sign,
    output_first,
    output_second
  );

integer q;

time starttime;

// Initialization routine
initial
begin

  // Disable verilog input logging
  $nokey;

  // Disable verilog output logging
  $nolog;

  // Initialize clock
  CK = 0;

  // Set initial input
  i = starter - 1;
```

```
  // Disable converter
  activate = 0;

  // Reset converter
  reset = 1;

end

// Set up the clock to pulse every 10 units
always #10 CK = !CK;

// On the rising edge
always @( posedge CK )
begin

  // When conversion is done, read converter results
  if( ready == 1 && activate == 0 )
  begin

    $display( "!total_time=%t", $time - starttime );

    // Copy the conerter results
    output_sign2 = output_sign;
    output_first2 = output_first;
    output_second2 = output_second;

    // Write out the input for comparison
    $write( "+%b", i0 );

    // Loop through all the digits to extract the sign and indices
    //  We can not use a variable in the indexing, so we use shifts
       instead
    for( q = 0 ; q < digits ; q = q + 1 )
    begin

      // Extract the sign
      final_sign[ q ] = output_sign2[ twobitmode : 0 ];
      // Shift it down for the next extraction
      output_sign2 = output_sign2 >> ( twobitmode + 1 );

      // Extract the first index
      final_first[ q ] = output_first2[ firstbasebits - 1 : 0 ];
      // Shift it down for the next extraction
      output_first2 = output_first2 >> firstbasebits;

      // Extract the second index
      final_second[ q ] = output_second2[ secondbasebits - 1 : 0 ];
      // Shift it down for the next extraction
```

```
        output_second2 = output_second2 >> secondbasebits;

        // Write out the sign and indices
        $write( "\t%b\t%b\t%b", final_sign[ q ], final_first[ q ],
                final_second[ q ] );

      end

      // Write a new line
      $write( "\n" );

      // Increment the test input
      i = i + 1;

      // Stop simulations when it reaches the last input
      if( i - 1 == stopper )
      begin

        $finish;

      end

      // Set the input
      i0 <= i[ inputbits - 1 : 0 ];

      // Request a conversion
      activate <= 1;

      // Record start time
      starttime = $time;

    end
    else
    begin

      // Conversion is either busy or in reset

      // Turn off conversion
      activate <= 0;

      // Take out of reset
      reset <= 0;

    end

  end

endmodule
```

```
`endif

// Define Serial 2 Digit High/Low Conversion Module

module serial2digithighlow(
  CK,
  reset,
  activate,
  i,
  ready,
  output_sign,
  output_first,
  output_second
);

// Define parameters

// inputbits: Input word size in bits
// internalbits: Internal working bit size (>= inputbits)
// firstbasebits: Number of bits for the first base index
// secondbasebits: Number of bits for the second base index
// secondbasereservedbits: Number of bits for exclusion on the
//                         secondbase index
// normalizerbits: Number of bits for shift from normalizer
// twobitmode: Sign mode (1 for two-bits, 0 for one-bit)

parameter inputbits = 16;
parameter internalbits = 17;
parameter firstbasebits = 6;
parameter secondbasebits = 5;
parameter secondbasereservedbits = 3;
parameter normalizerbits = 5;
parameter twobitmode = 0;

// Dummy parameter, should be for the number of digits
parameter dummy = 2;

// This is a two digit 2DLNS converter
parameter digits = 2;

// Internal comparitor accuracy, this should be adjusted after
   simulation
parameter shiftdifference = 20;

// Define ports

// Data is processed on rising edge
input CK;
```

```
// Reset is active high
input reset;

// Input word in 2's complement
input [ inputbits - 1 : 0 ] i;

// Set to 1 to run converstion on input
input activate;

// Is set to 1 when output data is ready
output ready;
reg ready;

// Output Signs (concatenated)
output [ ( twobitmode + 1 ) * digits - 1 : 0 ] output_sign;
reg [ ( twobitmode + 1 ) * digits - 1 : 0 ] output_sign;

// Output Binary exponent (concatenated)
output [ ( firstbasebits * digits ) - 1 : 0 ] output_first;
reg [ ( firstbasebits * digits ) - 1 : 0 ] output_first;

// Output OtherBase exponent (concatenated)
output [ ( secondbasebits * digits ) - 1 : 0 ] output_second;
reg [ ( secondbasebits * digits ) - 1 : 0 ] output_second;

reg [ inputbits - 1 : 0 ] sep_manin;
wire [ internalbits - 1 : 0 ] sep_manout;
wire [ twobitmode : 0 ] sep_signout;

separatesign_noclk
  #(
    inputbits,
    internalbits,
    twobitmode
  )
  ss
  (
    sep_manin,
    sep_manout,
    sep_signout
  );

reg [ internalbits - 1 : 0 ] norm_manin;
reg [ twobitmode : 0 ] norm_signin;

wire [ internalbits - 1 - twobitmode : 0 ] norm_manout;
wire [ twobitmode : 0 ] norm_signout;
wire [ normalizerbits - 1 : 0 ] norm_shift;
```

```
normalizer_noclk
  #(
    internalbits,
    normalizerbits,
    twobitmode
  )
  no
  (
    norm_manin,
    norm_signin,
    norm_manout,
    norm_signout,
    norm_shift
  );

reg [ internalbits - 1 - twobitmode : 0 ] ralut_manin;
wire [ internalbits - 1 - twobitmode : 0 ] ralut_manlow;
wire [ internalbits : 0 ] ralut_manhigh;
wire [ firstbasebits - 1 : 0 ] ralut_firstlow, ralut_firsthigh;
wire [ secondbasebits - 1 : 0 ] ralut_secondlow, ralut_secondhigh;

ralut6_178605502_2106_noclk
  #(
    internalbits - twobitmode,
    firstbasebits,
    secondbasebits,
    internalbits + 1,
    firstbasebits,
    secondbasebits,
    internalbits - twobitmode
  )
  ralut
  (
    ralut_manlow,
    ralut_firstlow,
    ralut_secondlow,
    ralut_manhigh,
    ralut_firsthigh,
    ralut_secondhigh,
    ralut_manin
  );

reg [ twobitmode : 0 ] final_sign[ 0 : digits - 1 ];
reg [ firstbasebits - 1 : 0 ] final_first[ 0 : digits - 1 ];
reg [ secondbasebits - 1 : 0 ] final_second[ 0 : digits - 1 ];

reg [ twobitmode : 0 ] other_sign;
reg [ firstbasebits - 1 :0 ] other_first[ 0 : digits - 1 ];
reg [ secondbasebits - 1 :0 ] other_second[ 0 : digits - 1 ];
```

```
reg [ internalbits : 0 ] error_low, error_high, other_error;

reg [ normalizerbits : 0 ] other_shiftdifference, best_shiftdifference;
reg [ normalizerbits - 1 : 0 ] other_lastshift, best_lastshift;
reg [ internalbits - 1 + shiftdifference : 0 ] other_errorcompare,
                                               best_errorcompare;

reg [ internalbits - 1 : 0 ] best_error;
reg [ normalizerbits : 0 ] best_shift, other_shift;

// State machine register
reg [ 3 : 0 ] state;

`ifdef DC

`else

// For simulation optimize the barrel shifter
integer max_shiftdifference;

// Initialization routine
initial
begin

  // Reset the shiftdifference maximum
  max_shiftdifference = 0;

  // Stop if more than 2 digits were passed
  if( dummy != digits ) $stop;

end

`endif

// Reset integer
integer j;

// Give Synopsys some hints on the synthesis
//synopsys state_vector state
//synopsys sync_set_reset "reset"

always @( posedge CK )
begin: main

  // Synchronous Reset active high
  if( reset )
  begin
```

```
    // Reset state machine
    state <= 0;

    // Set output to invalid
    ready <= 0;

    // Set all MDLNS output to zeros
    for( j = 0 ; j < 2 ; j = j + 1)
    begin

      final_sign[ j ] <= 0;
      final_first[ j ] <= 0;
      final_second[ j ] <= 0;

    end

  end
  else
  begin

    // Process each state
    case( state )

      0:
      begin

        // Conversion starts if activate line is high
        if( activate )
        begin

          // Load input data into sign separator
          sep_manin <= i;

          // Invalidate output
          ready <= 0;

          $display( "Conversion starts for %d", i );

          // Set all MDLNS output to zeros incase conversion finishes
             early
          for( j = 0 ; j < 2 ; j = j + 1)
          begin

            final_sign[ j ] <= 0;
            final_first[ j ] <= 0;
            final_second[ j ] <= 0;

          end
```

```
      // Move to next state
      state <= state + 1;

    end
    else
    begin

      // No conversion signal, continue looping
      ready <= 1;

    end

  end

  1: begin

    // Get sign and data from sign separator, put it in to the
       normalizer
    norm_signin <= sep_signout;
    norm_manin <= sep_manout;

    // Move to next state
    state <= state + 1;

  end

  2:
  begin

    if( twobitmode && norm_signout == 0 )
    begin

      // input is zero, end conversion
      ready <= 1;
      state <= 0;

    end
    else
    begin

      // Get shift and sign from normalizer
      final_sign[ 0 ] <= norm_signout;
      other_shift <= norm_shift;

      // Load RALUT with the mantissa
      ralut_manin <= norm_manout;

      // Move to next state
      state <= state + 1;
```

```
    end

  end

  3:
  begin

    // Determine the error between the two RALUT outputs and the
       input
    if( twobitmode )
    begin

      error_low = { 2'b01, ralut_manin } - { 2'b01, ralut_manlow };
      other_error <= ralut_manhigh - { 2'b01, ralut_manin };

    end
    else
    begin

      error_low = { 1'b0, ralut_manin } - { 1'b0, ralut_manlow };
      other_error <= ralut_manhigh - { 1'b0, ralut_manin };

    end

    // Load normalizer with low error and sign of input (to
       determine
    //                                                    if it is
       zero)
    norm_manin <= error_low[ internalbits - 1 : 0 ];
    norm_signin <= final_sign[ 0 ];

    // Save the low approximation as the current result
    final_first[ 0 ] <= ralut_firstlow - other_shift;
    final_second[ 0 ] <= ralut_secondlow;

    // Remember the high approximation for later
    other_first[ 0 ] <= ralut_firsthigh - other_shift;
    other_second[ 0 ] <= ralut_secondhigh;

    // Move to next state
    state <= state + 1;

  end

  4:
  begin

    // Find the second digit for the low approximation
```

```
    if( twobitmode && norm_signout == 0 )
    begin

      // Error is zero, end conversion
      ready <= 1;
      state <= 0;

    end
    else
    begin

      // Set output sign
      final_sign[ 1 ] <= norm_signout;
      // Save accumulated shift
      best_shift <= other_shift + norm_shift;
      // Load RALUT
      ralut_manin <= norm_manout;

      // Move to next state
      state <= state + 1;

    end

  end

  5:
  begin

    // Determine the error between the two RALUT outputs and the
       input
    //  We do this since we only have one RALUT, we have to find if
       the
    //   low or the high approximation is best
    if( twobitmode )
    begin

      error_low = { 2'b01, ralut_manin } - { 2'b01, ralut_manlow };
      error_high = ralut_manhigh - { 2'b01, ralut_manin };

    end
    else
    begin

      error_low = { 1'b0, ralut_manin } - { 1'b0, ralut_manlow };
      error_high = ralut_manhigh - { 1'b0, ralut_manin };

    end

    // Set the current result as the best approximation
```

```
    if( error_low < error_high )
    begin

      final_first[ 1 ] <= ralut_firstlow - best_shift;
      final_second[ 1 ] <= ralut_secondlow;
      best_error <= error_low;

    end
    else
    begin

      final_first[ 1 ] <= ralut_firsthigh - best_shift;
      final_second[ 1 ] <= ralut_secondhigh;
      best_error <= error_high;

    end

    // Load normalizer with high error from before and sign of -1
    norm_manin <= other_error[ internalbits - 1 : 0 ];

    if( twobitmode )
    begin

      if( final_sign[ 0 ] == 1 )
      begin

        norm_signin <= 2'b11;

      end
      else
      begin

        norm_signin <= 2'b01;

      end

    end
    else
    begin

      norm_signin <= ~final_sign[ 0 ];

    end

    // Move to next state
    state <= state + 1;

  end
```

```
6:
begin

  // Save accumulated shift and sign
  other_shift <= other_shift + norm_shift;
  other_sign <= norm_signout;

  // Load RALUT
  ralut_manin <= norm_manout;

  // Move to next state
  state <= state + 1;

end

7:
begin

  // Determine the error between the two RALUT outputs and the
     input
  if( twobitmode )
  begin

    error_low = { 2'b01, ralut_manin } - { 2'b01, ralut_manlow };
    error_high = ralut_manhigh - { 2'b01, ralut_manin };

  end
  else
  begin

    error_low = { 1'b0, ralut_manin } - { 1'b0, ralut_manlow };
    error_high = ralut_manhigh - { 1'b0, ralut_manin };

  end

  // Store this result temporarily
  if ( error_low < error_high )
  begin

    other_first[ 1 ] <= ralut_firstlow - other_shift;
    other_second[ 1 ] <= ralut_secondlow;
    other_error <= error_low;

  end
  else
  begin

    other_first[ 1 ] <= ralut_firsthigh - other_shift;
    other_second[ 1 ] <= ralut_secondhigh;
```

```
            other_error <= error_high;

          end

          // Move to next state
          state <= state + 1;

        end

        8:
        begin

          // Find the difference in shifts between the two approximations
          other_shiftdifference = other_shift - best_shift;
          best_shiftdifference = best_shift - other_shift;

          if( other_shiftdifference[ normalizerbits ] == 1'b0 )
          begin

            // high approximation must be shifted right
            other_lastshift <= other_shiftdifference;
            best_lastshift <= 0;

          end
          else
          begin

            // low approximation must be shifted right
            other_lastshift <= 0;
            best_lastshift <= best_shiftdifference;

          end

          // Move to next state
          state <= state + 1;

        end

        9:
        begin

`ifdef DC

// For synthsis do not do this

`else

          // Find the maximum shift to minimize the hardware
```

```
        if( best_shift != 0 )
        begin

          if( other_lastshift > max_shiftdifference )
          begin

            max_shiftdifference = other_lastshift;
            $display( "!max_shiftdifference=%d", max_shiftdifference );

          end
          if( best_lastshift > max_shiftdifference )
          begin

            max_shiftdifference = best_lastshift;
            $display( "!max_shiftdifference=%d", max_shiftdifference );

          end

        end

`endif

        // Shift both errors accordingly
        other_errorcompare = ( other_error << shiftdifference ) >>
                             other_lastshift;
        best_errorcompare = ( best_error << shiftdifference ) >>
                            best_lastshift;

        // Compare
        if( other_errorcompare < best_errorcompare )
        begin

          // Higher approximation is better, move into output results
          final_first[ 0 ] <= other_first[ 0 ];
          final_second[ 0 ] <= other_second[ 0 ];
          final_sign[ 1 ] <= other_sign;
          final_first[ 1 ] <= other_first[ 1 ];
          final_second[ 1 ] <= other_second[ 1 ];

        end

        // Conversion complete
        ready <= 1;
        state <= 0;

      end

      // All other cases, reset the state machine
      default:
```

```
      begin

        state <= 0;
        ready <= 0;

      end

    endcase

  end

end // always

`ifdef DC

// For synthesis
always

`else

// For simulation
always @( posedge ready )

`endif

begin

// Concatinate outputs for universal access
  output_sign = { final_sign[ 1 ], final_sign[ 0 ] };
  output_first = { final_first[ 1 ], final_first[ 0 ] };
  output_second = { final_second[ 1 ], final_second[ 0 ] };

end

endmodule

// Module for separating an integer into a number and a sign
//  This module doesn't generate a zero sign, simply a 1 or -1
module separatesign_noclk ( i, o, os );

// Default parameters

// Input word size in bits
parameter inputbits = 16;

// Output word size in bits (must be equal to or greater than inputbits)
parameter outputbits = 15;

// twobit sign indicator
```

```
parameter twobitmode = 1;

// Define ports

// Input of module in 2's complements form
input [ inputbits - 1 : 0 ] i;

// Output number of module in binary form
output [ outputbits - 1 : 0 ] o;
reg [ outputbits - 1 : 0 ] o;

// Output sign of module
output [ twobitmode : 0 ] os;
reg [ twobitmode : 0 ] os;

`ifdef DC

`else

initial
begin

  // Stop simulation if output is smaller than input
  if( outputbits < inputbits ) $stop;

end

`endif

// Two intermediate registers to expand word length
reg [ 31 : 0 ] t1, t2;

always @( i )
begin

  // Generate negative value of input, but extend the sign too
  t1 = 0 - { { ( 31 - inputbits ) { i[ inputbits - 1 ] } },
             i[ inputbits - 2 : 0 ] };

  // Copy input
  t2 = i;

  // Check the high bit of the input, if it is one, then input
  //  is negative
  if( i[ inputbits - 1 ] == 1'b1 )
  begin

    // Set output to negated input
    o = t1[ outputbits - 1 : 0 ];
```

```
    // Set sign to -1, 1 for one bit sign mode
    if( twobitmode )
    begin

      os = 2'b11;

    end
    else
    begin

      os = 1'b1;

    end

  end
  else
  begin

    // Set output to extended input
    o = t2[ outputbits - 1 : 0 ];

    // Set sign to 1, 0 for one bit sign mode
    if( twobitmode )
    begin

      os = 2'b01;

    end
    else
    begin

      os = 1'b0;

    end

  end

end

endmodule

// Module for normalizing and integer and possibly setting the sign to
   zero
module normalizer_noclk( i, is, o, os, s );

// Default parameters

// inputbits is the input number of bits, the output will be inputbits-1
```

```
//  bits since the first "1" will be omitted
parameter inputbits = 16;

// shiftbits is the number of bits to describe the shift of the
//  normalization
parameter shiftbits = 4;

// two bit mode
parameter twobitmode = 1;

// Define ports

// input
input [ inputbits - 1 : 0 ] i;

// input sign
input [ twobitmode : 0 ] is;

// output of normalization,
output [ inputbits - 1 - twobitmode : 0 ] o;
reg [ inputbits - 1 - twobitmode : 0 ] o;

// output sign, will be zero is input sign is also
output [ twobitmode : 0 ] os;
reg [ twobitmode : 0 ] os;

// output shift
output [ shiftbits - 1 : 0 ] s;
reg [ shiftbits - 1 : 0 ] s;

// temporary variables
reg [ inputbits - 1 : 0 ] t;
reg z;

// internal counter
integer c;

always @( i or is )
begin

  // there are other ways to do this procedure, but through synthesis
  //  this turns out to be the smallest

  // set shift to zero
  s = 0;

  // set loop control
  z = 0;
```

```
// loop through all bits from left to right
for( c = inputbits - 1 ; c >= 0 ; c = c - 1 )
begin

  // if the bit is zero and no ones have been encountered,
  //  keep updating this shift
  if( i[ c ] == 1'b0 && z == 1'b0 )
  begin

    // record the size
    s = inputbits - c;

    // set the loop to keep going
    z = 0;

  end
  else
  begin

    // don't infer any latches
    s = s;
    // a one has been spotted, no more updates
    z = 1;

  end

end

// check for twobit mode
if( twobitmode )
begin

  // check the loop control bit
  if( z == 0 )
  begin

    // if zero, set the output sign to zero
    os = 0;
    s = 0;

  end
  else
  begin

    // if not, pass the input sign through
    os = is;

  end
```

```
    end
    else
    begin

      // pass the input sign through
      os = is;

      // check the loop control bit
      if( z == 0 )
      begin

        s = 0;

      end

    end

    // shift temporary register
    t = i << s;

    // set output from temporary register (we can not do this in one step)
    o = t[ inputbits - 1 - twobitmode : 0 ];

end

endmodule

module ralut6_178605502_2106_noclk(
    o1,
    o2,
    o3,
    o4,
    o5,
    o6,
    i );

// Default parameters
parameter o1bits = 17;
parameter o2bits = 6;
parameter o3bits = 5;
parameter o4bits = 18;
parameter o5bits = 6;
parameter o6bits = 5;
parameter ibits = 17;
parameter rasize = 26;

// Define ports

// data to match
```

```
input [ ibits - 1 : 0 ] i;

// datas to output
output [ o1bits - 1 : 0 ] o1;
output [ o2bits - 1 : 0 ] o2;
output [ o3bits - 1 : 0 ] o3;
output [ o4bits - 1 : 0 ] o4;
output [ o5bits - 1 : 0 ] o5;
output [ o6bits - 1 : 0 ] o6;
reg [ o1bits - 1: 0 ] o1;
reg [ o2bits - 1: 0 ] o2;
reg [ o3bits - 1: 0 ] o3;
reg [ o4bits - 1: 0 ] o4;
reg [ o5bits - 1: 0 ] o5;
reg [ o6bits - 1: 0 ] o6;

reg [ rasize - 1 : 0 ] d;

reg [ rasize : 0 ] c;

reg [ ibits - 1 : 0 ] romAddr[ rasize - 1 : 0 ];
reg [ o1bits - 1 : 0 ] romOut1[ rasize - 1 : 0 ];
reg [ o2bits - 1 : 0 ] romOut2[ rasize - 1 : 0 ];
reg [ o3bits - 1 : 0 ] romOut3[ rasize - 1 : 0 ];
reg [ o4bits - 1 : 0 ] romOut4[ rasize - 1 : 0 ];
reg [ o5bits - 1 : 0 ] romOut5[ rasize - 1 : 0 ];
reg [ o6bits - 1 : 0 ] romOut6[ rasize - 1 : 0 ];

reg [ 30 : 0 ] high;

wire [ o1bits - 1 : 0 ] fin_o1;
wire [ o2bits - 1 : 0 ] fin_o2;
wire [ o3bits - 1 : 0 ] fin_o3;
wire [ o4bits - 1 : 0 ] fin_o4;
wire [ o5bits - 1 : 0 ] fin_o5;
wire [ o6bits - 1 : 0 ] fin_o6;
reg [ o1bits - 1 : 0 ] tri_o1[ rasize - 1 : 0 ];
reg [ o2bits - 1 : 0 ] tri_o2[ rasize - 1 : 0 ];
reg [ o3bits - 1 : 0 ] tri_o3[ rasize - 1 : 0 ];
reg [ o4bits - 1 : 0 ] tri_o4[ rasize - 1 : 0 ];
reg [ o5bits - 1 : 0 ] tri_o5[ rasize - 1 : 0 ];
reg [ o6bits - 1 : 0 ] tri_o6[ rasize - 1 : 0 ];

integer p;
integer j;
integer k;

`ifdef DC
`else
```

```
initial
begin

// include data information here

romAddr[0]=17'b00000000000000000;
romAddr[1]=17'b10000000000000000;
romAddr[2]=17'b10000011101010011;
romAddr[3]=17'b10000111001110010;
romAddr[4]=17'b10001011000101111;
romAddr[5]=17'b10001111000100101;
romAddr[6]=17'b10010010111100011;
romAddr[7]=17'b10010111001001100;
romAddr[8]=17'b10011011011110010;
romAddr[9]=17'b10011111101011011;
romAddr[10]=17'b10100100001111111;
romAddr[11]=17'b10101000111100101;
romAddr[12]=17'b10101101100001001;
romAddr[13]=17'b10110010011110111;
romAddr[14]=17'b10110111100101110;
romAddr[15]=17'b10111100110110000;
romAddr[16]=17'b11000001111100111;
romAddr[17]=17'b11000111100000001;
romAddr[18]=17'b11001101001101100;
romAddr[19]=17'b11010010110000110;
romAddr[20]=17'b11011000110010110;
romAddr[21]=17'b11011110111111110;
romAddr[22]=17'b11100101000001110;
romAddr[23]=17'b11101011100101010;
romAddr[24]=17'b11110010010100101;
romAddr[25]=17'b11111000111000001;

romOut1[0]=17'b00000000000000000;
romOut2[0]=6'b010000;
romOut3[0]=5'b10000;
romOut4[0]=18'b000000000000000000;
romOut5[0]=6'b010000;
romOut6[0]=5'b10000;
romOut1[1]=17'b10000000000000000;
romOut2[1]=6'b010000;
romOut3[1]=5'b00000;
romOut4[1]=18'b010000011101010011;
romOut5[1]=6'b010001;
romOut6[1]=5'b01000;
romOut1[2]=17'b10000011101010011;
romOut2[2]=6'b010001;
romOut3[2]=5'b01000;
romOut4[2]=18'b010000111001110010;
```

```
romOut5[2]=6'b001111;
romOut6[2]=5'b10111;
romOut1[3]=17'b10000111001110010;
romOut2[3]=6'b001111;
romOut3[3]=5'b10111;
romOut4[3]=18'b010001011000101111;
romOut5[3]=6'b010000;
romOut6[3]=5'b11111;
romOut1[4]=17'b10001011000101111;
romOut2[4]=6'b010000;
romOut3[4]=5'b11111;
romOut4[4]=18'b010001111000100101;
romOut5[4]=6'b010001;
romOut6[4]=5'b00111;
romOut1[5]=17'b10001111000100101;
romOut2[5]=6'b010001;
romOut3[5]=5'b00111;
romOut4[5]=18'b010010010111100011;
romOut5[5]=6'b001111;
romOut6[5]=5'b10110;
romOut1[6]=17'b10010010111100011;
romOut2[6]=6'b001111;
romOut3[6]=5'b10110;
romOut4[6]=18'b010010111001001100;
romOut5[6]=6'b010000;
romOut6[6]=5'b11110;
romOut1[7]=17'b10010111001001100;
romOut2[7]=6'b010000;
romOut3[7]=5'b11110;
romOut4[7]=18'b010011011011110010;
romOut5[7]=6'b010001;
romOut6[7]=5'b00110;
romOut1[8]=17'b10011011011110010;
romOut2[8]=6'b010001;
romOut3[8]=5'b00110;
romOut4[8]=18'b010011111101011011;
romOut5[8]=6'b001111;
romOut6[8]=5'b10101;
romOut1[9]=17'b10011111101011011;
romOut2[9]=6'b001111;
romOut3[9]=5'b10101;
romOut4[9]=18'b010100100001111111;
romOut5[9]=6'b010000;
romOut6[9]=5'b11101;
romOut1[10]=17'b10100100001111111;
romOut2[10]=6'b010000;
romOut3[10]=5'b11101;
romOut4[10]=18'b010101000111100101;
romOut5[10]=6'b010001;
```

```
romOut6[10]=5'b00101;
romOut1[11]=17'b10101000111100101;
romOut2[11]=6'b010001;
romOut3[11]=5'b00101;
romOut4[11]=18'b010101101100001001;
romOut5[11]=6'b001111;
romOut6[11]=5'b10100;
romOut1[12]=17'b10101101100001001;
romOut2[12]=6'b001111;
romOut3[12]=5'b10100;
romOut4[12]=18'b010110010011110111;
romOut5[12]=6'b010000;
romOut6[12]=5'b11100;
romOut1[13]=17'b10110010011110111;
romOut2[13]=6'b010000;
romOut3[13]=5'b11100;
romOut4[13]=18'b010110111100101110;
romOut5[13]=6'b010001;
romOut6[13]=5'b00100;
romOut1[14]=17'b10110111100101110;
romOut2[14]=6'b010001;
romOut3[14]=5'b00100;
romOut4[14]=18'b010111100110110000;
romOut5[14]=6'b010010;
romOut6[14]=5'b01100;
romOut1[15]=17'b10111100110110000;
romOut2[15]=6'b010010;
romOut3[15]=5'b01100;
romOut4[15]=18'b011000001111100111;
romOut5[15]=6'b010000;
romOut6[15]=5'b11011;
romOut1[16]=17'b11000001111100111;
romOut2[16]=6'b010000;
romOut3[16]=5'b11011;
romOut4[16]=18'b011000111100000001;
romOut5[16]=6'b010001;
romOut6[16]=5'b00011;
romOut1[17]=17'b11000111100000001;
romOut2[17]=6'b010001;
romOut3[17]=5'b00011;
romOut4[17]=18'b011001101001101100;
romOut5[17]=6'b010010;
romOut6[17]=5'b01011;
romOut1[18]=17'b11001101001101100;
romOut2[18]=6'b010010;
romOut3[18]=5'b01011;
romOut4[18]=18'b011010010110000110;
romOut5[18]=6'b010000;
romOut6[18]=5'b11010;
```

```
romOut1[19]=17'b11010010110000110;
romOut2[19]=6'b010000;
romOut3[19]=5'b11010;
romOut4[19]=18'b011011000110010110;
romOut5[19]=6'b010001;
romOut6[19]=5'b00010;
romOut1[20]=17'b11011000110010110;
romOut2[20]=6'b010001;
romOut3[20]=5'b00010;
romOut4[20]=18'b011011110111111110;
romOut5[20]=6'b010010;
romOut6[20]=5'b01010;
romOut1[21]=17'b11011110111111110;
romOut2[21]=6'b010010;
romOut3[21]=5'b01010;
romOut4[21]=18'b011100101000001110;
romOut5[21]=6'b010000;
romOut6[21]=5'b11001;
romOut1[22]=17'b11100101000001110;
romOut2[22]=6'b010000;
romOut3[22]=5'b11001;
romOut4[22]=18'b011101011100101010;
romOut5[22]=6'b010001;
romOut6[22]=5'b00001;
romOut1[23]=17'b11101011100101010;
romOut2[23]=6'b010001;
romOut3[23]=5'b00001;
romOut4[23]=18'b011110010010100101;
romOut5[23]=6'b010010;
romOut6[23]=5'b01001;
romOut1[24]=17'b11110010010100101;
romOut2[24]=6'b010010;
romOut3[24]=5'b01001;
romOut4[24]=18'b011111000111000001;
romOut5[24]=6'b010000;
romOut6[24]=5'b11000;
romOut1[25]=17'b11111000111000001;
romOut2[25]=6'b010000;
romOut3[25]=5'b11000;
romOut4[25]=18'b100000000000000000;
romOut5[25]=6'b010001;
romOut6[25]=5'b00000;

  high = 31'bz;
  c[ rasize ] = 0;
end
`endif

// include line shorting here
```

```
assign fin_o1 = tri_o1[0];
assign fin_o2 = tri_o2[0];
assign fin_o3 = tri_o3[0];
assign fin_o4 = tri_o4[0];
assign fin_o5 = tri_o5[0];
assign fin_o6 = tri_o6[0];
assign fin_o1 = tri_o1[1];
assign fin_o2 = tri_o2[1];
assign fin_o3 = tri_o3[1];
assign fin_o4 = tri_o4[1];
assign fin_o5 = tri_o5[1];
assign fin_o6 = tri_o6[1];
assign fin_o1 = tri_o1[2];
assign fin_o2 = tri_o2[2];
assign fin_o3 = tri_o3[2];
assign fin_o4 = tri_o4[2];
assign fin_o5 = tri_o5[2];
assign fin_o6 = tri_o6[2];
assign fin_o1 = tri_o1[3];
assign fin_o2 = tri_o2[3];
assign fin_o3 = tri_o3[3];
assign fin_o4 = tri_o4[3];
assign fin_o5 = tri_o5[3];
assign fin_o6 = tri_o6[3];
assign fin_o1 = tri_o1[4];
assign fin_o2 = tri_o2[4];
assign fin_o3 = tri_o3[4];
assign fin_o4 = tri_o4[4];
assign fin_o5 = tri_o5[4];
assign fin_o6 = tri_o6[4];
assign fin_o1 = tri_o1[5];
assign fin_o2 = tri_o2[5];
assign fin_o3 = tri_o3[5];
assign fin_o4 = tri_o4[5];
assign fin_o5 = tri_o5[5];
assign fin_o6 = tri_o6[5];
assign fin_o1 = tri_o1[6];
assign fin_o2 = tri_o2[6];
assign fin_o3 = tri_o3[6];
assign fin_o4 = tri_o4[6];
assign fin_o5 = tri_o5[6];
assign fin_o6 = tri_o6[6];
assign fin_o1 = tri_o1[7];
assign fin_o2 = tri_o2[7];
assign fin_o3 = tri_o3[7];
assign fin_o4 = tri_o4[7];
assign fin_o5 = tri_o5[7];
assign fin_o6 = tri_o6[7];
```

```
assign fin_o1 = tri_o1[8];
assign fin_o2 = tri_o2[8];
assign fin_o3 = tri_o3[8];
assign fin_o4 = tri_o4[8];
assign fin_o5 = tri_o5[8];
assign fin_o6 = tri_o6[8];
assign fin_o1 = tri_o1[9];
assign fin_o2 = tri_o2[9];
assign fin_o3 = tri_o3[9];
assign fin_o4 = tri_o4[9];
assign fin_o5 = tri_o5[9];
assign fin_o6 = tri_o6[9];
assign fin_o1 = tri_o1[10];
assign fin_o2 = tri_o2[10];
assign fin_o3 = tri_o3[10];
assign fin_o4 = tri_o4[10];
assign fin_o5 = tri_o5[10];
assign fin_o6 = tri_o6[10];
assign fin_o1 = tri_o1[11];
assign fin_o2 = tri_o2[11];
assign fin_o3 = tri_o3[11];
assign fin_o4 = tri_o4[11];
assign fin_o5 = tri_o5[11];
assign fin_o6 = tri_o6[11];
assign fin_o1 = tri_o1[12];
assign fin_o2 = tri_o2[12];
assign fin_o3 = tri_o3[12];
assign fin_o4 = tri_o4[12];
assign fin_o5 = tri_o5[12];
assign fin_o6 = tri_o6[12];
assign fin_o1 = tri_o1[13];
assign fin_o2 = tri_o2[13];
assign fin_o3 = tri_o3[13];
assign fin_o4 = tri_o4[13];
assign fin_o5 = tri_o5[13];
assign fin_o6 = tri_o6[13];
assign fin_o1 = tri_o1[14];
assign fin_o2 = tri_o2[14];
assign fin_o3 = tri_o3[14];
assign fin_o4 = tri_o4[14];
assign fin_o5 = tri_o5[14];
assign fin_o6 = tri_o6[14];
assign fin_o1 = tri_o1[15];
assign fin_o2 = tri_o2[15];
assign fin_o3 = tri_o3[15];
assign fin_o4 = tri_o4[15];
assign fin_o5 = tri_o5[15];
assign fin_o6 = tri_o6[15];
assign fin_o1 = tri_o1[16];
```

```
assign fin_o2 = tri_o2[16];
assign fin_o3 = tri_o3[16];
assign fin_o4 = tri_o4[16];
assign fin_o5 = tri_o5[16];
assign fin_o6 = tri_o6[16];
assign fin_o1 = tri_o1[17];
assign fin_o2 = tri_o2[17];
assign fin_o3 = tri_o3[17];
assign fin_o4 = tri_o4[17];
assign fin_o5 = tri_o5[17];
assign fin_o6 = tri_o6[17];
assign fin_o1 = tri_o1[18];
assign fin_o2 = tri_o2[18];
assign fin_o3 = tri_o3[18];
assign fin_o4 = tri_o4[18];
assign fin_o5 = tri_o5[18];
assign fin_o6 = tri_o6[18];
assign fin_o1 = tri_o1[19];
assign fin_o2 = tri_o2[19];
assign fin_o3 = tri_o3[19];
assign fin_o4 = tri_o4[19];
assign fin_o5 = tri_o5[19];
assign fin_o6 = tri_o6[19];
assign fin_o1 = tri_o1[20];
assign fin_o2 = tri_o2[20];
assign fin_o3 = tri_o3[20];
assign fin_o4 = tri_o4[20];
assign fin_o5 = tri_o5[20];
assign fin_o6 = tri_o6[20];
assign fin_o1 = tri_o1[21];
assign fin_o2 = tri_o2[21];
assign fin_o3 = tri_o3[21];
assign fin_o4 = tri_o4[21];
assign fin_o5 = tri_o5[21];
assign fin_o6 = tri_o6[21];
assign fin_o1 = tri_o1[22];
assign fin_o2 = tri_o2[22];
assign fin_o3 = tri_o3[22];
assign fin_o4 = tri_o4[22];
assign fin_o5 = tri_o5[22];
assign fin_o6 = tri_o6[22];
assign fin_o1 = tri_o1[23];
assign fin_o2 = tri_o2[23];
assign fin_o3 = tri_o3[23];
assign fin_o4 = tri_o4[23];
assign fin_o5 = tri_o5[23];
assign fin_o6 = tri_o6[23];
assign fin_o1 = tri_o1[24];
assign fin_o2 = tri_o2[24];
```

```
assign fin_o3 = tri_o3[24];
assign fin_o4 = tri_o4[24];
assign fin_o5 = tri_o5[24];
assign fin_o6 = tri_o6[24];
assign fin_o1 = tri_o1[25];
assign fin_o2 = tri_o2[25];
assign fin_o3 = tri_o3[25];
assign fin_o4 = tri_o4[25];
assign fin_o5 = tri_o5[25];
assign fin_o6 = tri_o6[25];

always @( i )
begin

`ifdef DC

romAddr[0]=17'b00000000000000000;
romAddr[1]=17'b10000000000000000;
romAddr[2]=17'b10000011101010011;
romAddr[3]=17'b10000111001110010;
romAddr[4]=17'b10001011000101111;
romAddr[5]=17'b10001111000100101;
romAddr[6]=17'b10010010111100011;
romAddr[7]=17'b10010111001001100;
romAddr[8]=17'b10011011011110010;
romAddr[9]=17'b10011111101011011;
romAddr[10]=17'b10100100001111111;
romAddr[11]=17'b10101000111100101;
romAddr[12]=17'b10101101100001001;
romAddr[13]=17'b10110010011110111;
romAddr[14]=17'b10110111100101110;
romAddr[15]=17'b10111100110110000;
romAddr[16]=17'b11000001111100111;
romAddr[17]=17'b11000111100000001;
romAddr[18]=17'b11001101001101100;
romAddr[19]=17'b11010010110000110;
romAddr[20]=17'b11011000110010110;
romAddr[21]=17'b11011110111111110;
romAddr[22]=17'b11100101000001110;
romAddr[23]=17'b11101011100101010;
romAddr[24]=17'b11110010010100101;
romAddr[25]=17'b11111000111000001;

  c[ rasize ] = 0;

`endif

  for( p = 0 ; p <= ( rasize - 1 ) ; p = p + 1 )
  begin
```

```
    c[ p ] = ( i >= romAddr[ p ] );

  end

  for( j = 0 ; j <= ( rasize - 1 ) ; j = j + 1 )
  begin

    d[ j ] <= ( c[ j ] != c[ j + 1 ] );

  end

end

`ifdef DC

always

`else

always @( d )

`endif

begin

`ifdef DC

// include data information here

romOut1[0]=17'b00000000000000000;
romOut2[0]=6'b010000;
romOut3[0]=5'b10000;
romOut4[0]=18'b000000000000000000;
romOut5[0]=6'b010000;
romOut6[0]=5'b10000;
romOut1[1]=17'b10000000000000000;
romOut2[1]=6'b010000;
romOut3[1]=5'b00000;
romOut4[1]=18'b010000011101010011;
romOut5[1]=6'b010001;
romOut6[1]=5'b01000;
romOut1[2]=17'b10000011101010011;
romOut2[2]=6'b010001;
romOut3[2]=5'b01000;
romOut4[2]=18'b010000111001110010;
romOut5[2]=6'b001111;
romOut6[2]=5'b10111;
romOut1[3]=17'b10000111001110010;
```

```
romOut2[3]=6'b001111;
romOut3[3]=5'b10111;
romOut4[3]=18'b010001011000101111;
romOut5[3]=6'b010000;
romOut6[3]=5'b11111;
romOut1[4]=17'b10001011000101111;
romOut2[4]=6'b010000;
romOut3[4]=5'b11111;
romOut4[4]=18'b010001111000100101;
romOut5[4]=6'b010001;
romOut6[4]=5'b00111;
romOut1[5]=17'b10001111000100101;
romOut2[5]=6'b010001;
romOut3[5]=5'b00111;
romOut4[5]=18'b010010010111100011;
romOut5[5]=6'b001111;
romOut6[5]=5'b10110;
romOut1[6]=17'b10010010111100011;
romOut2[6]=6'b001111;
romOut3[6]=5'b10110;
romOut4[6]=18'b010010111001001100;
romOut5[6]=6'b010000;
romOut6[6]=5'b11110;
romOut1[7]=17'b10010111001001100;
romOut2[7]=6'b010000;
romOut3[7]=5'b11110;
romOut4[7]=18'b010011011011110010;
romOut5[7]=6'b010001;
romOut6[7]=5'b00110;
romOut1[8]=17'b10011011011110010;
romOut2[8]=6'b010001;
romOut3[8]=5'b00110;
romOut4[8]=18'b010011111101011011;
romOut5[8]=6'b001111;
romOut6[8]=5'b10101;
romOut1[9]=17'b10011111101011011;
romOut2[9]=6'b001111;
romOut3[9]=5'b10101;
romOut4[9]=18'b010100100001111111;
romOut5[9]=6'b010000;
romOut6[9]=5'b11101;
romOut1[10]=17'b10100100001111111;
romOut2[10]=6'b010000;
romOut3[10]=5'b11101;
romOut4[10]=18'b010101000111100101;
romOut5[10]=6'b010001;
romOut6[10]=5'b00101;
romOut1[11]=17'b10101000111100101;
romOut2[11]=6'b010001;
```

```
romOut3[11]=5'b00101;
romOut4[11]=18'b010101101100001001;
romOut5[11]=6'b001111;
romOut6[11]=5'b10100;
romOut1[12]=17'b10101101100001001;
romOut2[12]=6'b001111;
romOut3[12]=5'b10100;
romOut4[12]=18'b010110010011110111;
romOut5[12]=6'b010000;
romOut6[12]=5'b11100;
romOut1[13]=17'b10110010011110111;
romOut2[13]=6'b010000;
romOut3[13]=5'b11100;
romOut4[13]=18'b010110111100101110;
romOut5[13]=6'b010001;
romOut6[13]=5'b00100;
romOut1[14]=17'b10110111100101110;
romOut2[14]=6'b010001;
romOut3[14]=5'b00100;
romOut4[14]=18'b010111100110110000;
romOut5[14]=6'b010010;
romOut6[14]=5'b01100;
romOut1[15]=17'b10111100110110000;
romOut2[15]=6'b010010;
romOut3[15]=5'b01100;
romOut4[15]=18'b011000001111100111;
romOut5[15]=6'b010000;
romOut6[15]=5'b11011;
romOut1[16]=17'b11000001111100111;
romOut2[16]=6'b010000;
romOut3[16]=5'b11011;
romOut4[16]=18'b011000111100000001;
romOut5[16]=6'b010001;
romOut6[16]=5'b00011;
romOut1[17]=17'b11000111100000001;
romOut2[17]=6'b010001;
romOut3[17]=5'b00011;
romOut4[17]=18'b011001101001101100;
romOut5[17]=6'b010010;
romOut6[17]=5'b01011;
romOut1[18]=17'b11001101001101100;
romOut2[18]=6'b010010;
romOut3[18]=5'b01011;
romOut4[18]=18'b011010010110000110;
romOut5[18]=6'b010000;
romOut6[18]=5'b11010;
romOut1[19]=17'b11010010110000110;
romOut2[19]=6'b010000;
romOut3[19]=5'b11010;
```

```
romOut4[19]=18'b011011000110010110;
romOut5[19]=6'b010001;
romOut6[19]=5'b00010;
romOut1[20]=17'b11011000110010110;
romOut2[20]=6'b010001;
romOut3[20]=5'b00010;
romOut4[20]=18'b011011110111111110;
romOut5[20]=6'b010010;
romOut6[20]=5'b01010;
romOut1[21]=17'b11011110111111110;
romOut2[21]=6'b010010;
romOut3[21]=5'b01010;
romOut4[21]=18'b011100101000001110;
romOut5[21]=6'b010000;
romOut6[21]=5'b11001;
romOut1[22]=17'b11100101000001110;
romOut2[22]=6'b010000;
romOut3[22]=5'b11001;
romOut4[22]=18'b011101011100101010;
romOut5[22]=6'b010001;
romOut6[22]=5'b00001;
romOut1[23]=17'b11101011100101010;
romOut2[23]=6'b010001;
romOut3[23]=5'b00001;
romOut4[23]=18'b011110010010100101;
romOut5[23]=6'b010010;
romOut6[23]=5'b01001;
romOut1[24]=17'b11110010010100101;
romOut2[24]=6'b010010;
romOut3[24]=5'b01001;
romOut4[24]=18'b011111000111000001;
romOut5[24]=6'b010000;
romOut6[24]=5'b11000;
romOut1[25]=17'b11111000111000001;
romOut2[25]=6'b010000;
romOut3[25]=5'b11000;
romOut4[25]=18'b100000000000000000;
romOut5[25]=6'b010001;
romOut6[25]=5'b00000;

  high = 31'bz;

`else

`endif

  for( k = 0 ; k <= ( rasize - 1 ) ; k = k + 1 )
  begin
```

```
    if( d[ k ] )
    begin

      tri_o1[ k ] = romOut1[ k ];
      tri_o2[ k ] = romOut2[ k ];
      tri_o3[ k ] = romOut3[ k ];
      tri_o4[ k ] = romOut4[ k ];
      tri_o5[ k ] = romOut5[ k ];
      tri_o6[ k ] = romOut6[ k ];

    end
    else
    begin

      tri_o1[ k ] = high[ o1bits - 1 : 0 ];
      tri_o2[ k ] = high[ o2bits - 1 : 0 ];
      tri_o3[ k ] = high[ o3bits - 1 : 0 ];
      tri_o4[ k ] = high[ o4bits - 1 : 0 ];
      tri_o5[ k ] = high[ o5bits - 1 : 0 ];
      tri_o6[ k ] = high[ o6bits - 1 : 0 ];

    end

  end

end

`ifdef DC
always o1 = fin_o1;
always o2 = fin_o2;
always o3 = fin_o3;
always o4 = fin_o4;
always o5 = fin_o5;
always o6 = fin_o6;
`else
always @( fin_o1 ) o1 = fin_o1;
always @( fin_o2 ) o2 = fin_o2;
always @( fin_o3 ) o3 = fin_o3;
always @( fin_o4 ) o4 = fin_o4;
always @( fin_o5 ) o5 = fin_o5;
always @( fin_o6 ) o6 = fin_o6;
`endif

endmodule

// RALUT contents ( address, output(s) ... )

/*
```

```
0000000000000000 0000000000000000 010000 10000 0000000000000000
    010000 10000
1000000000000000 1000000000000000 010000 00000 0100000111010011
    010001 01000
1000001110101011 1000001110101011 010001 01000 0100001110011100 10
    001111 10111
1000011100111001 0 1000011100111001 0 001111 10111 0100010110001011 11
    010000 11111
1000101100010111 1000101100010111 010000 11111 0100011110001001 01
    010001 00111
1000111100010010 1000111100010010 010001 00111 0100100101111000 11
    001111 10110
1001001011110001 1001001011110001 001111 10110 0100101110010011 00
    010000 11110
1001011100100110 0 1001011100100110 0 010000 11110 0100110110111100 10
    010001 00110
1001101101111001 0 1001101101111001 0 010001 00110 0100111111010110 11
    001111 10101
1001111110101101 1 1001111110101101 1 001111 10101 0101001000011111 11
    010000 11101
1010010000111111 1 1010010000111111 1 010000 11101 0101010001111001 01
    010001 00101
1010100011110010 1 1010100011110010 1 010001 00101 0101011001000010 01
    001111 10100
1010110110000100 1 1010110110000100 1 001111 10100 0101100100111101 11
    010000 11100
1011001001111011 1 1011001001111011 1 010000 11100 0101101111001011 10
    010001 00100
1011011110010111 0 1011011110010111 0 010001 00100 0101111001101100 00
    010010 01100
1011110011011000 0 1011110011011000 0 010010 01100 0110000011111001 11
    010000 11011
1100000111110011 1 1100000111110011 1 010000 11011 0110001111000000 01
    010001 00011
1100011110000000 1 1100011110000000 1 010001 00011 0110011010011011 00
    010010 01011
1100110100110110 0 1100110100110110 0 010010 01011 0110100101100001 10
    010000 11010
1101001011000011 0 1101001011000011 0 010000 11010 0110110001100101 10
    010001 00010
1101100011001011 0 1101100011001011 0 010001 00010 0110111101111111 10
    010010 01010
1101111011111111 0 1101111011111111 0 010010 01010 0111001010000011 10
    010000 11001
1110010100000111 0 1110010100000111 0 010000 11001 0111010111001010 10
    010001 00001
1110101110010101 0 1110101110010101 0 010001 00001 0111100100101001 01
    010010 01001
1111001001010010 1 1111001001010010 1 010010 01001 0111110001110000 01
```

```
   010000 11000
1111100011100000 1 1111100011100000 1 010000 11000 1000000000000000 0
   010001 00000

*/
```

A.2.13 The TLNS Binary / 2DLNS Conversion Register

This register reorders the output of the binary / 2DLNS converter.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity conv_out_reg is
  port ( ready : in std_logic;
         output_sign : in std_logic_vector(1 downto 0);
         output_first : in std_logic_vector(11 downto 0);
         output_second : in std_logic_vector(9 downto 0);
         tlns_output : out tlns_bus_word);
end entity conv_out_reg;


architecture behavior of conv_out_reg is

  signal stored_value : tlns_bus_word;

begin

  -- extracts the data related to each digit from inputs
  -- and merges them as an stored value
  conv_output : process ( output_sign,output_first,output_second ) is
  begin
       stored_value <= output_sign(1) & output_first(11 downto 6) &
                       output_second(9 downto 5) & output_sign(0) &
                       output_first(5 downto 0) &
                       output_second(4 downto 0);
  end process conv_output;

  -- stored value enabled to output based on ready signal
  tlns_output <= stored_value when ready = '1' else
                 disabled_tlns_word;

end architecture behavior;
```

A.2.14 The Multiply and Accumulate unit (MAC)

The MAC unit consists of several components. All components are instantiated in a top module which is shown here. The consecutive subsections include the HDL codes of all components.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity mac is
  port (clk,clr : in std_logic;
        channel_mux_sel : in std_logic;
        coefnum : in std_logic;
        evensym : in std_logic;
        x : in tlns_bus_word;
        y : in tlns_bus_word;
        p : out tlns_bus_word);
end entity mac;


architecture rtl of mac is

  -- The xor unit
  component x_or
    port ( a, b : in std_logic;
           c : out std_logic );
    end component ;

  -- The first exponent adder
  component adder
    port ( a, b : in std_logic_vector(5 downto 0);
           s : out std_logic_vector(6 downto 0) );
    end component;

  -- The second exponent adder
  component adder_r
    port ( a : in std_logic_vector(4 downto 0);
           b : in std_logic_vector(2 downto 0);
           s : out std_logic_vector(4 downto 0) );
  end component;

  -- The 2DLNS / Binary Converter
  component  convert2dlnstobinary
```

```
  port ( signin : in std_logic;
         firstbaseindex : in std_logic_vector(6 downto 0);
         secondbaseindex: in std_logic_vector(4 downto 0);
         binaryout : out std_logic_vector(19 downto 0);
         signout : out std_logic );
end component;

-- The 20-bit adder_subtracter
component adder_subtracter_20
  port ( a, b : in std_logic_vector(19 downto 0);
         s : out std_logic_vector(20 downto 0);
         sign1, sign2 : in std_logic );
  end component;

-- The 21-bit adder_subtracter
component adder_subtracter_21
  port ( a, b : in std_logic_vector(20 downto 0);
         s : out std_logic_vector(21 downto 0);
         sign1, sign2 : in std_logic );
 end component;

-- The 23-bit adder_subtracter for low channel
component adder_subtracter_23_low
  port ( a : in std_logic_vector(21 downto 0);
         b : in std_logic_vector(23 downto 0);
         s : out std_logic_vector(23 downto 0);
         sign1 : in std_logic );
  end component;

-- The 23-bit adder_subtracter for high channel
component adder_subtracter_23_high
  port ( a : in std_logic_vector(21 downto 0);
         b : in std_logic_vector(23 downto 0);
         s : out std_logic_vector(23 downto 0);
         sign1 : in std_logic;
        num : in std_logic;
        sym : in std_logic );
  end component;

-- The register in feedback loop of accumulator
component accumulator_regp
  port ( clk, clr : in std_logic;
         d : in std_logic_vector(23 downto 0);
         q : out std_logic_vector(23 downto 0));
end component;

-- The output multiplexer of channels
component muxlowhigh
  port ( i0 : in std_logic_vector(23 downto 0);
```

```
            i1 : in std_logic_vector(23 downto 0);
            y : out std_logic_vector(23 downto 0);
            sel : in std_logic);
  end component;

  -- The register which latches the high channel output
  component channelhigh_reg is
    port (clk : in std_logic;
          d : in std_logic_vector(23 downto 0);
          q : out std_logic_vector(23 downto 0));
  end component channelhigh_reg;

  signal p_p1_dig1 : std_logic_vector(12 downto 0);
  signal p_p1_dig2 : std_logic_vector(12 downto 0);
  signal p_p2_dig1 : std_logic_vector(12 downto 0);
  signal p_p2_dig2 : std_logic_vector(12 downto 0);
  signal p_p1_b1 : std_logic_vector(19 downto 0);
  signal p_p1_b2 : std_logic_vector(19 downto 0);
  signal p_p2_b1 : std_logic_vector(19 downto 0);
  signal p_p2_b2 : std_logic_vector(19 downto 0);
  signal p_p1_b : std_logic_vector(20 downto 0);
  signal p_p2_b : std_logic_vector(20 downto 0);
  signal p_b : std_logic_vector(21 downto 0);
  signal channellow : std_logic_vector(23 downto 0);
  signal channelhigh : std_logic_vector(23 downto 0);
  signal channelhigh_out : std_logic_vector(23 downto 0);
  signal acc_channellow : std_logic_vector(23 downto 0);
  signal acc_channelhigh : std_logic_vector(23 downto 0);
  signal outchannel : std_logic_vector(23 downto 0);
  signal s1, s2, s3, s4 : std_logic;
  signal dummy_s1, dummy_s2, dummy_s3, dummy_s4 : std_logic;

begin

  -- The component instantiations

  p_p1_dig1_s :  x_or
    port map ( a => x(23), b => y(19), c => p_p1_dig1(12));

  s1 <= p_p1_dig1(12);

  p_p1_dig2_s :  x_or
    port map ( a => x(11), b => y(9), c => p_p1_dig2(12));

  s2 <= p_p1_dig2(12);


  p_p2_dig1_s :  x_or
    port map ( a => x(11), b => y(19), c => p_p2_dig1(12));
```

```
 s3 <= p_p2_dig1(12);

p_p2_dig2_s :  x_or
  port map ( a => x(23), b => y(9), c => p_p2_dig2(12));

s4 <= p_p2_dig2(12);

p_p1_dig1_a :  adder
  port map ( a => x(22 downto 17), b => y(18 downto 13),
             s => p_p1_dig1(11 downto 5));

p_p1_dig1_b :  adder_r
  port map ( a => x(16 downto 12), b => y(12 downto 10),
             s => p_p1_dig1(4 downto 0));

p_p1_dig2_a :  adder
  port map ( a => x(10 downto 5), b => y(8 downto 3),
             s => p_p1_dig2(11 downto 5));

p_p1_dig2_b :  adder_r
  port map ( a => x(4 downto 0), b => y(2 downto 0),
             s => p_p1_dig2(4 downto 0));

p_p2_dig1_a :  adder
  port map ( a => x(10 downto 5), b => y(18 downto 13),
             s => p_p2_dig1(11 downto 5));

p_p2_dig1_b :  adder_r
  port map ( a => x(4 downto 0), b => y(12 downto 10),
             s => p_p2_dig1(4 downto 0));

p_p2_dig2_a :  adder
  port map ( a => x(22 downto 17), b => y(8 downto 3),
             s => p_p2_dig2(11 downto 5));

p_p2_dig2_b :  adder_r
  port map ( a => x(16 downto 12), b => y(2 downto 0),
             s => p_p2_dig2(4 downto 0));


p_p1_b1_con :  convert2dlnstobinary
  port map ( signin => p_p1_dig1(12),
             firstbaseindex => p_p1_dig1(11 downto 5),
             secondbaseindex => p_p1_dig1(4 downto 0),
             binaryout => p_p1_b1(19 downto 0),
             signout => dummy_s1);

p_p1_b2_con :  convert2dlnstobinary
```

```
  port map ( signin => p_p1_dig2(12),
             firstbaseindex => p_p1_dig2(11 downto 5),
             secondbaseindex => p_p1_dig2(4 downto 0),
             binaryout => p_p1_b2(19 downto 0),
             signout => dummy_s2);

p_p2_b1_con :  convert2dlnstobinary
  port map ( signin => p_p2_dig1(12),
             firstbaseindex => p_p2_dig1(11 downto 5),
             secondbaseindex => p_p2_dig1(4 downto 0),
             binaryout => p_p2_b1(19 downto 0),
             signout => dummy_s3);

p_p2_b2_con :  convert2dlnstobinary
  port map ( signin => p_p2_dig2(12),
             firstbaseindex => p_p2_dig2(11 downto 5),
             secondbaseindex => p_p2_dig2(4 downto 0),
             binaryout => p_p2_b2(19 downto 0),
             signout => dummy_s4);


p1_bin_add_sub :  adder_subtracter_20
  port map ( a => p_p1_b1, b => p_p1_b2,
             s => p_p1_b, sign1 => s1, sign2 => s2);

p2_bin_add_sub :  adder_subtracter_20
  port map ( a => p_p2_b1, b => p_p2_b2,
             s => p_p2_b, sign1 => s3, sign2 => s4);


p_bin_add_sub :  adder_subtracter_21
  port map ( a => p_p1_b, b => p_p2_b, s => p_b, sign1 => s1,
             sign2 => s3);


add_sub_low :  adder_subtracter_23_low
  port map ( a => p_b, b => acc_channellow, s => channellow,
             sign1 => s1);


channellow_accumulator_reg :  accumulator_regp
  port map ( clk => clk, clr => clr, d => channellow,
             q => acc_channellow);


add_sub_high :  adder_subtracter_23_high
  port map ( a => p_b, b => acc_channelhigh, s => channelhigh,
             sign1 => s1, num => coefnum, sym => evensym);
```

```
  channelhigh_accumulator_reg :  accumulator_regp
    port map ( clk => clk, clr => clr, d => channelhigh,
               q => acc_channelhigh);


  channelhigh_out_reg :  channelhigh_reg
    port map ( clk => clk, d => channelhigh, q => channelhigh_out);


  mux :  muxlowhigh
    port map ( i0 => channellow, i1 => channelhigh_out,
               y => outchannel, sel => channel_mux_sel);

  -- removes the repeatative bits and disables output based on clr
     signal
  with clr select
    p <=  outchannel(23) & outchannel(23) & outchannel(23) & outchannel
       (23) &
          outchannel(23 downto 4) when '0',
          disabled_tlns_word  when others;

end architecture rtl;
```

A.2.14.1 The Exclusive-or unit

This code, simply makes an exclusive-or of its inputs.

```
library ieee;
use ieee.std_logic_1164.all;

entity x_or is
  port ( a,b :   in std_logic;
         c :     out std_logic );
end entity x_or;


architecture behavioral of x_or is
begin
      c <= a xor b;

end architecture behavioral;
```

A.2.14.2 The First Exponent Adders

This code shows a two's-complement bit by bit adder. The MAC unit makes use of this adder to add the first base exponents.

```
library ieee;
use ieee.std_logic_1164.all;

entity adder is
  port ( a, b : in std_logic_vector(5 downto 0);
         s :    out std_logic_vector(6 downto 0) );
end entity adder;


architecture behavioral of adder is
begin

  behavior : process (a, b) is

    variable carry_in : std_logic;
    variable carry_out : std_logic;

  begin
    -- carry in to the first bit
    carry_out := '0';

    -- computes sum and carry for all bits
    -- carry out of each order is carry in for the next one
    for index in 0 to a'left loop
      carry_in := carry_out;
      s(index) <= a(index) xor b(index) xor carry_in ;
      carry_out := (a(index) and b(index))
                   or (carry_in and (a(index) xor b(index)));
    end loop;

    -- one extra bit is considered for sum
    s(a'left + 1) <= a(a'left) xor b(a'left) xor carry_out ;
  end process behavior;

end architecture behavioral;
```

A.2.14.3 The Second Exponent Adders

Since the size of operands of this adder are different, the smaller operand is sign extended before addition. In our design, the most negative second base exponent is used to represent zero. Therefore, if either multiplicand or multiplier are zero, the product will be set to zero.

```
library ieee;
use ieee.std_logic_1164.all;

entity adder_r is
  port ( a : in std_logic_vector(4 downto 0);
         b : in std_logic_vector(2 downto 0);
         s : out std_logic_vector(4 downto 0) );
end entity adder_r;


architecture behavioral of adder_r is

begin

  behavior : process (a, b) is

    variable carry_in : std_logic;
    variable carry_out : std_logic;
    variable ext_b : std_logic_vector(4 downto 0);

  begin

  -- checks if either of inputs represents zero,
  -- the product should also be set to zero
  if a = "10000" or b = "100" then
                              s <= "10000" ;
  else
    -- carry in to the first bit
    carry_out := '0';

    -- operand b is sign extended
    ext_b(4 downto 0) := b(2) & b(2) & b(2 downto 0);

    -- computes sum and carry for all bits
    -- carry out of each order is carry in for the next one
    for index in 0 to a'left loop
      carry_in := carry_out;
      s(index) <= a(index) xor ext_b(index) xor carry_in ;
```

```
      carry_out := (a(index) and ext_b(index))
                   or (carry_in and (a(index) xor ext_b(index)));
    end loop;
  end if;
  end process behavior;

end architecture behavioral;
```

A.2.14.4 The 2DLNS / Binary Converter

The Verilog code in [13], is used for the 2DLNS / binary conversion. This HDL code has also been written fully parametrized and the parameters should be set when the shell script which generates Verilog module is run. The definition of these parameters were also included in [13]. Before running this script, the optimal base has been computed and stored in an ASCII file, 32768-13mn2unz.out. The 2DLNS / binary converter module in TLNS CPU has been generated by setting the parameters as:

makeconvert2dlnstobinary.sh 32768-13mn2unz.out 16 7 5 4 -nz -ns > converter.v

The name of generated module is **convert2dlnstobinary** which is instantiated, as an component, in the MAC top module. This Verilog file is shown here.

```
'timescale 1ns/10ps
module convert2dlnstobinary(
signin,
firstbaseindex,
secondbaseindex,
binaryout,
signout
);
// Default parameters
parameter firstbasebits = 7;
parameter secondbasebits = 5;
parameter outputbits = 16;
parameter extrabits = 4;
parameter memfirstbasebits = 6;
parameter subbits = 8;
parameter twobitmode = 0;
```

```
parameter nosignmode = 0;
// Define ports
input [ twobitmode : 0 ] signin;
input [ firstbasebits - 1 : 0 ] firstbaseindex;
input [ secondbasebits - 1 : 0 ] secondbaseindex;
output [ nosignmode + outputbits + extrabits - 1 : 0 ] binaryout;
reg [ nosignmode + outputbits + extrabits - 1 : 0 ] binaryout;
output signout;
reg signout;
reg [ subbits - 1 : 0 ] arg1, arg2, sum;
reg [ outputbits + extrabits - 1 : 0 ] absolute;
wire [ outputbits + extrabits - 1 : 0 ] mantissa;
wire [ memfirstbasebits - 1 : 0 ] shift;
lut2_452845339_1088_noclk
lut
(
mantissa,
shift,
secondbaseindex
);
// Perform shift
always @( mantissa or shift or firstbaseindex )
begin
// Extend signs on each argument
if( subbits > memfirstbasebits )

begin
arg1 = { { ( subbits - memfirstbasebits ) { shift[ memfirstbasebits - 1
    ] }
}, shift[ memfirstbasebits - 1 : 0 ] };
end
else
begin
arg1 = shift;
end
if( subbits > firstbasebits )
begin
arg2 = { { ( subbits - firstbasebits ) { firstbaseindex[ firstbasebits -
    1 ]
} }, firstbaseindex[ firstbasebits - 1 : 0 ] };
end
else
begin
arg2 = firstbaseindex;
end
sum = arg1 - arg2;
absolute = mantissa >> sum;
if( twobitmode )
begin
```

```
if( signin == 0 )
begin
absolute = 0;
end
end
if( nosignmode )
begin
if( signin[ twobitmode ] )
begin
binaryout <= - absolute;
end
else
begin
binaryout <= absolute;
end

end
else
begin
binaryout <= absolute;
signout <= signin[ twobitmode ];
end
// $display("@@ %b %b %b %b %b %b %b",mantissa, shift, arg1, firstbaseindex
   ,
// arg2,sum,binaryout);
end
endmodule
module lut2_452845339_1088_noclk(
  o1,
  o2,
  id
);

// Default parameters
parameter o1bits = 20;
parameter o2bits = 6;
parameter idbits = 5;
parameter rasize = 32;

// Define ports

// direct data access line
input [ idbits - 1 : 0 ] id;

// datas to output
output [ o1bits - 1 : 0 ] o1;
output [ o2bits - 1 : 0 ] o2;
reg [ o1bits - 1 : 0 ] o1;
reg [ o2bits - 1 : 0 ] o2;
```

```
reg [ rasize - 1 : 0 ] d;

reg [ idbits - 1 : 0 ] romAddr[ rasize - 1 : 0 ];
reg [ o1bits - 1 : 0 ] romOut1[ rasize - 1 : 0 ];
reg [ o2bits - 1 : 0 ] romOut2[ rasize - 1 : 0 ];

reg [ 30 : 0 ] high;

wire [ o1bits - 1 : 0 ] fin_o1;
wire [ o2bits - 1 : 0 ] fin_o2;
reg [ o1bits - 1 : 0 ] tri_o1[ rasize - 1 : 0 ];
reg [ o2bits - 1 : 0 ] tri_o2[ rasize - 1 : 0 ];

integer p;
integer j;
integer k;

`ifdef DC
`else

initial
begin

// include data information here

romAddr[0]=5'b00000;
romAddr[1]=5'b00001;
romAddr[2]=5'b00010;
romAddr[3]=5'b00011;
romAddr[4]=5'b00100;
romAddr[5]=5'b00101;
romAddr[6]=5'b00110;
romAddr[7]=5'b00111;
romAddr[8]=5'b01000;
romAddr[9]=5'b01001;
romAddr[10]=5'b01010;
romAddr[11]=5'b01011;
romAddr[12]=5'b01100;
romAddr[13]=5'b01101;
romAddr[14]=5'b01110;
romAddr[15]=5'b01111;
romAddr[16]=5'b10000;
romAddr[17]=5'b10001;
romAddr[18]=5'b10010;
romAddr[19]=5'b10011;
romAddr[20]=5'b10100;
romAddr[21]=5'b10101;
romAddr[22]=5'b10110;
```

```
romAddr[23]=5'b10111;
romAddr[24]=5'b11000;
romAddr[25]=5'b11001;
romAddr[26]=5'b11010;
romAddr[27]=5'b11011;
romAddr[28]=5'b11100;
romAddr[29]=5'b11101;
romAddr[30]=5'b11110;
romAddr[31]=5'b11111;

romOut1[0]=20'b10000000000000000000;
romOut2[0]=6'b001111;
romOut1[1]=20'b11101011100101010001;
romOut2[1]=6'b010000;
romOut1[2]=20'b11011000110010110001;
romOut2[2]=6'b010000;
romOut1[3]=20'b11000111100000001011;
romOut2[3]=6'b010000;
romOut1[4]=20'b10110111100101110101;
romOut2[4]=6'b010000;
romOut1[5]=20'b10101000111100101101;
romOut2[5]=6'b010000;
romOut1[6]=20'b10011011011110010101;
romOut2[6]=6'b010000;
romOut1[7]=20'b10001111000100101110;
romOut2[7]=6'b010000;
romOut1[8]=20'b10000011101010011011;
romOut2[8]=6'b010000;
romOut1[9]=20'b11110010010100101111;
romOut2[9]=6'b010001;
romOut1[10]=20'b11011110111111110100;
romOut2[10]=6'b010001;
romOut1[11]=20'b11001101001101100011;
romOut2[11]=6'b010001;
romOut1[12]=20'b10111100110110000100;
romOut2[12]=6'b010001;
romOut1[13]=20'b10101101110010001000;
romOut2[13]=6'b010001;
romOut1[14]=20'b10011111111011000100;
romOut2[14]=6'b010001;
romOut1[15]=20'b10010011001010110001;
romOut2[15]=6'b010001;
romOut1[16]=20'b00000000000000000000;
romOut2[16]=6'b001111;
romOut1[17]=20'b11011110101010000010;
romOut2[17]=6'b001110;
romOut1[18]=20'b11001100111001100000;
romOut2[18]=6'b001110;
romOut1[19]=20'b10111100100011100111;
```

```
romOut2[19]=6'b001110;
romOut1[20]=20'b10101101100001001001;
romOut2[20]=6'b001110;
romOut1[21]=20'b10011111101011011100;
romOut2[21]=6'b001110;
romOut1[22]=20'b10010010111100011000;
romOut2[22]=6'b001110;
romOut1[23]=20'b10000111001110010101;
romOut2[23]=6'b001110;
romOut1[24]=20'b11111000111000001100;
romOut2[24]=6'b001111;
romOut1[25]=20'b11100101000001110100;
romOut2[25]=6'b001111;
romOut1[26]=20'b11010010110000110001;
romOut2[26]=6'b001111;
romOut1[27]=20'b11000001111100111101;
romOut2[27]=6'b001111;
romOut1[28]=20'b10110010011110111100;
romOut2[28]=6'b001111;
romOut1[29]=20'b10100100001111111001;
romOut2[29]=6'b001111;
romOut1[30]=20'b10010111001001100000;
romOut2[30]=6'b001111;
romOut1[31]=20'b10001011000101111111;
romOut2[31]=6'b001111;

// Verify that the addresses are in order

  for( p = 0 ; p <= ( rasize - 1) ; p = p + 1 )
  begin

    if( romAddr[ p ] != p )
    begin

      $stop;

    end

  end

  high = 31'bz;

end
`endif

// include line shorting here

assign fin_o1 = tri_o1[0];
assign fin_o2 = tri_o2[0];
```

```
assign fin_o1 = tri_o1[1];
assign fin_o2 = tri_o2[1];
assign fin_o1 = tri_o1[2];
assign fin_o2 = tri_o2[2];
assign fin_o1 = tri_o1[3];
assign fin_o2 = tri_o2[3];
assign fin_o1 = tri_o1[4];
assign fin_o2 = tri_o2[4];
assign fin_o1 = tri_o1[5];
assign fin_o2 = tri_o2[5];
assign fin_o1 = tri_o1[6];
assign fin_o2 = tri_o2[6];
assign fin_o1 = tri_o1[7];
assign fin_o2 = tri_o2[7];
assign fin_o1 = tri_o1[8];
assign fin_o2 = tri_o2[8];
assign fin_o1 = tri_o1[9];
assign fin_o2 = tri_o2[9];
assign fin_o1 = tri_o1[10];
assign fin_o2 = tri_o2[10];
assign fin_o1 = tri_o1[11];
assign fin_o2 = tri_o2[11];
assign fin_o1 = tri_o1[12];
assign fin_o2 = tri_o2[12];
assign fin_o1 = tri_o1[13];
assign fin_o2 = tri_o2[13];
assign fin_o1 = tri_o1[14];
assign fin_o2 = tri_o2[14];
assign fin_o1 = tri_o1[15];
assign fin_o2 = tri_o2[15];
assign fin_o1 = tri_o1[16];
assign fin_o2 = tri_o2[16];
assign fin_o1 = tri_o1[17];
assign fin_o2 = tri_o2[17];
assign fin_o1 = tri_o1[18];
assign fin_o2 = tri_o2[18];
assign fin_o1 = tri_o1[19];
assign fin_o2 = tri_o2[19];
assign fin_o1 = tri_o1[20];
assign fin_o2 = tri_o2[20];
assign fin_o1 = tri_o1[21];
assign fin_o2 = tri_o2[21];
assign fin_o1 = tri_o1[22];
assign fin_o2 = tri_o2[22];
assign fin_o1 = tri_o1[23];
assign fin_o2 = tri_o2[23];
assign fin_o1 = tri_o1[24];
assign fin_o2 = tri_o2[24];
assign fin_o1 = tri_o1[25];
```

```
assign fin_o2 = tri_o2[25];
assign fin_o1 = tri_o1[26];
assign fin_o2 = tri_o2[26];
assign fin_o1 = tri_o1[27];
assign fin_o2 = tri_o2[27];
assign fin_o1 = tri_o1[28];
assign fin_o2 = tri_o2[28];
assign fin_o1 = tri_o1[29];
assign fin_o2 = tri_o2[29];
assign fin_o1 = tri_o1[30];
assign fin_o2 = tri_o2[30];
assign fin_o1 = tri_o1[31];
assign fin_o2 = tri_o2[31];

always @( id )
begin

`ifdef DC

romAddr[0]=5'b00000;
romAddr[1]=5'b00001;
romAddr[2]=5'b00010;
romAddr[3]=5'b00011;
romAddr[4]=5'b00100;
romAddr[5]=5'b00101;
romAddr[6]=5'b00110;
romAddr[7]=5'b00111;
romAddr[8]=5'b01000;
romAddr[9]=5'b01001;
romAddr[10]=5'b01010;
romAddr[11]=5'b01011;
romAddr[12]=5'b01100;
romAddr[13]=5'b01101;
romAddr[14]=5'b01110;
romAddr[15]=5'b01111;
romAddr[16]=5'b10000;
romAddr[17]=5'b10001;
romAddr[18]=5'b10010;
romAddr[19]=5'b10011;
romAddr[20]=5'b10100;
romAddr[21]=5'b10101;
romAddr[22]=5'b10110;
romAddr[23]=5'b10111;
romAddr[24]=5'b11000;
romAddr[25]=5'b11001;
romAddr[26]=5'b11010;
romAddr[27]=5'b11011;
romAddr[28]=5'b11100;
romAddr[29]=5'b11101;
```

```
romAddr[30]=5'b11110;
romAddr[31]=5'b11111;

`endif

  for( j = 0 ; j <= ( rasize - 1) ; j = j + 1 )
  begin

    d[ j ] <= ( j == id ? 1 : 0 );

  end

end

`ifdef DC

always

`else

always @( d )

`endif

begin

`ifdef DC

// include data information here

romOut1[0]=20'b10000000000000000000;
romOut2[0]=6'b001111;
romOut1[1]=20'b11101011100101010001;
romOut2[1]=6'b010000;
romOut1[2]=20'b11011000110010110001;
romOut2[2]=6'b010000;
romOut1[3]=20'b11000111100000001011;
romOut2[3]=6'b010000;
romOut1[4]=20'b10110111100101110101;
romOut2[4]=6'b010000;
romOut1[5]=20'b10101000111100101101;
romOut2[5]=6'b010000;
romOut1[6]=20'b10011011011110010101;
romOut2[6]=6'b010000;
romOut1[7]=20'b10001111000100101110;
romOut2[7]=6'b010000;
romOut1[8]=20'b10000011101010011011;
romOut2[8]=6'b010000;
romOut1[9]=20'b11110010010100101111;
```

```
romOut2[9]=6'b010001;
romOut1[10]=20'b11011110111111110100;
romOut2[10]=6'b010001;
romOut1[11]=20'b11001101001101100011;
romOut2[11]=6'b010001;
romOut1[12]=20'b10111100110110000100;
romOut2[12]=6'b010001;
romOut1[13]=20'b10101101110010001000;
romOut2[13]=6'b010001;
romOut1[14]=20'b10011111111011000100;
romOut2[14]=6'b010001;
romOut1[15]=20'b10010011001010110001;
romOut2[15]=6'b010001;
romOut1[16]=20'b00000000000000000000;
romOut2[16]=6'b001111;
romOut1[17]=20'b11011110101010000010;
romOut2[17]=6'b001110;
romOut1[18]=20'b11001100111001100000;
romOut2[18]=6'b001110;
romOut1[19]=20'b10111100100011100111;
romOut2[19]=6'b001110;
romOut1[20]=20'b10101101100001001001;
romOut2[20]=6'b001110;
romOut1[21]=20'b10011111101011011100;
romOut2[21]=6'b001110;
romOut1[22]=20'b10010010111100011000;
romOut2[22]=6'b001110;
romOut1[23]=20'b10000111001110010101;
romOut2[23]=6'b001110;
romOut1[24]=20'b11111000111000001100;
romOut2[24]=6'b001111;
romOut1[25]=20'b11100101000001110100;
romOut2[25]=6'b001111;
romOut1[26]=20'b11010010110000110001;
romOut2[26]=6'b001111;
romOut1[27]=20'b11000001111100111101;
romOut2[27]=6'b001111;
romOut1[28]=20'b10110010011110111100;
romOut2[28]=6'b001111;
romOut1[29]=20'b10100100001111111001;
romOut2[29]=6'b001111;
romOut1[30]=20'b10010111001001100000;
romOut2[30]=6'b001111;
romOut1[31]=20'b10001011000101111111;
romOut2[31]=6'b001111;

  high = 31'bz;

`else
```

```
`endif

  for( k = 0 ; k <= ( rasize - 1 ) ; k = k + 1 )
  begin

    if( d[ k ] )
    begin

      tri_o1[ k ] = romOut1[ k ];
      tri_o2[ k ] = romOut2[ k ];

    end
    else
    begin

      tri_o1[ k ] = high[ o1bits - 1 : 0 ];
      tri_o2[ k ] = high[ o2bits - 1 : 0 ];

    end

  end

end

`ifdef DC
always o1 = fin_o1;
always o2 = fin_o2;
`else
always @( fin_o1 ) o1 = fin_o1;
always @( fin_o2 ) o2 = fin_o2;
`endif

endmodule

// LUT contents ( address, output(s) ... )

/*

00000   10000000000000000000   001111
00001   11101011100101010001   010000
00010   11011000110010110001   010000
00011   11000111100000001011   010000
00100   10110111100101110101   010000
00101   10101000111100101101   010000
00110   10011011011110010101   010000
00111   10001111000100101110   010000
01000   10000011101010011011   010000
01001   11110010010100101111   010001
```

```
01010   11011110111111110100    010001
01011   11001101001101100011    010001
01100   10111100110110000100    010001
01101   10101101110010001000    010001
01110   10011111111011000100    010001
01111   10010011001010110001    010001
10000   00000000000000000000    001111
10001   11011110101010000010    001110
10010   11001100111001100000    001110
10011   10111100100011100111    001110
10100   10101101100001001001    001110
10101   10011111101011011100    001110
10110   10010010111100011000    001110
10111   10000111001110010101    001110
11000   11111000111000001100    001111
11001   11100101000001110100    001111
11010   11010010110000110001    001111
11011   11000001111100111101    001111
11100   10110010011110111100    001111
11101   10100100001111111001    001111
11110   10010111001001100000    001111
11111   10001011000101111111    001111

*/
```

A.2.14.5 20-bit Adder / Subtracter

In this module, the type of operands are converted to **unsigned**. Therefore, addition or subtraction of unsigned values, are performed using the functions which have been defined in the **numeric_bit** package.

```
library ieee;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity adder_subtracter_20 is
  port ( a, b : in std_logic_vector(19 downto 0);
         s : out std_logic_vector(20 downto 0);
         sign1, sign2 : in std_logic );
end entity adder_subtracter_20;


architecture behavioral of adder_subtracter_20 is
begin
```

```
  behavior : process (a, b) is

    variable sign : std_logic;
    variable ext_a : unsigned(20 downto 0);
    variable ext_b : unsigned(20 downto 0);
    variable result : unsigned(20 downto 0);

  begin

    -- the sign is xor of input signs
    sign := sign1 xor sign2;

    -- the operands are unsigned extended by one zero bit
    ext_a := unsigned(to_bitvector("0" & a(19 downto 0)));
    ext_b := unsigned(to_bitvector("0" & b(19 downto 0)));

    -- addition or subtraction is done based on the computed sign
    if sign = '1' then
       result := ext_a - ext_b ;
    else
       result := ext_a + ext_b ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;

end architecture behavioral;
```

A.2.14.6 21-bit Adder / Subtracter

The only difference with the previous module is the size of operands and result.

```
library ieee;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity adder_subtracter_21 is
  port ( a, b : in std_logic_vector(20 downto 0);
         s : out std_logic_vector(21 downto 0);
         sign1, sign2 : in std_logic );
end entity adder_subtracter_21;


architecture behavioral of adder_subtracter_21 is
```

```
begin

  behavior : process (a, b) is

    variable sign : std_logic;
    variable ext_a : unsigned(21 downto 0);
    variable ext_b : unsigned(21 downto 0);
    variable result : unsigned(21 downto 0);

  begin

    -- the sign is xor of input signs
    sign := sign1 xor sign2;

    -- the operands are signed extended by one bit
    ext_a := unsigned(to_bitvector(a(20) & a(20 downto 0)));
    ext_b := unsigned(to_bitvector(b(20) & b(20 downto 0)));

    -- addition or subtraction is done based on the computed sign
    if sign = '1' then
       result := ext_a - ext_b ;
    else
       result := ext_a + ext_b ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;

end architecture behavioral;
```

A.2.14.7 23-bit Adder / Subtracter

There are two 23-bit adder / subtracter in the MAC design. They differ in the way that sign is determined. This is the VHDL code for 23-bit adder / subtracter of lower channel.

```
library ieee;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity adder_subtracter_23_low is
  port ( a : in std_logic_vector(21 downto 0);
         b : in std_logic_vector(23 downto 0);
```

```
        s : out std_logic_vector(23 downto 0);
        sign1 : in std_logic );
end entity adder_subtracter_23_low;


architecture behavioral of adder_subtracter_23_low is
begin

  behavior : process (a, b) is

    variable sign : std_logic;
    variable ext_a : unsigned(23 downto 0);
    variable ext_b : unsigned(23 downto 0);
    variable result : unsigned(23 downto 0);

  begin

    -- the sign is determined by input sign
    sign := sign1;

    -- this operand is signed extended by two bits
    ext_a := unsigned(to_bitvector(a(21) & a(21) & a(21 downto 0)));

    -- this operand does not need to be extended
    ext_b := unsigned(to_bitvector(b(23 downto 0)));

    -- addition or subtraction is done based on the sign
    if sign = '1' then
       result := ext_b - ext_a ;
    else
       result := ext_b + ext_a ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;

end architecture behavioral;
```

This VHDL file shows the 23-bit adder / subtracter of higher channel.

```
library ieee;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity adder_subtracter_23_high is
  port ( a : in std_logic_vector(21 downto 0);
         b : in std_logic_vector(23 downto 0);
```

```
        s : out std_logic_vector(23 downto 0);
        sign1 : in std_logic;
        num : in std_logic;
        sym : in std_logic );
end entity adder_subtracter_23_high;


architecture behavioral of adder_subtracter_23_high is
begin

  behavior : process (a, b) is

    variable sign : std_logic;
    variable ext_a : unsigned(23 downto 0);
    variable ext_b : unsigned(23 downto 0);
    variable result : unsigned(23 downto 0);

  begin

    -- the sign is determined by xoring of input sign and control
       signals
    sign := sign1 xor (num xor sym);

    -- this operand is signed extended by two bits
    ext_a := unsigned(to_bitvector(a(21) & a(21) & a(21 downto 0)));

    -- this operand does not need to be extended
    ext_b := unsigned(to_bitvector(b(23 downto 0)));

    -- addition or subtraction is done based on the computed sign
    if sign = '1' then
       result := ext_b - ext_a ;
    else
       result := ext_b + ext_a ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;

end architecture behavioral;
```

A.2.14.8 Accumulator Register

This register should be cleared whenever a MAC operation commences.

```
library ieee;
use ieee.std_logic_1164.all;

entity accumulator_regp is
  port ( clk : in std_logic;
         clr : in std_logic;
         d : in std_logic_vector(23 downto 0);
         q : out std_logic_vector(23 downto 0));
end entity accumulator_regp;

architecture behavioral of accumulator_regp is
begin

  -- when register is cleared the output is zero
  behavior : process (clk) is

  begin
    if rising_edge(clk) then
      if To_X01(clr) = '1' then
        q <= (others => '0') ;
      else
        q <= d ;
      end if;
    end if;
  end process behavior;

end architecture behavioral;
```

A.2.14.9 High Channel Register

This VHDL code shows a simple register behavior.

```
library ieee;
use ieee.std_logic_1164.all,
    work.tlns_types.all,
    work.tlns_instr.all,
    work.alu_types.all,
    work.numeric_bit.all;

entity channelhigh_reg is
  port ( clk : in std_logic;
         d : in std_logic_vector(23 downto 0);
         q : out std_logic_vector(23 downto 0));
end entity channelhigh_reg;
```

```
architecture behavioral of channelhigh_reg is
begin

  -- reads the input to output in the next clock edge
  behavior : process (clk) is

  begin
    if rising_edge(clk) then
        q <= d ;
    end if;
  end process behavior;

end architecture behavioral;
```

A.2.14.10 Channel Multiplexer

This multiplexer specifies the data which MAC unit places onto the destination bus of CPU.

```
library ieee;
use ieee.std_logic_1164.all;

use work.tlns_types.all,
    work.numeric_bit.all;

entity muxlowhigh is
  port ( i0 : in std_logic_vector(23 downto 0);
         i1 : in std_logic_vector(23 downto 0);
         y : out std_logic_vector(23 downto 0);
         sel : in std_logic);
end muxlowhigh;


architecture behavioral of muxlowhigh is
begin

  -- selects inputs based on sel signal
  -- and disables output when MAC unit is not in use
  with sel select
    y <=  i0  when '0',
          i1  when '1',
          disabled_tlns_word when others;
end architecture behavioral;
```

A.2.15 The Controller

The VHDL code for this unit, realizes a complicated state machine. This file is well documented to be self explanatory.

```
library ieee;

use ieee.std_logic_1164.all,
    ieee.std_logic_unsigned.all;

use work.tlns_types.all,
    work.tlns_instr.all,
    work.alu_types.all,
    work.numeric_bit.all;

entity controller is
  port ( clk : in std_logic;
         reset : in std_logic;
         halt : out std_logic;
         ir_mem_enable : out std_logic;
         ifetch : out std_logic;
         alu_function : out alu_func;
         alu_zero, alu_negative, alu_overflow : in bit;
         reg_s1_addr, reg_s2_addr, reg_dest_addr : out tlns_reg_addr;
         reg_write : out std_logic;
         a_enable : out std_logic;
         a_out_en : out std_logic;
         b_enable : out std_logic;
         b_out_en : out std_logic;
         pc_enable : out std_logic;
         pc_out_en : out std_logic;
         mar_enable : out std_logic;
         ir_immed1_size_18, ir_immed2_size_18 : out bit;
         ir_immed1_unsigned, ir_immed2_unsigned : out bit;
         ir_immed1_en, ir_immed2_en : out bit;
         current_instruction : in tlns_word;
         const2 : out tlns_bus_word;
         mem_write_en : out std_logic;
         mem_enable : out std_logic;
         mac_clr : out std_logic;
         alu_clr : out std_logic;
         btc_reset : out std_logic;
         btc_ready : in std_logic;
         btc_activate : out std_logic;
         in_reg_enable : out std_logic;
         in_reg_out_en : out std_logic;
         out_reg_enable : out std_logic;
```

```
        ctrl_mem_a : out tlns_bus_word;
        ctrl_ir_mem_a : out tlns_bus_word;
        ma_mux_sel : out bit;
        ir_ma_mux_sel : out bit;
        ctrl_direct : out bit;
        mac_ch_mux_sel : out std_logic;
        mac_coefnum : out std_logic;
        mac_evensym : out std_logic;
        s1_bus_content : in tlns_bus_word;
        s2_bus_content : in tlns_bus_word );

end entity controller;


architecture behavior of controller is

    type state_type is ( S1,S1_a,S2,S2_a,S3,S4,S4_a,S4_b,S5,S6,S7,S7_a,
                         S8,S8_a,S9,S9_a,S9_b,S9_c,S11,S11_a,S12,S12_a,
                         S12_b,S13,S13_a,S14,S14_a,S14_b,S15,S15_a,S15_b
                           ,
                         S16,S17,S17_a,S18,S18_a,S19,S19_a,S20,S20_a,S21
                           ,
                         S22,S23,S24,S25);
    signal state : state_type;

begin

  sequencer : process(clk, reset ) is

    alias IR_opcode : tlns_opcode is current_instruction(23 downto 18);
    alias IR_sp_func : tlns_sp_func is current_instruction(5 downto 0);
    alias IR_rs1 : tlns_reg_addr is current_instruction(17 downto 14);
    alias IR_rs2 : tlns_reg_addr is current_instruction(13 downto 10);
    alias IR_Itype_rd : tlns_reg_addr is current_instruction(13 downto
       10);
    alias IR_Rtype_rd : tlns_reg_addr is current_instruction(9 downto 6)
       ;
    alias IR_filter_order : unsigned(6 downto 0) is
                            current_instruction(6 downto 0);
    alias IR_filter_coef_sym : bit is current_instruction(7);
    alias IR_filter_bands_sym : unsigned(1 downto 0) is
                                current_instruction(9 downto 8);

    variable result_of_set_is_1 , branch_taken : boolean;
    variable filter_tap : unsigned(6 downto 0);
    variable filter_order : unsigned(6 downto 0);
    variable filter_coef_sym : bit;
    variable filter_bands_sym : unsigned(1 downto 0);
    variable first_half : bit;
```

```
variable coef_address : unsigned(23 downto 0);
variable coef_end_address : unsigned(23 downto 0);
variable top_coef_address : unsigned(6 downto 0);
variable coef_number : std_logic;
variable filt_data_address : std_logic_vector(23 downto 0);
variable filt_coef_address : std_logic_vector(23 downto 0);
variable filt_data_start : std_logic_vector(23 downto 0);
variable filt_data_end : std_logic_vector(23 downto 0);

-- fetches the instruction
procedure bus_instruction_fetch_1 is
begin
  -- address is determined by PC
  ir_ma_mux_sel <= '1';

  -- initialization
  b_out_en <= '0' ;
  mem_write_en <= '0' ;
  mem_enable <= '0' ;
  reg_write <= '0' ;
  ifetch <= '1' ;

  -- the instruction memory is enabled
  ir_mem_enable <= '1' ;
end procedure bus_instruction_fetch_1;

procedure bus_instruction_fetch_2 is
begin
  -- disables the instruction memory
  ir_mem_enable <= '0' ;
end procedure bus_instruction_fetch_2;

-- decodes the instruction
procedure instruction_decode_1( rs1,rs2 : tlns_reg_addr ;
                                opcode : tlns_opcode ) is
begin
  reg_s1_addr <= rs1 ;

  -- for 2DLNS/Binary conversion r14 must be read to register B
  if opcode = op_tbc then
     reg_s2_addr <= to_unsigned( unity_reg , 4 ) ;
  else
     reg_s2_addr <= rs2 ;
  end if;

  -- filter specifications are read to variables
  if opcode = op_filt then
     filter_order := IR_filter_order;
     filter_coef_sym := IR_filter_coef_sym;
```

```
      filter_bands_sym := IR_filter_bands_sym;
    end if;

    -- register file output registers are enabled
    a_enable <= '1' ;
    b_enable <= '1' ;

    -- PC is incremented
    pc_out_en <= '1' ;
    const2 <= X"000_001" ;
    alu_function <= alu_addu ;

    -- the new value is written to PC
    pc_enable <= '1' ;
  end procedure instruction_decode_1;

  -- disables A, B registers and PC
  procedure instruction_decode_2 is
  begin
    a_enable <= '0' ;
    b_enable <= '0' ;
    pc_out_en <= '0' ;
    const2 <= disabled_tlns_word ;
    pc_enable <= '0' ;
  end procedure instruction_decode_2;

  -- reads the registers which contain addresses for filtering
  procedure do_EX_filt_start_1 is
  begin
    a_out_en <= '1' ;
    b_out_en <= '1' ;
  end procedure do_EX_filt_start_1;

  -- prepares addresses for filtering
  procedure do_EX_filt_start_2 is
  begin
    -- disables registers
    a_out_en <= '0' ;
    b_out_en <= '0' ;

    -- writes addresses into variables
    filt_data_address := "00000000000000" & s1_bus_content(23 downto
      14);
    filt_coef_address := "00000000000000" & s1_bus_content(9 downto 0)
      ;
    filt_data_start := "00000000000000" & s2_bus_content(9 downto 0);
    filt_data_end := "00000000000000" & s2_bus_content(23 downto 14);

    -- enables both memories
```

```
    ir_mem_enable <= '1' ;
    mem_enable <= '1' ;

    -- memory addresses are determined by the controller
    ma_mux_sel <= '0' ;
    ir_ma_mux_sel <= '0' ;

    -- determines the coefficients ' address range
    coef_address := unsigned(to_bitvector(filt_coef_address));
    top_coef_address := filter_order srl 1 ;
    coef_end_address := coef_address + top_coef_address;

    filter_tap := B"0000000";
    first_half := '1';

    -- specifies the coefficient symmetry signal for the MAC unit
    if filter_bands_sym = "01" then
       coef_number := '1' ;
    else
       coef_number := '0' ;
    end if;
    mac_coefnum <= coef_number;

    -- addresses memories
    ctrl_mem_a <= filt_data_address;
    ctrl_ir_mem_a <= to_x01(bit_vector(coef_address));
  end procedure do_EX_filt_start_2;

  -- performs MAC operations
  procedure do_EX_filt_mac is
  begin
    -- directs coefficients to the data bus
    ir_immed2_en <= '1' ;
    ctrl_direct <= '1';

    -- one MAC operation is performed
    mac_clr <= '0' ;
    alu_clr <= '1' ;

    filter_tap := filter_tap + 1 ;

    -- toggles coefficient symmetry signal
    coef_number := not coef_number ;
    mac_coefnum <= coef_number ;

    -- specifies the next coefficient address
    if filter_coef_sym = '0' then
      coef_address := coef_address + 1;
    else
```

```
      if coef_address < coef_end_address then
        if first_half = '1' then
          coef_address := coef_address + 1;
        else
          coef_address := coef_address - 1;
        end if;
      else
        coef_address := coef_address - 1;
        first_half := '0';
      end if;
    end if;

    -- determines the next data address
    filt_data_address := filt_data_address - 1 ;
    if filt_data_address = filt_data_start - 1 then
       filt_data_address := filt_data_end;
    end if;

    -- addresses memories
    ctrl_mem_a <= filt_data_address;
    ctrl_ir_mem_a <= to_x01(bit_vector(coef_address));
  end procedure do_EX_filt_mac;

  -- performs the last iteration of MAC operation
  procedure do_EX_filt_last is
  begin
    -- output multiplexer is set and destination register is written
    mac_ch_mux_sel <= '0' ;
    reg_dest_addr <= to_unsigned(output_reg_1 , 4) ;
    reg_write <= '1' ;

    -- memories are disabled
    ir_mem_enable <= '0' ;
    mem_enable <= '0' ;
    ctrl_mem_a <= disabled_tlns_word ;
    ctrl_ir_mem_a <= disabled_tlns_word ;
  end procedure do_EX_filt_last;

  -- all control signals are reset to their defaults
  procedure do_EX_filt_out is
  begin
    mac_coefnum <= '0';
    ir_immed2_en <= '0' ;
    ctrl_direct <= '0';
    reg_write <= '0' ;
    mac_ch_mux_sel <= '0' ;
    mac_clr <= '1' ;
    alu_clr <= '0' ;
    ir_ma_mux_sel <= '1' ;
```

```
    ma_mux_sel <= '1' ;
  end procedure do_EX_filt_out;

  -- performs a subtraction for unsigned relational instructions
  procedure do_EX_set_unsigned_1 ( immed : boolean ) is
  begin
    a_out_en <= '1' ;
    if immed then
      ir_immed2_size_18 <= '0' ;
      ir_immed2_unsigned <= '1' ;
      ir_immed2_en <= '1' ;
    else
      b_out_en <= '1' ;
    end if;
    alu_function <= alu_subu ;
  end procedure do_EX_set_unsigned_1;

  -- sets the result based on status flags
  procedure do_EX_set_unsigned_2 ( Rd : tlns_reg_addr ;
                                   immed : boolean ) is
  begin
    a_out_en <= '0' ;
    if immed then
      ir_immed2_en <= '0' ;
    else
      b_out_en <= '0' ;
    end if;

    if immed then
      case IR_opcode is
        when op_sequi =>
          result_of_set_is_1 := alu_zero = '1';
        when op_sneui =>
          result_of_set_is_1 := alu_zero /= '1';
        when op_sltui =>
          result_of_set_is_1 := alu_overflow = '1';
        when op_sgtui =>
          result_of_set_is_1 := alu_overflow /= '1' and alu_zero /=
              '1';
        when op_sleui =>
          result_of_set_is_1 := alu_overflow = '1' or alu_zero = '1';
        when op_sgeui =>
          result_of_set_is_1 := alu_overflow /= '1';
        when others =>
          null;
      end case;
    else
      case IR_sp_func is
        when sp_func_sequ =>
```

```
        result_of_set_is_1 := alu_zero = '1';
      when sp_func_sneu =>
        result_of_set_is_1 := alu_zero /= '1';
      when sp_func_sltu =>
        result_of_set_is_1 := alu_overflow = '1';
      when sp_func_sgtu =>
        result_of_set_is_1 := alu_overflow /= '1' and alu_zero /=
            '1';
      when sp_func_sleu =>
        result_of_set_is_1 := alu_overflow = '1' or alu_zero = '1';
      when sp_func_sgeu =>
        result_of_set_is_1 := alu_overflow /= '1';
      when others =>
        null;
    end case;
  end if;

  -- do_set_result;
  if result_of_set_is_1 then
    const2 <= X"000_001" ;
  else
    const2 <= X"000_000" ;
  end if;
  alu_function <= alu_pass_s2 ;
  reg_dest_addr <= Rd ;
  reg_write <= '1' ;
end procedure do_EX_set_unsigned_2;

-- resets signals
procedure do_EX_set_unsigned_3 is
begin
  const2 <= disabled_tlns_word ;
  reg_write <= '0' ;
end procedure do_EX_set_unsigned_3;

-- performs a subtraction for signed relational instructions
procedure do_EX_set_signed_1 ( immed : boolean ) is
begin
  a_out_en <= '1' ;
  if immed then
    ir_immed2_size_18 <= '0' ;
    ir_immed2_unsigned <= '0' ;
    ir_immed2_en <= '1' ;
  else
    b_out_en <= '1' ;
  end if;
  alu_function <= alu_sub ;
end procedure do_EX_set_signed_1;
```

```
-- sets the result based on status flags
procedure do_EX_set_signed_2 ( Rd : tlns_reg_addr ;
                               immed : boolean ) is
begin
  a_out_en <= '0' ;
  if immed then
    ir_immed2_en <= '0' ;
  else
    b_out_en <= '0' ;
  end if;

  if immed then
    case IR_opcode is
      when op_seqi =>
        result_of_set_is_1 := alu_zero = '1';
      when op_snei =>
        result_of_set_is_1 := alu_zero /= '1';
      when op_slti =>
        result_of_set_is_1 := alu_negative = '1';
      when op_sgti =>
        result_of_set_is_1 := alu_negative /= '1' and alu_zero /=
            '1';
      when op_slei =>
        result_of_set_is_1 := alu_negative = '1' or alu_zero = '1';
      when op_sgei =>
        result_of_set_is_1 := alu_negative /= '1';
      when others =>
        null;
    end case;
  else
    case IR_sp_func is
      when sp_func_seq =>
        result_of_set_is_1 := alu_zero = '1';
      when sp_func_sne =>
        result_of_set_is_1 := alu_zero /= '1';
      when sp_func_slt =>
        result_of_set_is_1 := alu_negative = '1';
      when sp_func_sgt =>
        result_of_set_is_1 := alu_negative /= '1' and alu_zero /=
            '1';
      when sp_func_sle =>
        result_of_set_is_1 := alu_negative = '1' or alu_zero = '1';
      when sp_func_sge =>
        result_of_set_is_1 := alu_negative /= '1';
      when others =>
        null;
    end case;
  end if;
```

```
  -- do_set_result;
  if result_of_set_is_1 then
    const2 <= X"000_001" ;
  else
    const2 <= X"000_000" ;
  end if;
  alu_function <= alu_pass_s2 ;
  reg_dest_addr <= Rd ;
  reg_write <= '1' ;
end procedure do_EX_set_signed_2;

-- resets signals
procedure do_EX_set_signed_3 is
begin
  const2 <= disabled_tlns_word ;
  reg_write <= '0' ;
end procedure do_EX_set_signed_3;

-- performs MAC operation
procedure do_EX_mac_1( Rd : tlns_reg_addr ) is
begin
  -- reads operands from registers
  a_out_en <= '1' ;
  b_out_en <= '1' ;

  -- activates the MAC unit
  mac_clr <= '0' ;
  alu_clr <= '1' ;

  -- enables destination register
  reg_dest_addr <= Rd ;
  reg_write <= '1' ;
end procedure do_EX_mac_1;

-- clears all signals to their defaults
procedure do_EX_mac_2 is
begin
  a_out_en <= '0' ;
  b_out_en <= '0' ;
  reg_write <= '0' ;
  mac_clr <= '1' ;
  alu_clr <= '0' ;
end procedure do_EX_mac_2;

-- performs binary / 2DLNS conversion
procedure do_EX_btconvert_1 is
begin
  -- reads binary data from register
  a_out_en <= '1' ;
```

```
  -- activates converter
  alu_clr <= '1' ;
  btc_reset <= '0' ;
  btc_activate <= '1';
end procedure do_EX_btconvert_1;

-- prepares destination register
procedure do_EX_btconvert_2( Rd : tlns_reg_addr ) is
begin
  a_out_en <= '0' ;
  reg_dest_addr <= Rd ;
  reg_write <= '1' ;
  btc_activate <= '0';
end procedure do_EX_btconvert_2;

-- disables register file
procedure do_EX_btconvert_3 is
begin
  reg_write <= '0' ;
end procedure do_EX_btconvert_3;

-- resets control signals
procedure do_EX_btconvert_4 is
begin
  btc_reset <= '1' ;
  alu_clr <= '0' ;
end procedure do_EX_btconvert_4;

-- reads external data into register file
procedure do_EX_input_1( Rd : tlns_reg_addr ) is
begin
  -- input register is enabled to receive data
  in_reg_enable <= '1' ;

  -- data is directed through the ALU to register file
  in_reg_out_en <= '1' ;
  alu_function <= alu_pass_s1 ;
  reg_dest_addr <= Rd ;
  reg_write <= '1' ;
end procedure do_EX_input_1;

-- resets control signals
procedure do_EX_input_2 is
begin
  in_reg_enable <= '0' ;
  in_reg_out_en <= '0' ;
  reg_write <= '0' ;
end procedure do_EX_input_2;
```

```
-- reads a register content to the output register
procedure do_EX_output_1 is
begin
  b_out_en <= '1' ;
  out_reg_enable <= '1' ;
end procedure do_EX_output_1;

-- control signals are reset
procedure do_EX_output_2 is
begin
  b_out_en <= '0' ;
  out_reg_enable <= '0' ;
end procedure do_EX_output_2;

-- executes arithmetic and logic operations on register contents
procedure do_EX_arith_logic_1( Rd : tlns_reg_addr ) is
begin
  -- reads operands from registers
  a_out_en <= '1' ;
  b_out_en <= '1' ;

  -- specifies ALU function
  case IR_sp_func is
    when sp_func_add =>
      alu_function <= alu_add ;
    when sp_func_addu =>
      alu_function <= alu_addu ;
    when sp_func_sub =>
      alu_function <= alu_sub ;
    when sp_func_subu =>
      alu_function <= alu_subu ;
    when sp_func_and =>
      alu_function <= alu_and ;
    when sp_func_or =>
      alu_function <= alu_or ;
    when sp_func_xor =>
      alu_function <= alu_xor ;
    when sp_func_sll =>
      alu_function <= alu_sll ;
    when sp_func_srl =>
      alu_function <= alu_srl ;
    when sp_func_sra =>
      alu_function <= alu_sra ;
    when others =>
      null;
  end case;

  -- determines destination register
```

```
    reg_dest_addr <= Rd ;
    reg_write <= '1' ;
  end procedure do_EX_arith_logic_1;

  -- resets control signals
  procedure do_EX_arith_logic_2 is
  begin
    a_out_en <= '0' ;
    b_out_en <= '0' ;
    reg_write <= '0' ;
  end procedure do_EX_arith_logic_2;

  -- executes arithmetic and logic operations on an immediate value
  procedure do_EX_arith_logic_immed_1( Rd : tlns_reg_addr ) is
  begin
    -- reads one operand from register
    a_out_en <= '1' ;

    -- just for addition and subtraction, extension is signed
    if IR_opcode = op_addi or IR_opcode = op_subi
    then
      ir_immed2_unsigned <= '0' ;
    else
      ir_immed2_unsigned <= '1' ;
    end if;

    -- extends 10-bit immediate value
    ir_immed2_size_18 <= '0' ;
    ir_immed2_en <= '1' ;

    -- specifies ALU function
    case IR_opcode is
      when op_addi =>
        alu_function <= alu_add ;
      when op_subi =>
        alu_function <= alu_sub ;
      when op_addui =>
        alu_function <= alu_addu ;
      when op_subui =>
        alu_function <= alu_subu ;
      when op_andi =>
        alu_function <= alu_and ;
      when op_ori =>
        alu_function <= alu_or ;
      when op_xori =>
        alu_function <= alu_xor ;
      when op_slli =>
        alu_function <= alu_sll ;
      when op_srli =>
```

```
        alu_function <= alu_srl ;
      when op_srai =>
        alu_function <= alu_sra ;
      when others =>
        null;
    end case;

    -- determines destination register
    reg_dest_addr <= Rd ;
    reg_write <= '1' ;
  end procedure do_EX_arith_logic_immed_1;

  -- resets control signals
  procedure do_EX_arith_logic_immed_2 is
  begin
    a_out_en <= '0' ;
    ir_immed2_en <= '0' ;
    reg_write <= '0' ;
  end procedure do_EX_arith_logic_immed_2;

  -- writes the PC content to r15
  procedure do_EX_link_1 ( Rd : tlns_reg_addr ) is
  begin
    pc_out_en <= '1' ;
    alu_function <= alu_pass_s1 ;
    reg_dest_addr <= Rd ;
    reg_write <= '1' ;
  end procedure do_EX_link_1;

  -- clears signals to their defaults
  procedure do_EX_link_2 is
  begin
    pc_out_en <= '0' ;
    reg_write <= '0' ;
  end procedure do_EX_link_2;

  -- loads a 10-bit immediate value to a register
  procedure do_EX_lhi_1( Rd : tlns_reg_addr ) is
  begin
    -- the immediate value is unsigned extended
    ir_immed1_size_18 <= '0' ;
    ir_immed1_unsigned <= '1' ;
    ir_immed1_en <= '1' ;

    -- it is shifted to left by 14 bits
    const2 <= X"000_00E" ;
    alu_function <= alu_sll ;

    -- shifted value is written to destination register
```

```
    reg_dest_addr <= Rd ;
    reg_write <= '1' ;
  end procedure do_EX_lhi_1;

  -- disables control signals
  procedure do_EX_lhi_2 is
  begin
    ir_immed1_en <= '0' ;
    const2 <= disabled_tlns_word ;
    reg_write <= '0' ;
  end procedure do_EX_lhi_2;

  -- reads a register content to the ALU
  procedure do_EX_branch_1 is
  begin
    a_out_en <= '1' ;
    alu_function <= alu_pass_s1 ;
  end procedure do_EX_branch_1;

  -- branch is taken based on zero flag
  procedure do_EX_branch_2 is
  begin
    a_out_en <= '0' ;
    if IR_opcode = op_beqz then
      branch_taken := alu_zero = '1';
    else
      branch_taken := alu_zero /= '1';
    end if;
  end procedure do_EX_branch_2;

  -- specifies memory address for load and store instructions
  procedure do_EX_load_store_1 is
  begin
    -- reads the content of source register
    a_out_en <= '1' ;

    -- the immediate value is sign extended
    ir_immed2_size_18 <= '0' ;
    ir_immed2_unsigned <= '0' ;
    ir_immed2_en <= '1' ;

    -- the register content and extended value are added
    alu_function <= alu_add ;

    -- memory address register receives the address
    mar_enable <= '1' ;
    ma_mux_sel <= '1' ;
  end procedure do_EX_load_store_1;
```

```
-- control signals are reset
procedure do_EX_load_store_2 is
begin
  a_out_en <= '0' ;
  ir_immed2_en <= '0' ;
  mar_enable <= '0' ;
end procedure do_EX_load_store_2;

-- specifies a new address for PC
procedure do_MEM_jump_1 is
begin
  -- reads the PC content
  pc_out_en <= '1'  ;

  -- the 18-bit immediate value is sign extended
  ir_immed2_size_18 <= '1' ;
  ir_immed2_unsigned <= '0' ;
  ir_immed2_en <= '1' ;

  -- the PC content and immediate value are added
  alu_function <= alu_add ;

  -- the new address is written to PC
  pc_enable <= '1' ;
end procedure do_MEM_jump_1;

-- control signals are reset to their defaults
procedure do_MEM_jump_2 is
begin
  pc_out_en <= '0' ;
  ir_immed2_en <= '0' ;
  pc_enable <= '0' ;
end procedure do_MEM_jump_2;

-- the new address for PC is a register content
procedure do_MEM_jump_reg_1 is
begin
  a_out_en <= '1' ;
  alu_function <= alu_pass_s1 ;
  pc_enable <= '1' ;
end procedure do_MEM_jump_reg_1;

-- resets signals
procedure do_MEM_jump_reg_2 is
begin
  a_out_en <= '0' ;
  pc_enable <= '0' ;
end procedure do_MEM_jump_reg_2;
```

```
-- specifies a new address for PC
procedure do_MEM_branch_1 is
begin
  -- reads the PC content
  pc_out_en <= '1' ;

  -- the 10-bit immediate value is sign extended
  ir_immed2_size_18 <= '0' ;
  ir_immed2_unsigned <= '0' ;
  ir_immed2_en <= '1' ;

  -- the PC content and immediate value are added
  alu_function <= alu_add ;

  -- the new address is written to PC
  pc_enable <= '1' ;
end procedure do_MEM_branch_1;

-- control signals are reset to their defaults
procedure do_MEM_branch_2 is
begin
  pc_out_en <= '0' ;
  ir_immed2_en <= '0' ;
  pc_enable <= '0' ;
end procedure do_MEM_branch_2;

-- loads a register with a data which is read from memory
procedure do_MEM_load_1 is
begin
  ifetch <= '0' ;
  mem_write_en <= '0' ;

  -- memory is enabled
  mem_enable <= '1' ;

  -- the ALU directs the read data to the destination bus
  alu_function <= alu_pass_s1 ;
end procedure do_MEM_load_1;

-- destination register is written
procedure do_MEM_load_2( Rd : tlns_reg_addr ) is
begin
  mem_enable <= '0' ;
  reg_dest_addr <= Rd ;
  reg_write <= '1' ;
end procedure do_MEM_load_2;

-- stores a register content into the data memory
procedure do_MEM_store_1 is
```

```
    begin
      -- reads content of B register
      b_out_en <= '1' ;

      -- the ALU directs data to the destination bus
      alu_function <= alu_pass_s2 ;

      ifetch <= '0' ;

      -- memory is enabled to be written
      mem_write_en <= '1' ;
      mem_enable <= '1' ;
    end procedure do_MEM_store_1;

  begin -- sequencer
    if reset = '1' then

    ------------------------------------------------------------

    -- initialize all control signals

    ------------------------------------------------------------

    halt <= '0' ;
    ir_mem_enable <= '0' ;
    ifetch <= '0' ;
    alu_function <= alu_add ;
    reg_s1_addr <= B"0000" ;
    reg_s2_addr <= B"0000" ;
    reg_dest_addr <= B"0000" ;
    reg_write <= '0' ;
    a_enable <= '0' ;
    a_out_en <= '0' ;
    b_enable <= '0' ;
    b_out_en <= '0' ;
    pc_enable <= '0' ;
    pc_out_en <= '0' ;
    mar_enable <= '0' ;
    ir_immed1_size_18 <= '0' ;
    ir_immed2_size_18 <= '0' ;
    ir_immed1_unsigned <= '0' ;
    ir_immed2_unsigned <= '0' ;
    ir_immed1_en <= '0' ;
    ir_immed2_en <= '0' ;
    const2 <= disabled_tlns_word ;
    mem_write_en <= '0' ;
    mem_enable <= '0' ;
    mac_clr <= '1';
    alu_clr <= '0';
    btc_reset <= '1';
    btc_activate <= '0';
```

```
in_reg_enable <= '0';
in_reg_out_en <= '0';
out_reg_enable <= '0';
ma_mux_sel <= '1';
ir_ma_mux_sel <= '1';
ctrl_mem_a <= disabled_tlns_word ;
ctrl_ir_mem_a <= disabled_tlns_word ;
ctrl_direct <= '0';
mac_ch_mux_sel <= '0';
mac_coefnum <= '0';
mac_evensym <= '0';
state <= s1 ;

elsif rising_edge(clk) then

  ------------------------------------------------------------
  -- control loop
  ------------------------------------------------------------

  case state is

    --
      ------------------------------------------------------------

    --      fetch next instruction (IF)
    --
      ------------------------------------------------------------


    when s1     => bus_instruction_fetch_1 ;
                   state <= s1_a ;

    when s1_a   => bus_instruction_fetch_2 ;
                   state <= s2 ;

    --
      ------------------------------------------------------------

    -- instruction decode, source register read and PC increment (
       ID)
    --
      ------------------------------------------------------------


    when s2     => instruction_decode_1(IR_rs1, IR_rs2, IR_opcode)
       ;
                   state <= s2_a ;

    when s2_a   => instruction_decode_2 ;
```

```
case IR_opcode is
    when op_special =>
        case IR_sp_func is

            when sp_func_nop =>
                null ;
                state <= s1 ;

            when sp_func_mult =>
                do_EX_mac_1(IR_Rtype_rd);
                state <= s3 ;

            when sp_func_halt =>
                state <= s24 ;

            when sp_func_add | sp_func_addu
                | sp_func_sub | sp_func_subu
                | sp_func_sll | sp_func_srl
                | sp_func_sra | sp_func_and
                | sp_func_or  | sp_func_xor =>
                do_EX_arith_logic_1(
                    IR_Rtype_rd);
                state <= s6 ;

            when sp_func_sequ | sp_func_sneu
                | sp_func_sltu | sp_func_sgtu
                | sp_func_sleu | sp_func_sgeu
                    =>
                do_EX_set_unsigned_1(immed =>
                    false);
                state <= s7 ;

            when sp_func_seq | sp_func_sne
                | sp_func_slt | sp_func_sgt
                | sp_func_sle | sp_func_sge =>
                do_EX_set_signed_1(immed =>
                    false);
                state <= s8 ;

            when others =>
                null;
                state <= s1 ;

        end case;

    when op_btc =>
        do_EX_btconvert_1;
        state <= s4 ;
```

```
when op_tbc =>
     do_EX_mac_1(IR_Itype_rd);
     state <= s5 ;

when op_filt =>
     -- the type of symmetry is specified
     if filter_bands_sym = "01" then
        mac_evensym <= '1' ;
     elsif filter_bands_sym = "10" then
        mac_evensym <= '0' ;
     end if;

     do_EX_filt_start_1;
     state <= s9 ;

when op_j =>
     state <= s11 ;

when op_jal =>
     do_EX_link_1(to_unsigned(link_reg, 4));
     state <= s12 ;

when op_jr =>
     state <= s13 ;

when op_jalr =>
     do_EX_link_1(to_unsigned(link_reg, 4));
     state <= s14 ;

when op_beqz | op_bnez =>
     do_EX_branch_1;
     state <= s15 ;

when op_addi | op_subi | op_addui | op_subui
     | op_slli | op_srli | op_srai
     | op_andi | op_ori | op_xori =>
     do_EX_arith_logic_immed_1(IR_Itype_rd);
     state <= s16 ;

when op_lhi =>
     state <= s17 ;

when op_sequi | op_sneui | op_sltui
     | op_sgtui | op_sleui | op_sgeui =>
     do_EX_set_unsigned_1(immed => true);
     state <= s18 ;

when op_seqi | op_snei | op_slti
```

```
                    | op_sgti | op_slei | op_sgei =>
                    do_EX_set_signed_1(immed => true);
                    state <= s19 ;

                when op_lw  =>
                    do_EX_load_store_1;
                    state <= s20 ;

                when op_sw =>
                    do_EX_load_store_1;
                    state <= s21 ;

                when op_inpt =>
                    do_EX_input_1(IR_Itype_rd);
                    state <= s22 ;

                when op_oupt =>
                    do_EX_output_1;
                    state <= s23 ;

                when others =>
                    null;
                    state <= s1 ;

            end case;

--
-- ------------------------------------------------------------

-- execute instruction, (EX, MEM, WB)
--
-- ------------------------------------------------------------

when s3     => do_EX_mac_2;
               state <= s1 ;

when s4     => do_EX_btconvert_2(IR_Itype_rd);
               state <= s4_a;

when s4_a   => -- remains here while conversion is in process
               if btc_ready = '0' then
                  state <= s4_a;
               else
                  do_EX_btconvert_3;
                  state <= s4_b;
               end if;

when s4_b   => do_EX_btconvert_4;
```

```
                    state <= s1;

when s5         => do_EX_mac_2 ;
                   state <= s1 ;

when s6         => do_EX_arith_logic_2;
                   state <= s1 ;

when s7         => do_EX_set_unsigned_2(IR_Rtype_rd, false);
                   state <= s7_a ;

when s7_a       => do_EX_set_unsigned_3;
                   state <= s1 ;

when s8         => do_EX_set_signed_2(IR_Rtype_rd, false);
                   state <= s8_a ;

when s8_a       => do_EX_set_signed_3;
                   state <= s1 ;

when s9         => do_EX_filt_start_2;
                   state <= s9_a ;

when s9_a       => do_EX_filt_mac;
                   -- MAC operation continues for all
                       coefficients
                   if filter_tap < filter_order then
                      state <= s9_a ;
                   else
                      do_EX_filt_last;
                      state <= s9_b ;
                   end if;

when s9_b       => -- checks if filters are symmetric
                   if filter_bands_sym = "00" or
                      filter_bands_sym = "11" then
                      do_Ex_filt_out;
                      state <= s1 ;
                   else
                      -- sets signals to write the dual filter
                          output
                      mac_ch_mux_sel <= '1' ;
                      reg_dest_addr <= to_unsigned(output_reg_2,
                          4) ;
                      state <= s9_c ;
                   end if;

when s9_c       =>  do_Ex_filt_out;
                    state <= s1 ;
```

```
when s11      => do_MEM_jump_1;
                 state <= s11_a ;

when s11_a    => do_MEM_jump_2;
                 state <= s1 ;

when s12      => do_EX_link_2;
                 state <= s12_a ;

when s12_a    => do_MEM_jump_1;
                 state <= s12_b ;

when s12_b    => do_MEM_jump_2;
                 state <= s1 ;

when s13      => do_MEM_jump_reg_1;
                 state <= s13_a ;

when s13_a    => do_MEM_jump_reg_2;
                 state <= s1 ;

when s14      => do_EX_link_2;
                 state <= s14_a ;

when s14_a    => do_MEM_jump_reg_1;
                 state <= s14_b ;

when s14_b    => do_MEM_jump_reg_2;
                 state <= s1 ;

when s15      => do_EX_branch_2;
                 -- determines the next state in a branch
                    instruction
                 if branch_taken then
                    state <= s15_a ;
                 else
                    state <= s1 ;
                 end if;

when s15_a    => do_MEM_branch_1;
                 state <= s15_b ;

when s15_b    => do_MEM_branch_2;
                 state <= s1 ;

when s16      => do_EX_arith_logic_immed_2;
                 state <= s1 ;
```

```
        when s17      => do_EX_lhi_1(IR_Itype_rd);
                         state <= s17_a ;

        when s17_a    => do_EX_lhi_2;
                         state <= s1 ;

        when s18      => do_EX_set_unsigned_2(IR_Itype_rd , immed =>
           true);
                         state <= s18_a ;

        when s18_a    => do_EX_set_unsigned_3;
                         state <= s1 ;

        when s19      => do_EX_set_signed_2(IR_Itype_rd , immed => true)
           ;
                         state <= s19_a ;

        when s19_a    => do_EX_set_signed_3;
                         state <= s1 ;

        when s20      => do_EX_load_store_2;
                         do_MEM_load_1;
                         state <= s20_a ;

        when s20_a    => do_MEM_load_2(IR_Itype_rd);
                         state <= s1 ;

        when s21      => do_EX_load_store_2;
                         do_MEM_store_1;
                         state <= s1 ;

        when s22      => do_EX_input_2;
                         state <= s1 ;

        when s23      => do_EX_output_2;
                         state <= s1 ;

        when s24      => null;
                         state <= s1 ;

        when s25      => halt <= '1' ;
                         state <= s25 ;

        when others   => null;
                         state <= s1 ;
    end case;

end if;
```

```
  end process sequencer;

end architecture behavior;
```

A.3 TLNS CPU Test

This section contains the VHDL files of CPU test bench module and its components. These files also include some structures which are added to the CPU test bench for filterbank application.

A.3.1 The TLNS CPU Test Bench

This code is the top module of test bench model. The test bench includes instances of the TLNS CPU, the instruction and data memories, the clock generator, and the input data reader.

```
library ieee;
use ieee.std_logic_1164.all,
    ieee.std_logic_textio.all;

library work;
use work.tlns_types.all,
    work.numeric_bit.all;

use std.textio.all;

entity tlns_test is

end entity tlns_test;


architecture bench of tlns_test is

  -- The Input Data Reader
  component input_gen is
    port ( clk : in std_logic;
           data_in : out external_data;
           out_en : in std_logic );
  end component input_gen;
```

```
-- The Clock Generator
component clock_gen is
  generic ( Tclk : delay_length := 20 ns );
  port ( clk : out std_logic;
         reset : out std_logic );
end component clock_gen;

-- The Instruction Memory
component memory is
  generic ( mem_size : positive := 1024;
            Tac_first : delay_length := 70 ns;
            Tpd_clk_out : delay_length := 2 ns );
  port ( clk : in std_logic;
           a : in tlns_bus_word;
           d : out tlns_bus_word;
           ir_mem_enable : in std_logic );
end component memory;

-- The Data Memory
component data_memory is
    generic ( data_memory_size : positive := 1024);
    port ( clk : in std_logic;
         mem_a : in mem_bus_addr;
         mem_d_in : in tlns_bus_word;
         mem_d_out : out tlns_bus_word;
         mem_write_en : in std_logic;
         mem_enable : in std_logic );
end component data_memory;

-- The TLNS CPU
component tlns is
  port ( clk : in std_logic;
         reset : in std_logic;
         halt : out std_logic;
         input_data : in external_data;
         output_data : out tlns_bus_word;
         a : out tlns_bus_word;
         d : in tlns_bus_word;
         ifetch : out std_logic;
         ir_mem_enable : out std_logic;
         mem_a : out mem_bus_addr;
         mem_d_out : in tlns_bus_word;
         mem_d_in : out tlns_bus_word;
         mem_write_en : out std_logic;
         output_enable : out std_logic;
         mem_enable : out std_logic );
end component tlns;
```

```
  signal clk : std_logic;
  signal reset : std_logic;
  signal a : tlns_bus_word;
  signal d : tlns_bus_word;
  signal halt : std_logic;
  signal ir_mem_enable, ifetch : std_logic;
  signal mem_a : mem_bus_addr;
  signal mem_d_in : tlns_bus_word;
  signal mem_d_out : tlns_bus_word;
  signal mem_enable, mem_write_en : std_logic;
  signal data_in : external_data;
  signal data_out : tlns_bus_word;
  signal out_en : std_logic;
  signal sync : std_logic;

begin

  -- The component instantiations

  input : component input_gen
     port map (  clk => clk, data_in => data_in, out_en => sync );

  cg : component clock_gen
     port map ( clk => clk, reset => reset );

  mem : component memory
    port map ( clk => clk,
               a => a, d => d,
               ir_mem_enable => ir_mem_enable );

  data_mem : component data_memory
    port map ( clk => clk,
               mem_a => mem_a,
               mem_d_in => mem_d_in,
               mem_d_out => mem_d_out,
               mem_write_en => mem_write_en,
               mem_enable => mem_enable );

  proc : component tlns
    port map ( clk => clk,
               reset => reset,
               halt => halt,
               input_data => data_in,
               output_data => data_out,
               a => a,
               d => d,
               ifetch => ifetch,
               ir_mem_enable => ir_mem_enable,
               mem_a => mem_a,
```

```
                mem_d_in => mem_d_in,
                mem_d_out => mem_d_out,
                mem_write_en => mem_write_en,
                output_enable => out_en,
                mem_enable => mem_enable );

  -- this process writes the output into a file
  write_output: process( clk)

    file output_file : text open write_mode is "filter_output";
    variable line_out : line;

  begin
   if rising_edge(clk) then
     -- writes the output when output is enabled
     if out_en = '1' then
        write(line_out, data_out(3 downto 0));
        write(line_out, string'(" "));
        write(line_out, data_out(23 downto 4));
        writeline(output_file, line_out);

        -- when a data is written to output, the next input data is read
        if data_out(3 downto 0) = "0000" then
          sync <= '1';
        else
          sync <= '0';
        end if;
      else
        sync <= '0';
      end if;

    end if;

  end process write_output;

end architecture bench;
```

A.3.2 The Test Bench Clock Generator

The VHDL code for this module, generates clock and reset signals for the CPU.

```
library ieee;
use ieee.std_logic_1164.all;

entity clock_gen is
  generic ( Tclk : delay_length );
```

```
  port ( clk : out std_logic;
         reset : out std_logic );
end entity clock_gen;


architecture behavior of clock_gen is
begin

  -- sets reset port of the CPU
  reset_driver :
    reset <= '1', '0' after 3.5 * Tclk;

  -- generates the clock signal
  clock_driver : process is
  begin
    clk <= '0';
    wait for Tclk;
    loop
      clk <= '1', '0' after Tclk / 2;
      wait for Tclk;
    end loop;
  end process clock_driver;

end architecture behavior;
```

A.3.3 The Test Bench Instruction Memory

This file shows the instruction memory for the TLNS CPU which is preloaded with filterbank program. The coefficients of all filters are also stored in this memory.

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.tlns_types.all,
    work.numeric_bit.all;

entity memory is
  generic ( mem_size : positive := 1024;
            Tac_first : delay_length;
            Tpd_clk_out : delay_length);
  port ( clk : in std_logic;
         a : in tlns_bus_word;
         d : out tlns_bus_word;
         ir_mem_enable : in std_logic);
```

```
end entity memory;


architecture preloaded of memory is
begin

  mem_behavior : process is

    constant high_address : natural := mem_size - 1;

    type memory_array is array ( natural range <> ) of tlns_word;

    variable mem : memory_array(0 to high_address)
      := ( others => X"000000" );

    variable byte_address, word_address : natural;
    variable write_access : boolean;

    -- The TLNS program and filter coefficients are loaded into memory
    procedure load is

      constant program : memory_array
        := ( X"8C3801",    --0          lw      r14,M[1]
                           --                   Writting addresses to
                                registers
             X"201404",    --1          addi    r0, r5, dstart
             X"2429FF",    --2          addi    r0, r10, dend
             X"3C11FF",    --3          lhi     r0, r4, dend
             X"0114E5",    --4          or      r4 ,r5, r3
             X"20044E",    --5          addi    r0, r1, data_address
             X"200840",    --6    next  addi    r0, r2, coef_address
             X"50600E",    --7          slli    r1, r8, E
             X"0208A5",    --8          or      r8, r2, r2
                           --                   Entering Data
             X"401800",    --9          inpt    r0, r6, 0
                           --                   Converting Data
             X"199C00",    --a          btc     r6, r7
                           --                   Storing Data
             X"AC5C00",    --b          sw      M[r1], r7
                           --                   Filters 0,7
             X"548DCB",    --c          filter  r2, r3, coef sym, even,
                75
                           --                   Writing output
             X"202400",    --d          addi    r0, r9, band_tag
             X"533004",    --e          slli    r12, r12, 4
             X"027325",    --f          or      r9, r12, r12
             X"443000",    --10         oupt    r0, r12, 0
             X"202C07",    --11         addi    r0, r11, dual_band_tag
             X"537404",    --12         slli    r13, r13, 4
```

```
X"02F765",   --13    or      r11, r13, r13
X"443400",   --14    oupt    r0, r13, 0
             --              Next set of Coefficients
X"208826",   --15    addi    r2, r2, next_coef_address
             --              Filters 1,6
X"548DCB",   --16    filter  r2, r3, coef sym, even,
   75
             --              Writing output
X"202401",   --17    addi    r0, r9, band_tag
X"533004",   --18    slli    r12, r12, 4
X"027325",   --19    or      r9, r12, r12
X"443000",   --1a    oupt    r0, r12, 0
X"202C06",   --1b    addi    r0, r11, dual_band_tag
X"537404",   --1c    slli    r13, r13, 4
X"02F765",   --1d    or      r11, r13, r13
X"443400",   --1e    oupt    r0, r13, 0
             --              Next set of Coefficients
X"208826",   --1f    addi    r2, r2, next_coef_address
             --              Filters 2,5
X"548DCB",   --20    filter  r2, r3, coef sym, even,
   75
             --              Writing output
X"202402",   --21    addi    r0, r9, band_tag
X"533004",   --22    slli    r12, r12, 4
X"027325",   --23    or      r9, r12, r12
X"443000",   --24    oupt    r0, r12, 0
X"202C05",   --25    addi    r0, r11, dual_band_tag
X"537404",   --26    slli    r13, r13, 4
X"02F765",   --27    or      r11, r13, r13
X"443400",   --28    oupt    r0, r13, 0
             --              Next set of Coefficients
X"208826",   --29    addi    r2, r2, next_coef_address
             --              Filters 3,4
X"548DCB",   --2a    filter  r2, r3, coef sym, even,
   75
             --              Writing output
X"202403",   --2b    addi    r0, r9, band_tag
X"533004",   --2c    slli    r12, r12, 4
X"027325",   --2d    or      r9, r12, r12
X"443000",   --2e    oupt    r0, r12, 0
X"202C04",   --2f    addi    r0, r11, dual_band_tag
X"537404",   --30    slli    r13, r13, 4
X"02F765",   --31    or      r11, r13, r13
X"443400",   --32    oupt    r0, r13, 0
             --      Next Data address
X"204401",   --33    addi    r1, r1, 1
X"0069AB",   --34    sgt     r1, r10, r6
X"118001",   --35    beqz    r6, cont
X"214400",   --36    addi    r5, r1, 0
```

```
X"0BFFCE",    --37   cont  j      next
X"000000",    --38         null
X"000000",    --39         null
X"000000",    --3a         null
X"000000",    --3b         null
X"000000",    --3c         null
X"000000",    --3d         null
X"000000",    --3e         null
X"000000",    --3f         null
B"000001100110000110001110",    --40----------------0,7
B"000001101010011110100011",
B"000001101010011110010110",
B"000001101011111110001000",
B"000001101011110110010101",
B"000001101011100110011010",
B"000001101100101110011110",
B"000001101011011110001110",
B"000001101010111101111101",
B"000000000001000000000100",
B"000011101011111101111000",
B"000011101101010101111011",
B"000011101111110101111001",
B"000011110000010110101000",
B"000011110001110110010110",
B"000011110010010110111000",
B"000011110010111110011010",
B"000011110010101110100101",
B"000011110010111110110101",
B"000011110011100110111000",
B"000011110010100101111111",
B"000011110010010110111000",
B"000011110000111101110001",
B"000000000001000000000100",
B"000001110000100110101101",
B"000001110011110110100000",
B"000001110100011110010000",
B"000001110101100110111111",
B"000001110111111111001010",
B"000001110111110110101010",
B"000001111001111111011010",
B"000001111000111110101011",
B"000001111000100110111001",
B"000001111000001110111111",
B"000001111000000110111101",
B"000001111001110110101101",
B"000001111000010111010010",
B"000001111000100111010110",
B"000011101000100101111101",    ----------------------1,6
B"000011101010011110000110",
```

```
B"000011101100010110011110",
B"000011101100011101101101",
B"000011101011011110011111",
B"000011101001101110000001",
B"000001101001011101111110",
B"000001101110011110110111",
B"000001101010110110100110",
B"000000000001000000000100",
B"000011101100000101111010",
B"000011101110111110001111",
B"000011101100000110001110",
B"000001101110011110101010",
B"000001110001010110100011",
B"000001110011100110100010",
B"000001110100110110100010",
B"000001110100110110100101",
B"000001110011101110010110",
B"000001110000101110010110",
B"000011110000110110011010",
B"000011110010100110011011",
B"000011110001111110110101",
B"000000000001000000000100",
B"000001110010011110011010",
B"000001110101111111001011",
B"000001110010101110100000",
B"000011110011100110010110",
B"000011110101101111010000",
B"000011111000000110101110",
B"000011111011110111100010",
B"000011111011110111100011",
B"000011111001111110100101",
B"000011110111010111010010",
B"000001110110010110100011",
B"000001111000110111011110",
B"000001111011111111001111",
B"000001111010100111011110",
B"000011100101100101101001",
B"000001101000110110011111",
B"000001101100011110011110",
B"000001101011011101110011",
B"000011101001111101111011",
B"000011101111110110110001",
B"000011101101101110010101",
B"000011101001110101101101",
B"000001101010001110001010",
B"000000000001000000000100",
B"000011101011101110011011",
B"000001101010000110010111",
B"000001110000001110011011",
```

25

B"000001110100001111001101",
B"000001101111101110101001",
B"000011110010100110011011",
B"000011110100111110100010",
B"000011110011100110101110",
B"000001101111010110110011",
B"000001110011110111000110",
B"000001110100111110100111",
B"000001101110010110101111",
B"000011110000011110100010",
B"000000000001000000000100",
B"000001110001011110011010",
B"000011110000000110000101",
B"000011110110100110101101",
B"000011110111101110011101",
B"000011110100110110000000",
B"000001111000101111001101",
B"000001111011111111100010",
B"000001111000001110111110",
B"000011110110111101101011",
B"000011111010101111000011",
B"000011111010000111001000",
B"000011110110000110110011",
B"000001111001100111010001",
B"000001111010100111011110",
B"000001101001101101110001",
B"000011100110101110000011",
B"000011101100010110011110",
B"000011101000010101101111",
B"000001101101101110010001",
B"000001101100011110010010",
B"000011101111010110111010",
B"000011101101111101101101",
B"000011001110101011111111",
B"000000000001000000000100",
B"000011101000000101110110",
B"000001110000101110110111",
B"000001101111010110011111",
B"000011110000000110001000",
B"000011110001011110111101",
B"000001101111110110100010",
B"000001110100110110100010",
B"000001110000101110100001",
B"000011110100100101111010",
B"000011110011100110011101",
B"000001110011111110100000",
B"000001110011100110100010",
B"000011101101011110011010",
B"000000000001000000000100",

3,4

```
                B"00000110111001111 0010011",
                B"00001111010000111 0011001",
                B"00001111010010111 1000110",
                B"00000111101010111 1100101",
                B"00000111100011011 0101111",
                B"00001111010000011 0100101",
                B"00001111101111011 1100010",
                B"00001111010110111 1000011",
                B"00000111100101011 1010110",
                B"00000111100110011 0111011",
                B"00001111100101111 0110001",
                B"00001111101101011 1011110",
                B"00000111011000011 0100010",
                B"00000111101010011 1011110"
                                          );
    begin
      mem(program'range) := program;
    end load;


  begin
    load;
    -- initialize output
    d <= disabled_tlns_word;

    -- process memory read
    loop
      -- wait for a command, valid on leading edge of clk
      wait on clk until rising_edge(clk);
      -- decode address and perform command if selected
      word_address := to_integer(unsigned(to_bitvector(a)));
      if word_address <= high_address then
        if(ir_mem_enable = '1') then
            d <= To_X01( bit_vector(mem(word_address)) );
        end if;
      end if;
    end loop;
  end process mem_behavior;

end architecture preloaded;
```

A.3.4 The Test Bench Data Memory

The VHDL code for the data memory includes memory addresses which have been loaded with 2DLNS value of 1. Since the order of all filters in filterbank application

is 75, the last 75 locations of memory before the incoming input data, is filled with 1. Therefore, it is guaranteed that for the first 75 iteration of MAC operations, no overflow occurs, although they are not valid values and should be ignored.

```
library ieee;
library work;

use ieee.std_logic_1164.all,
    work.tlns_types.all,
    work.numeric_bit.all;

entity data_memory is
    generic ( data_memory_size : positive := 1024);
    port ( clk : in std_logic;
           mem_a : in mem_bus_addr;
           mem_d_in : in tlns_bus_word;
           mem_d_out : out tlns_bus_word;
           mem_write_en : in std_logic;
           mem_enable : in std_logic);
end entity data_memory;


architecture behavior of data_memory is
begin

  data_memory_behavior : process is

    constant high_address : natural := data_memory_size - 1;

    type data_memory_array is array ( natural range <> ) of
       tlns_bus_word;

    variable row_address : natural;
    variable write_access : boolean;
    variable data_memory : data_memory_array(0 to high_address)
      := ( others => X"000000" );

    -- The filterbank input data  are loaded into memory
    procedure load is

      constant data : data_memory_array
        := ( B"00000000000000000000000000",    -- 0
             B"00000000000000000000000000",    -- 1
             B"00000000000000000000000000",    -- 2
             B"00000000000000000000000000",    -- 3
             B"01000001010001000001010 0",     -- 4
```

```
B"01000001010001000010100",    -- 5
B"01000001010001000010100",    -- 6
B"01000001010001000010100",    -- 7
B"01000001010001000010100",    -- 8
B"01000001010001000010100",    -- 9
B"01000001010001000010100",    -- a
B"01000001010001000010100",    -- b
B"01000001010001000010100",    -- c
B"01000001010001000010100",    -- d
B"01000001010001000010100",    -- e
B"01000001010001000010100",    -- f
B"01000001010001000010100",    -- 10
B"01000001010001000010100",    -- 11
B"01000001010001000010100",    -- 12
B"01000001010001000010100",    -- 13
B"01000001010001000010100",    -- 14
B"01000001010001000010100",    -- 15
B"01000001010001000010100",    -- 16
B"01000001010001000010100",    -- 17
B"01000001010001000010100",    -- 18
B"01000001010001000010100",    -- 19
B"01000001010001000010100",    -- 1a
B"01000001010001000010100",    -- 1b
B"01000001010001000010100",    -- 1c
B"01000001010001000010100",    -- 1d
B"01000001010001000010100",    -- 1e
B"01000001010001000010100",    -- 1f
B"01000001010001000010100",    -- 20
B"01000001010001000010100",    -- 21
B"01000001010001000010100",    -- 22
B"01000001010001000010100",    -- 23
B"01000001010001000010100",    -- 24
B"01000001010001000010100",    -- 25
B"01000001010001000010100",    -- 26
B"01000001010001000010100",    -- 27
B"01000001010001000010100",    -- 28
B"01000001010001000010100",    -- 29
B"01000001010001000010100",    -- 2a
B"01000001010001000010100",    -- 2b
B"01000001010001000010100",    -- 2c
B"01000001010001000010100",    -- 2d
B"01000001010001000010100",    -- 2e
B"01000001010001000010100",    -- 2f
B"01000001010001000010100",    -- 30
B"01000001010001000010100",    -- 31
B"01000001010001000010100",    -- 32
B"01000001010001000010100",    -- 33
B"01000001010001000010100",    -- 34
B"01000001010001000010100",    -- 35
```

```
                B"01000001010001000010100",    -- 36
                B"01000001010001000010100",    -- 37
                B"01000001010001000010100",    -- 38
                B"01000001010001000010100",    -- 39
                B"01000001010001000010100",    -- 3a
                B"01000001010001000010100",    -- 3b
                B"01000001010001000010100",    -- 3c
                B"01000001010001000010100",    -- 3d
                B"01000001010001000010100",    -- 3e
                B"01000001010001000010100",    -- 3f
                B"01000001010001000010100",    -- 40
                B"01000001010001000010100",    -- 41
                B"01000001010001000010100",    -- 42
                B"01000001010001000010100",    -- 43
                B"01000001010001000010100",    -- 44
                B"01000001010001000010100",    -- 45
                B"01000001010001000010100",    -- 46
                B"01000001010001000010100",    -- 47
                B"01000001010001000010100",    -- 48
                B"01000001010001000010100",    -- 49
                B"01000001010001000010100",    -- 4a
                B"01000001010001000010100",    -- 4b
                B"01000001010001000010100",    -- 4c
                B"01000001010001000010100"     -- 4d
                                            );
  begin
    data_memory(data'range) := data;
  end load;

begin
  load;
  -- initialize output
  mem_d_out <= disabled_tlns_word;

  -- process memory cycles
  loop
    -- wait for a lw or sw instruction, valid on leading edge of clk
    wait on clk until rising_edge(clk);
      -- decode address and perform command if selected
      row_address := to_integer(unsigned(to_bitvector(mem_a)));
      write_access := mem_write_en = '1';
      if row_address <= high_address then
        if(mem_enable = '1') then
          if write_access then
            -- write cycle
            data_memory(row_address) := mem_d_in ;
            mem_d_out <= disabled_tlns_word;
          else
            -- read cycle
```

```
            mem_d_out <= data_memory(row_address);
          end if;
        else
          mem_d_out <= disabled_tlns_word;
        end if;
      end if;
    end loop;

  end process data_memory_behavior;

end architecture behavior;
```

A.3.5 The Test Bench Input Data Reader

This module reads the input data for the filterbank application from a file.

```
library ieee;

use ieee.std_logic_1164.all,
    ieee.std_logic_textio.all,
    work.tlns_types.all,
    work.numeric_bit.all;

use std.textio.all;

entity input_gen is
  port ( clk : in std_logic;
         data_in : out external_data;
         out_en : in std_logic);
end entity input_gen;

architecture behavior of input_gen is
begin

  -- this process reads the input from a file
  read_input : process(clk) is

    file input_file : text open read_mode is "filter_input";
    variable line_in : line;
    variable data : external_data;


  begin
    if rising_edge(clk) then
     -- reads input when an output is completed
     if out_en = '1' then
```

```
        readline(input_file, line_in);
        read(line_in, data);
        data_in <= data;
        if endfile(input_file) then
          assert false report "Simulation is complete. End of Stimulus
              File"
                  severity note;
        end if;
      end if;
    end if;
  end process read_input;

end architecture behavior;
```

VITA AUCTORIS

Mahzad Azarmehr was born in Esfahan, Iran, on 1965. She received her B.A.Sc. degree in electrical engineering in 1990 from Tehran University. She is currently a candidate in the electrical and computer engineering M.A.Sc. program at the University of Windsor. Her research interests include VLSI circuit design, computer arithmetic, HDL synthesis and digital signal processing.