# Formal Verification of VIPER's ALU

Wai Wong
University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG

15 April 1993

# Contents

# List of Figures



# List of Tables



# Acknowledgement


The research described in this report was supported by a grant No. AT/2029/266 from the Defence Research Agency (Malvern) (formly RSRE). This report first appeared as the final report of the above contract.

The author would like to thank Dr. C. Pygott of DRA(Malvern) who provided the opportunity for me to carry out the research, and gave valuable comments to the work througout the whole period. I would also like to thank Dr. M. J. C. Gordon who carefully guided me through the whole project and commented on a draft of this report. Dr. P. Curzon also commented on a draft of the report. Furthermore, I would like to thank everyone in the Hardware Verification Group who had helped me during the project.

# Abstract

This research report describes the formal verification of an arithmetic logic unit of the VIPER microprocessor. VIPER is one of the first processors designed using formal methods. A formal model in HOL has been created which models the ALU at two levels: on the higher level, the ALU is specified as a function taking two 32-bit operands and returning a result; on the lower level, the ALU is implemented by a number of 4-bit slices which should takes the same operands and returns the same result. The ALU is capable of performing thirteen different operations. A formal proof of functional equivalence of these two levels has been completed successfully. The complete HOL text of the ALU formal model and details of the proof procedures are included in this report. It has demonstrated that the HOL system is powerful and efficient enough to perform formal verification of realistic hardware design.

# 1 Introduction

This report describes the verification of the Arithmetic Logic Unit (ALU) of the VIPER microprocessor. VIPER is one of the first microprocessors which has been designed using formal methods. The functional behaviour of VIPER was specified in three separate levels with increasing details: the high-level specification, the host machine level and the electronic block level [4] [3]. Formal verification using the HOL theorem prover was carried out by Cohn who showed the equivalence between the high-level and the host machine level, and between the high-level and the electronic block level [1] [2]. Throughout this verification the ALU was treated as a black box whose functional behaviour was specified as a function in the HOL logic. However, the ALU was implemented by a number of bit slices in the design. Since the correctness of the ALU function is crucial to the correct functioning of VIPER, it is very interesting to see whether the bit slice implementation of the ALU is functionally correct. A manual proof of this correctness has been carried out by Pygott in NODEN_HDL. He concluded in [6] that although it was not a complete formal proof, it shown convincingly that the NODEN_HDL ALU specification and implementation are functionally equivalent. Nevertheless, it is still worthwhile to perform a formal proof to confirm the functional equivalence between the specification and the implementation. This will be of benefit to anyone who wishes to use the microprocessor in any critical applications as one will have more confidence in the correct functioning of the device.

This report describes the formal proof of the ALU in the HOL theorem prover. The functional equivalence between the two levels of the ALU model is confirmed. This report contains sufficient details for the designers to follow the proof and for any independent experts to check its validity.

The report begins by describing the ALU informally, and this is followed by a description of the ALU formal model. The strategy of the proof is then outlined, and some typical operations are described in detail. A conclusion is drawn based on the proof. Some notes on the notations used in the report, listings of original NODEN_HDL description can be found in the appendices.

# 2 The ALU models

The ALU has been defined in two levels: the *specification* and the *implementation*. They were written in the hardware specification language NODEN_HDL. This section begins with an informal description of the ALU, a discussion on how to represent this model in HOL follows, and then the formal description of the ALU is presented.

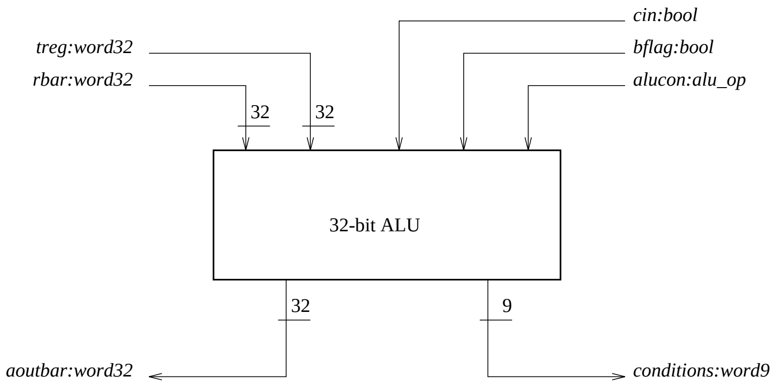


Figure 1: A 32-bit ALU

## 2.1 An informal description

At the specification level, the ALU is viewed as a black box which is capable of performing thirteen different operations. It is specified as a function which takes as its inputs two 32-bit words, namely $rbar$ and $treg$, as the operands, two boolean flags $cin$ and $bflag$, and a value $alucon$ specifying the required operation. It delivers a 32-bit word $aoutbar$ as the result of the operation and a 9-bit word $conditions$ indicating various status values. This view of the ALU is shown in Figure 1.

At the implementation level, the ALU is constructed from eight 4-bit slices. Each of these slices is capable of performing the same set of operations as the specification but on smaller operands, namely 4-bit words. Each of these 4-bit slices is treated as a black-box in this verification. A block diagram of the slice is shown in Figure 2. It takes two 4-bit words as its operands, namely $rabr$ and $treg$, two boolean flag inputs $cin$ and $srbar$, and a value $alucon$ indicating the required operation. It outputs the main result of operation as a 4-bit word $aoutbar$, a carry $pg$ and six boolean flags ($aout4$, $r4$, $r3$, $r1$, $rm31$, $zero$) indicating various status values.

Following conventional design found in many ALUs, these eight slices are organized into a two tier structure: the lower tier consists of groups of four slices, each group forming a 16-bit unit; the upper tier consists of two 16-bit units.

A schematic block diagram showing the interconnection of the 4-bit slices within a 16-bit unit can be found in Figure 3. On the input side of a 16-bit unit, there are two 16-bit words $rbar$ and $treg$ as its operands, two boolean flags $srbar$ and $cin$. Then, there are four inputs indicating the required operation of each of the slices ($alucn_0$ through $alucn_3$). In theory, each of the slices may perform a different operation at the same time if the values to the inputs $alucn_i$ are different. In fact, such situations should never happen. As can be seen in

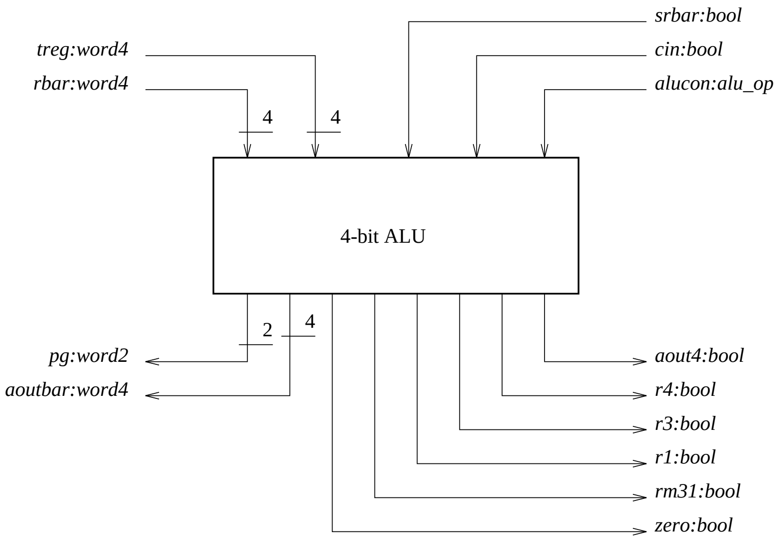


Figure 2: A 4-bit ALU

the formal description, the same value is always passed to these four inputs at the same time. The block labelled 'LOOK AHEAD' in the figure implements the carry lookahead circuitry. It combines the carry output *pg* from each slice and the carry input *cinbar* into this unit and distributes the appropriate carry input back to each slice. This appears to involve a feedback loop. When translating literally into HOL, it requires mutually dependent local variables in function definitions. This is not supported in the HOL system. This problem is solved by breaking down the 'LOOK AHEAD' block into four smaller blocks, one for each slice. Each smaller block then takes input only from its least significant slice(s), so the mutual dependence is eliminated.

A schematic block diagram showing the 32-bit ALU implementation constructed from two 16-bit units can be found in Figure 4. This implementation has exactly the same inputs and outputs as the specification. The block labelled 'INVCIN_SRS' provides the *srbar* inputs to the 16-bit units. Like the carry lookahead block in the 16-bit unit, this also appears to involve a feedback loop. In fact, this loop is even tighter than the one in the 16-bit unit. From the diagram, it can be seen the output $rt31$ of the more significant 16-bit unit '16-bit ALU 1' is combined in the 'INVCIN_SRS' block with the input *cin* and *bflag*, and the output of this block is routed back to the same unit as the *srbar* input. This input is only used by the most significant slice, but the output $rt31$ is also generated from the same slice. Therefore, the loop is actually around the most significant 4-bit slice. However, when examining the original definition further, it is discovered that $rt31$ is only the most significant bit of the first operand *rbar*. Based on this

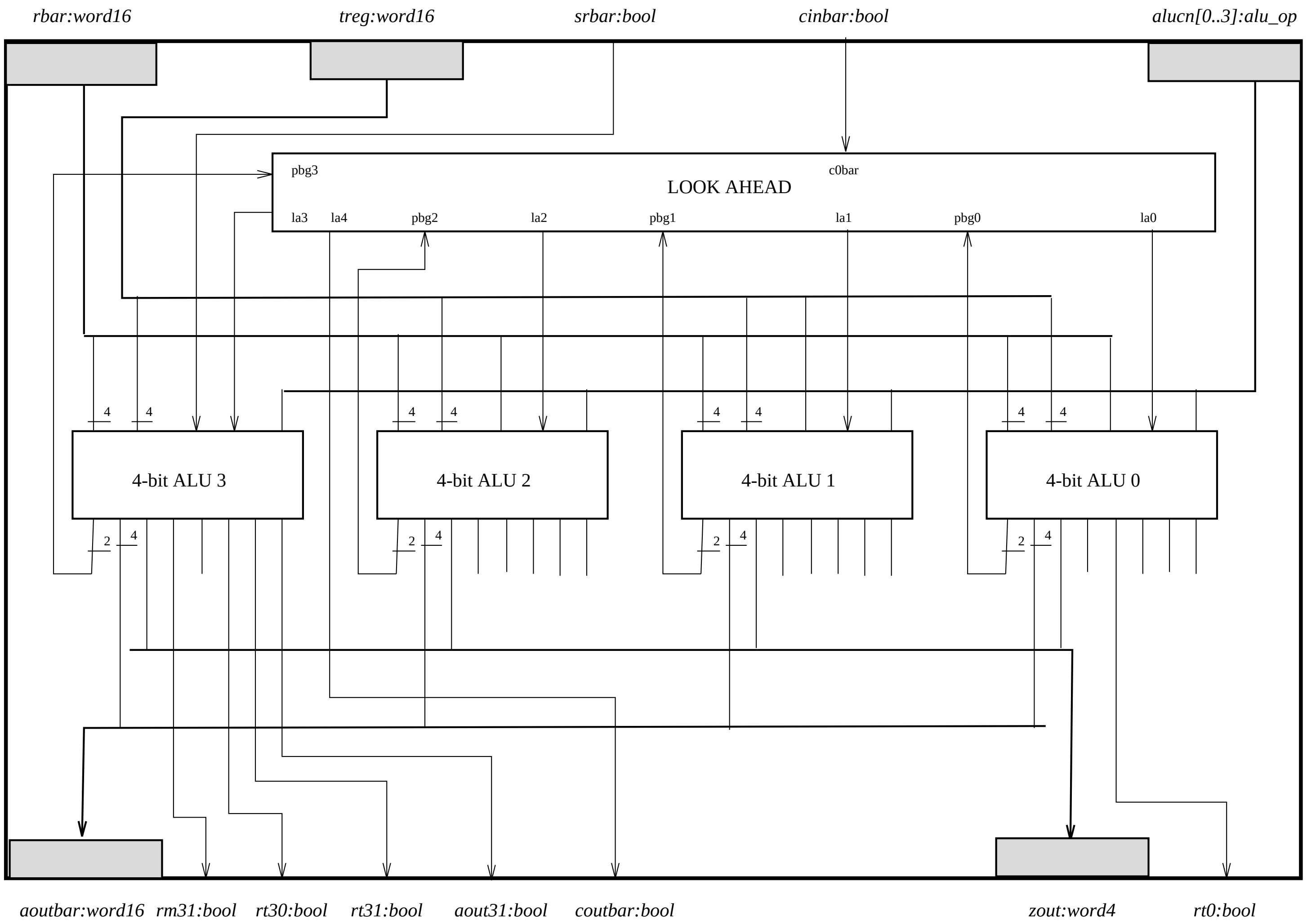


Figure 3: A 16-bit ALU constructed from four 4-bit slices

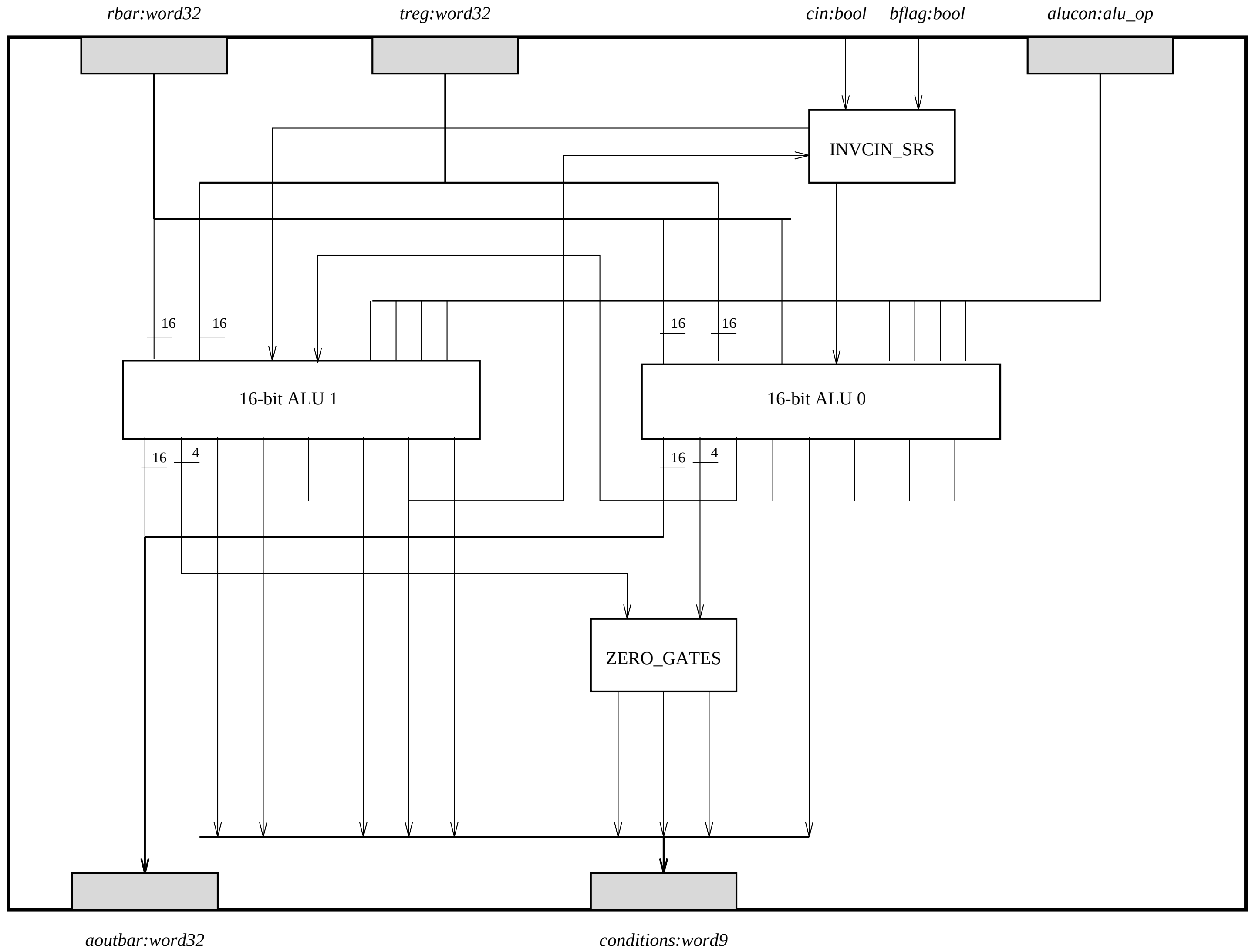


Figure 4: A 32-bit ALU constructed from two 16-bit ALUs

| | |
|---|---|
| `alu_and` | bitwise AND $r$ and $m$ |
| `alu_rmb` | bitwise AND $r$ and NOT $m$ |
| `alu_xor` | bitwise XOR $r$ and $m$ |
| `alu_nor` | bitwise NOR $r$ and $m$ |
| `alu_com` | complement $m$ (bitwise NOT) |
| `alu_m` | always return $m$ |
| `alu_r` | always return $r$ |
| `alu_0` | always return 0 (zero) |
| `alu_sr` | shift $r$ to the right |
| `alu_sl` | shift $r$ to the left |
| `alu_add` | $r + m + c_{in}$ |
| `alu_sub` | subtract $m$ from $r$ |
| `alu_inc` | increment $r$ |

Table 1: ALU operations

observation, the loop can be eliminated by connecting the most significant bit of the first operand directly to the 'INVCIN_SRS' block.

The ALU is capable of performing the thirteen different operations which are listed in Table 1. The left column lists the mnemonic names of the operations and the right column describes the operations. The variables $r$ and $m$ are the first and second operand, respectively, and $c_{in}$ is the carry input. The operations can be divided into two groups: *bitwise* and *arithmetic* operations. The latter groups contains the addition, subtraction, increment and shift-left operations. These are more complicated than the bitwise ones. The subtraction is actually specified as adding the negated second operand to the first, and the increment operation is specified as adding the carry input bit to the operand.

## 2.2 Creating the formal model

After the informal description of the ALU in Section 2.1, a discussion of how to formally represent the ALU in HOL will help the reader to understand the formal model and the proof described in subsequent sections.

The most important problem in creating the formal model is to preserve the meaning of the original design or description in whatever language it has been written. If it has been written in a formal language which has a well-defined and well-understood semantics, the problem can be solved relatively easily. If the verification is to be carried out in the same language as the original description there may even not be a problem. In practice, this will be very rare since almost all engineering requirements and designs will first be described in natural language.

These will then be translated or re-written in more formal language(s) such as programming languages and hardware description languages. How accurate the translation is largely depends on how precise the source description is and how well understood.

In the current project, the description of ALU model provided by Defence Research Agency (Malvern) was written in the hardware description language NODEN_HDL. This is a well-defined language with very little ambiguity. Since the verification is to be carried out in a mechanical theorem prover, the HOL system, which does not support NODEN_HDL, the translation is necessary.

Although NODEN_HDL is a hardware description language whereas HOL implements simple typed lambda calculus, many features are very similar, that is there are one-one correspondences between some similar features in these two languages. Both languages are well typed: all variables must possess a type. Similar basic built-in types exist in both languages, such as booleans, natural numbers and tuples. Both languages allow user-defined functions and local variables in these functions. Taking advantage of this similarity, the basic principle in translating the ALU description from NODEN_HDL into HOL is to keep the new description as close to the original as possible using corresponding features in the two languages. In addition, the same names are chosen for all functions and variables with only a few exceptions. For example, the function `ALU` that is the ALU specification in NODEN_HDL is translated into HOL as a function of the same name which has the same number of inputs and outputs in the same order. It is believed that this principle and naming convention will help to ensure that the translation is faithful to the original model. Any discrepancies between the two descriptions will be pointed out clearly and reasons given when the formal model is presented in Section 2.4.

## 2.3 Representation of words

Words or bit vectors are one of the fundamental data objects one has to deal with in hardware design, specification and verification. They are ubiquitous. Words in NODEN_HDL are represented by the type `word`$n$ where $n$ is the size, for example, 8-bit words are of type `word8`. These types are represented as boolean vectors of the appropriate length. Every vector has a fixed length at compile time. There are built-in vector operations which can be applied to vectors of any length. For example, the bits can be accessed using the indexing operation. Two words of size `m` and `n` can be concatenated to form a new word of size `(m + n)` using the vector concatenation operator `CONC`.

When the original VIPER verification project was carried out, there was no high level support for modelling words in the HOL system, and there was not even proper support for defining recursive types. In the original VIPER verification, words were modelled by simply introducing into the HOL logic a new type named

`:word`$n$ for each required word size $n$ together with a number of new constants which stand for the basic operators of the new type. Axioms describing the basic properties of words had to be added to the system to allow reasoning about word operations. This was not very satisfactory and against the common practice of using only definitional extension to the logic which has since been established in the HOL user community.

One possible approach is to define a type `:word`$n$ for every required size of words and a number of operators of each of the newly defined types. However, unlike NODEN_HDL, list operators may not be used on these words even if their underlying representation is list. Considering word concatenation as an example, an operator is required for every combination of `m` and `n`. This is not quite satisfactory either.

As part of the work carried out, a new HOL library `word` has been developed which takes a more pragmatic approach and provides a generic and flexible infrastructure for modelling bit vectors. This is used in the verification of the ALU. The major features in this library are:

- a polymorphic type `:(*)word` to represent all words;
- using restricted quantifiers to simulate dependent types for different word sizes;
- a number of generic word operators;
- many theorems about the basic properties of bit vectors.

The polymorphic type `:(*)word` allows the user to create instances which are better suited to particular applications. In this project, boolean words, i.e., words of type `:(bool)word` are used exclusively. This also allows words of different sizes to have the same basic properties and to share the same operators. For example, $\mathsf{WNOT}$ is the bitwise negation operator which can be applied to a word of any size.

Restricted universal quantifier is defined by the following equation:

$$\vdash (\forall x :: P.\, Q\, x) = (\forall x.\, P\, x \supset Q\, x)$$

where the double colon (::) indicates that the universal quantified variable $x$ is restricted by the predicate $P$. This has the same meaning (by definition) as the implication on the right hand side. The syntax of restricted quantifiers closely resembles the syntax of types.

The predicate $\mathsf{PWORDLEN}\, n$ is defined in the library to discriminate words of different sizes. This predicate evaluates to $\mathsf{T}$ when applied to a word $w$ if and only if $w$ is of size $n$. Using restricted quantifiers with this predicate, one can express the statement 'for all words $w$ of size $n$, ...' as

$$\forall w :: \mathsf{PWORDLEN}\, n.\ \ldots$$

Most of the theorems in the library are restricted quantifications like this.

The user can also define predicates for specific word size in terms of the predefined function PWORDLEN, such as defining word32 by the equation

$$\vdash \mathsf{word32} = \mathsf{PWORDLEN}\,32$$

Then, one can write expressions like $\forall w :: \mathsf{word32}. \ldots$ Other useful constants defined in the library are listed in Table 2. For convenience and readability, all these constants in the 'Basic word operations' group may be used in the definition of the ALU model, but in the proof, these will be converted into a smaller canonical set of basic operations. This set includes only the following: WCAT, SEG and BIT.

## 2.4 A formal ALU model in HOL

This section describes a formal model of the ALU in HOL. This is based on the original NODEN_HDL description. The translation from NODEN_HDL to HOL follows the principle described in Section 2.2 and uses the `word` library. This model consists of two levels: the specification and the implementation. After comparing the HOL description with the original NODEN_HDL description, it is believed that they are faithful to each other.

The HOL description of the ALU model is organized into a hierarchy of five theories. Figure 5 shows the ancestry of these theories. Details of the theories will be described in the following sections[1], meanwhile, a brief outline of the theories is given here:

`LOGIC` — definitions of several logical connectives;

`WORD_WIDTHS` — definitions of predicates for required word sizes;

`COMMON` — definitions of type and functions common to both levels;

`ALU` — the ALU specification model;

`ALUBIT` the ALU implementation model.

### 2.5 Logical connectives—the theory `LOGIC`

This file contains definitions of a number of basic logical connectives. They are either synonyms of HOL built-in constants or simple combinations of them. (This was taken from the original VIPER verification. They were defined to improve readability.)

There are several theorems in this theory which assert the facts that these connectives are equivalent to the built-in constants.

[1]Text in these sections are generated from the master file using the utility program `mweave`. Appendix A will explain this process briefly.

| NAME | TYPE | DESCRIPTION |
|---|---|---|
| **Basic word operations** | | |
| PWORDLEN | `:num -> (*)word -> bool` | word size discriminator |
| WORDLEN | `:num -> (*)word -> num` | word size |
| SEG | `:num -> num -> (*)word -> (*)word` | word segment |
| BIT | `:num -> (*)word -> *` | a single bit |
| MSB | `:(*)word -> *` | most significant bit |
| LSB | `:(*)word -> *` | least significant bit |
| WCAT | `:(*)word # (*)word -> (*)word` | word concatenation |
| WSPLIT | `:num -> (*)word -> ((*)word # (*)word)` | splitting a word into two parts |
| **Bitwise operations** | | |
| PBITOP | `:((*)word -> (**)word) -> bool` | predicate for bitwise unary operators |
| PBITBOP | `:((*)word -> (**)word -> (***)word) -> bool` | predicate for bitwise binary operators |
| SHR | `:* -> (*)word -> ((*)word # *)` | shift right |
| SHL | `:(*)word -> * -> (* # (*)word)` | shift left |
| WNOT | `:(bool)word -> (bool)word` | bitwise NOT |
| WAND | `:(bool)word -> (bool)word -> (bool)word` | bitwise AND |
| WOR | `:(bool)word -> (bool)word -> (bool)word` | bitwise OR |
| WXOR | `:(bool)word -> (bool)word -> (bool)word` | bitwise exclusive-OR |
| **Word–natural number conversion** | | |
| BV | `:bool -> num` | convert a bit to a number |
| VB | `:num -> bool` | convert a number to a bit |
| BNVAL | `:(bool)word -> num` | convert a word to a number |
| NBWORD | `:num -> num ->(bool)word` | convert a number to a word |

Table 2: Pre-defined constants in the `word` library

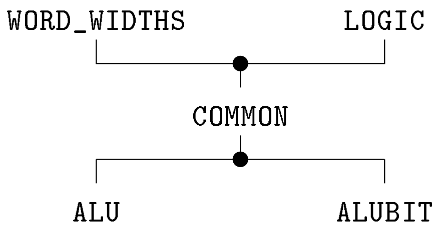


Figure 5: Theory hierarchy of the ALU definitions

#### 2.5.1 XOR — exclusive OR

**HOL Definition** (`XOR_DEF`)

$$\vdash \forall b_1\, b_2.\, b_1 \ \mathsf{XOR}\ b_2 = b_1 \wedge \neg b_2 \vee b_2 \wedge \neg b_1$$

#### 2.5.2 NOT

**HOL Definition** (`NOT_DEF`)

$$\vdash \forall b.\, \mathsf{NOT}\, b = \neg b$$

#### 2.5.3 OR

**HOL Definition** (`OR_DEF`)

$$\vdash \forall b_1\, b_2.\, b_1 \ \mathsf{OR}\ b_2 = b_1 \vee b_2$$

#### 2.5.4 XNOR — exclusiv NOR

**HOL Definition** (`XNOR_DEF`)

$$\vdash \forall b_1\, b_2.\, b_1 \ \mathsf{XNOR}\ b_2 = (b_1 = b_2)$$

#### 2.5.5 AND

**HOL Definition** (`AND_DEF`)

$$\vdash \forall b_1\, b_2.\, b_1 \ \mathsf{AND}\ b_2 = b_1 \wedge b_2$$

**2.5.6 Theorems** We can now prove some simple theorems to equate the new connectives to HOL built-in constants so they can be used in the proof to rewrite these new constants.

**HOL Theorem** (`NOT_THM1`) $\vdash \mathsf{NOT} = \$\neg$

**HOL Theorem** (`OR_THM1`) $\vdash \$\mathsf{OR} = \$\vee$

**HOL Theorem** (`AND_THM1`) $\vdash \$\mathsf{AND} = \$\wedge$

**HOL Theorem** (`XNOR_THM1`) $\vdash \$\mathsf{XNOR} = \$=$

**HOL Theorem** (`XOR_THM1`) $\vdash b1 \ \mathsf{XOR}\ b2 = \neg(b1 = b2)$

## 2.6 Declare required word sizes—the theory WORD_WIDTHS

In this theory, several predicates are defined for discriminating words of different sizes.

**2.6.1 Function for declaring word size** The ML function `declare_word_sizes` is defined which takes a list of numbers and defines a HOL constant for each of the elements in the list. The names of the constants are $\mathsf{word}n$. It represents a predicate which will be $\mathsf{T}$ if its boolean word argument is of size $n$. The new constants are defined in terms of $\mathsf{PWORDLEN}$ in the `word` library. The definition is stored under the name `word`$n$`_def`.

**2.6.2 Declare required word sizes** For the ALU proof, predicates of the following word sizes are needed: [4; 16; 32]. The ML function defined above is called, and the resulting definitions are listed below:

**HOL Definition** (`word4_def`)

$$\vdash \mathsf{word4} = \mathsf{PWORDLEN}\,4$$

**HOL Definition** (`word16_def`)

$$\vdash \mathsf{word16} = \mathsf{PWORDLEN}\,16$$

**HOL Definition** (`wor32_def`)

$$\vdash \mathsf{word32} = \mathsf{PWORDLEN}\,32$$

## 2.7 Common definitions–the theory COMMON

This theory contains a number of definitions which are common to both the specification and the implementation of the ALU.

**2.7.1 The type `:alu_op`** The enumeration type `":alu_op"` represents the possible ALU operations. There are 13 operations defined. The names and their semantics are listed in the table below where $r$ and $m$ are the operands, $c_{in}$ is the carry input and $b$ is a status flag.

| | |
|---|---|
| alu_and | bitwise AND $r$ and $m$ |
| alu_rmb | bitwise AND $r$ and NOT $m$ |
| alu_0 | always return 0 (zero) |
| alu_m | always return $m$ |
| alu_com | complement $m$ (bitwise NOT) |
| alu_r | always return $r$ |
| alu_sr | shift $r$ to the right |
| alu_xor | bitwise XOR $r$ and $m$ |
| alu_nor | bitwise NOR $r$ and $m$ |
| alu_sl | shift $r$ to the left |
| alu_add | $r + m + c_{in}$ |
| alu_sub | $r + (\mathsf{NOT}\, m) + c_{in}$ |
| alu_inc | $r + c_{in}$ |

The ML function `define_type` in the type definition package is used to defint the type `":alu_op"`. The type specification passed to this function is

```
`alu_op = alu_and | alu_rmb | alu_0   | alu_m   | alu_com |
          alu_r   | alu_sr  | alu_xor | alu_nor | alu_sl  |
          alu_add | alu_sub | alu_inc`
```

**2.7.2 IS_BITOP_DEF** The predicate $\mathsf{IS\_BITOP}$ is defined to discriminate the ALU operations. For any operation *aluop*, it is a bitwise operation if it satisfies this predicate, i.e., $\mathsf{IS\_BITOP}\ aluop = \mathsf{T}$.

**HOL Definition** (`IS_BITOP_DEF`)

$$\begin{aligned}
\vdash\ & (\mathsf{IS_BITOP}\,\mathsf{alu_and} = \mathsf{T}) \wedge (\mathsf{IS_BITOP}\,\mathsf{alu_rmb} = \mathsf{T}) \wedge \\
& (\mathsf{IS_BITOP}\,\mathsf{alu_0} = \mathsf{T}) \wedge (\mathsf{IS_BITOP}\,\mathsf{alu_m} = \mathsf{T}) \wedge \\
& (\mathsf{IS_BITOP}\,\mathsf{alu_com} = \mathsf{T}) \wedge (\mathsf{IS_BITOP}\,\mathsf{alu_r} = \mathsf{T}) \wedge \\
& (\mathsf{IS_BITOP}\,\mathsf{alu_sr} = \mathsf{T}) \wedge (\mathsf{IS_BITOP}\,\mathsf{alu_xor} = \mathsf{T}) \wedge \\
& (\mathsf{IS_BITOP}\,\mathsf{alu_nor} = \mathsf{T}) \wedge (\mathsf{IS_BITOP}\,\mathsf{alu_sl} = \mathsf{F}) \wedge \\
& (\mathsf{IS_BITOP}\,\mathsf{alu_add} = \mathsf{F}) \wedge (\mathsf{IS_BITOP}\,\mathsf{alu_sub} = \mathsf{F}) \wedge \\
& (\mathsf{IS_BITOP}\,\mathsf{alu_inc} = \mathsf{F})
\end{aligned}$$

**2.7.3 GET_RM31_DEF** This function is used in the definition of the ALU. The argument $t_{31}$ is the most significant bit of the second operand, i.e., the sign bit if it is interpreted as a 2's complement integer. Depending on the operation *alucon*, it returns $t_{31}$ or negated $t_{31}$ or $\mathsf{F}$.

**HOL Definition** (`GET_RM31_DEF`)

$$\begin{aligned}
\vdash\ & \forall t_{31}\ alucon.\ \mathsf{GET_RM31}\,t_{31}\ alucon = \\
& \quad ((alucon = \mathsf{alu_sub}) \Rightarrow \mathsf{NOT}\,t_{31} \mid \\
& \quad ((alucon = \mathsf{alu_add}) \Rightarrow t_{31} \mid \mathsf{F}))
\end{aligned}$$

**2.7.4 Projection operatores** This section defines a number of projection operators for extracting bits and fields from the values delivered by the two ALU functions. The ALU delivers a pair: the first field `AOUT` is the result of the computation which is a 32-bit word, and the second field `COUT` is a 9-bit word indicating various conditions. Two functions $\mathsf{GET\_AOUT}$ and $\mathsf{GET\_COUT}$ return these two fields, respectively.

**HOL Definition** (`GET_AOUT_DEF`)

$$\vdash \forall aout\ cout.\ \mathsf{GET\_AOUT}\,(aout, cout) = aout$$

**HOL Definition** (`GET_COUT_DEF`)

$$\vdash \forall aout\ cout.\ \mathsf{GET\_COUT}\,(aout, cout) = cout$$

For the condition word returned by $\mathsf{GET\_COUT}$, each bit can be accessed using a function whose name is the name of the required bit prefixed by $\mathsf{COND\_}$. The definitions of these functions are listed below while the meanings of the condition bits will be described in Section 2.8.5.

**HOL Definition** (`COND_AOUT31_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_AOUT31}\, cond = \mathsf{BIT}\, 8\, cond$$

**HOL Definition** (`COND_R0_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_R0}\, cond = \mathsf{BIT}\, 7\, cond$$

**HOL Definition** (`COND_R30_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_R30}\, cond = \mathsf{BIT}\, 6\, cond$$

**HOL Definition** (`COND_R31_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_R31}\, cond = \mathsf{BIT}\, 5\, cond$$

**HOL Definition** (`COND_RM31_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_RM31}\, cond = \mathsf{BIT}\, 4\, cond$$

**HOL Definition** (`COND_COUTBAR_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_COUTBAR}\, cond = \mathsf{BIT}\, 3\, cond$$

**HOL Definition** (`COND_NZTOP12_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_NZTOP12}\, cond = \mathsf{BIT}\, 2\, cond$$

**HOL Definition** (`COND_ZMID4_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_ZMID4}\, cond = \mathsf{BIT}\, 1\, cond$$

**HOL Definition** (`COND_NZBOT16_DEF`)

$$\vdash \forall cond.\ \mathsf{COND\_NZBOT16}\, cond = \mathsf{BIT}\, 0\, cond$$

## 2.8 Specification of the ALU—the theory ALU

**2.8.1 ADD32_DEF** The function ADD32 performs the 32-bit word addition operation. It is specified in terms of the natural number addition '+'. The operands are converted to natural numbers first. If the carry input *cin* is set, a 1 is added to the sum. The result is then converted back to a 33-bit word to accommodate the possible carry.

**HOL Definition** (`ADD32_DEF`)

$$\begin{array}{l} \vdash \forall r\, t :: \mathsf{word32}.\, \forall cin. \\ \quad \mathsf{ADD32}\, r\, t\, cin = \\ \quad\quad (cin \Rightarrow \mathsf{NBWORD}\, 33\, (\mathsf{BNVAL}\, r + \mathsf{BNVAL}\, t + 1) \mid \\ \quad\quad\quad \mathsf{NBWORD}\, 33\, (\mathsf{BNVAL}\, r + \mathsf{BNVAL}\, t)) \end{array}$$

**2.8.2 ADD32BIT_DEF** This is the top level 32-bit word addition function. It takes two 32-bit words and a carry input, and returns a pair consisting of to a 32-bit sum and a carry ouput as the result.

**HOL Definition** (`ADD32BIT_DEF`)

$$\begin{array}{l} \vdash \forall r\, t :: \mathsf{word32}.\, \forall cin. \\ \quad \mathsf{ADD32BIT}\, r\, t\, cin = \\ \quad\quad \mathbf{let}\ sum_{33} = \mathsf{ADD32}\, r\, t\, cin\ \mathbf{in} \\ \quad\quad \mathbf{let}\ aout = \mathsf{SEG}\, 32\, 0\, sum_{33}\ \mathbf{in} \\ \quad\quad \mathbf{let}\ carry = \mathsf{MSB}\, sum_{33}\ \mathbf{in} \\ \quad\quad\quad (aout, carry) \end{array}$$

**2.8.3 FIND_SR_DEF** This function works like a two-input multiplexer. It is called by BITOP to work out the value to be padded to the MSB of the result of a shift right operation. $r_{31}$ is the MSB of the operand. If *cin* is set, FIND_SR returns $r_{31}$, the shift right operation implements an arithmetic shift, ie., the sign bit is copied.

**HOL Definition** (`FIND_SR_DEF`)

$$\begin{array}{l} \vdash \forall cin\, bflag\, r_{31}. \\ \quad \mathsf{FIND\_SR}\, (cin, bflag, r_{31}) = (cin \Rightarrow r_{31} \mid bflag) \end{array}$$

**2.8.4 BITOP_DEF** This is the function specifying the actual operation performed by the ALU. The first two arguments are the operands. The next three, *bflag*, $r_{31}$ and *cin* are boolean flags, they are used only in arithmetic and shift operations. The last argument *op* indicates what operation is to be performed. It returns a pair whose first field is the operation result and whose second field is the carry bit if the operation is arithmetic. Otherwise, it should be *don't care*, i.e. the value `?bool` in NODEN_HDL. However, there is no such value in the built-in boolean type `":bool"` in HOL, so the value F is returned instead. This changes the meaning of the specification, but it is compensated for in the final goal (see Section 3).

**HOL Definition** (`BITOP_DEF`)

$$
\begin{array}{l}
\vdash \forall r\, t :: \mathsf{word32}.\ \forall bflag\, r_{31}\, cin\, op.\\
\quad \mathsf{BITOP}\, r\, t\, bflag\, r_{31}\, cin\, op =\\
\quad\quad \textbf{let } cout\_x = \mathsf{F} \textbf{ in}\\
\quad\quad \textbf{let } tbar = \mathsf{WNOT}\, t \textbf{ in}\\
\quad\quad \textbf{let } rbar = \mathsf{WNOT}\, r \textbf{ in}\\
\quad\quad \textbf{let } sr = \mathsf{FIND\_SR}\, (cin, bflag, r_{31}) \textbf{ in}\\
\quad\quad \textbf{let } r\_xor\_t = r\ \mathsf{WXOR}\ t \textbf{ in}\\
\quad\quad \textbf{let } shift\_right = \mathsf{FST}\, (\mathsf{SHR}\, \mathsf{F}\, sr\, r) \textbf{ in}\\
\quad\quad ((op = \mathsf{alu\_and}) \Rightarrow r\ \mathsf{WAND}\ t, cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_rmb}) \Rightarrow r\ \mathsf{WAND}\ tbar, cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_0}) \Rightarrow \mathsf{NBWORD}\, 32\, 0, cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_m}) \Rightarrow t, cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_com}) \Rightarrow tbar, cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_r}) \Rightarrow r, cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_sr}) \Rightarrow shift\_right, cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_xor}) \Rightarrow r\_xor\_t, cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_nor}) \Rightarrow \mathsf{WNOT}\, (r\ \mathsf{WOR}\ t), cout\_x \mid\\
\quad\quad ((op = \mathsf{alu\_sl}) \Rightarrow \mathsf{ADD32BIT}\, r\, r\, cin \mid\\
\quad\quad ((op = \mathsf{alu\_add}) \Rightarrow \mathsf{ADD32BIT}\, r\, t\, cin \mid\\
\quad\quad ((op = \mathsf{alu\_sub}) \Rightarrow \mathsf{ADD32BIT}\, r\, tbar\, cin \mid\\
\quad\quad ((op = \mathsf{alu\_inc}) \Rightarrow \mathsf{ADD32BIT}\, r\, (\mathsf{NBWORD}\, 32\, 0)\, cin \mid \mathsf{ARB})))))))))))))
\end{array}
$$

**2.8.5 ALU_DEF** This function is the specification of the 32-bit ALU. It takes five inputs: *rbar* is the first operand which has been complemented, *treg* is the second operand, *cin* is the carry input, *bflag* is the status flag, and *alucon* specifies the required operation. The output is a pair whose first field is the operation result and whose second field is a 9-bit word indicating various conditions. The names of these bits and their meanings are described in the table below:

| | |
|---|---|
| `aout31` | MSB of the result |
| `r0` | LSB of the first operand |
| `r30` | 30th of the first operand |
| `r31` | MSB of the first operand |
| `rm31` | MSB of the second operand if addition<br>negated MSB of the second operand if subtraction<br>otherwise, this is F |
| `coutbar` | negated carry output |
| `nztop12` | T if any of the most significant 12 bits are non-zero |
| `zmid4` | T if the middle 4 bits are all zero |
| `nzbot16` | T if any of the least significant 16 bits are non-zero |

**HOL Definition** (`ALU_DEF`)

$\vdash \forall rbar\, treg :: \mathsf{word32}.\ \forall cin\, bflag\, alucon.$

$$
\begin{array}{l}
\mathsf{ALU}\ rbar\ treg\ cin\ bflag\ alucon = \\
\quad \textbf{let}\ r = \mathsf{WNOT}\ rbar\ \textbf{in} \\
\quad \textbf{let}\ t_{31} = \mathsf{MSB}\ treg\ \textbf{in} \\
\quad \textbf{let}\ rm_{31} = \mathsf{GET\_RM31}\ t_{31}\ alucon\ \textbf{in} \\
\quad \textbf{let}\ r_0 = \mathsf{LSB}\ r\ \textbf{in} \\
\quad \textbf{let}\ r_{30} = \mathsf{BIT}\ 30\ r\ \textbf{in} \\
\quad \textbf{let}\ r_{31} = \mathsf{MSB}\ r\ \textbf{in} \\
\quad \textbf{let}\ a\_c = \mathsf{BITOP}\ r\ treg\ bflag\ r_{31}\ cin\ alucon\ \textbf{in} \\
\quad \textbf{let}\ aout = \mathsf{GET\_AOUT}\ a\_c\ \textbf{in} \\
\quad \textbf{let}\ aoutbar = \mathsf{WNOT}\ aout\ \textbf{in} \\
\quad \textbf{let}\ coutbar = \mathsf{NOT}\ (\mathsf{GET\_COUT}\ a\_c)\ \textbf{in} \\
\quad \textbf{let}\ aout_{31} = \mathsf{MSB}\ aout\ \textbf{in} \\
\quad \textbf{let}\ nzbot_{16} = \mathsf{NOT}\ (\mathsf{BNVAL}\ (\mathsf{SEG}\ 16\ 0\ aout) = 0)\ \textbf{in} \\
\quad \textbf{let}\ zmid_4 = (\mathsf{BNVAL}\ (\mathsf{SEG}\ 4\ 16\ aout) = 0)\ \textbf{in} \\
\quad \textbf{let}\ nztop_{12} = \mathsf{NOT}\ (\mathsf{BNVAL}\ (\mathsf{SEG}\ 12\ 20\ aout) = 0)\ \textbf{in} \\
\quad \textbf{let}\ conditions = \mathsf{WORD}\ [aout_{31}; r_0; r_{30}; r_{31}; rm_{31}; coutbar; nztop_{12}; zmid_4; nzbot_{16}]\ \textbf{in} \\
\qquad (aoutbar, conditions)
\end{array}
$$

## 2.9 ALU bit slices—the theory ALUBIT

This theory defines the ALU implementation constructed from 4-bit slices.

**2.9.1 SRSELECT_DEF** This is just a multiplexer function with inverting output.

**HOL Definition** (SRSELECT_DEF)

$$
\begin{array}{l}
\vdash \forall cin\ bflag\ rt_{31}. \\
\qquad \mathsf{SRSELECT}\ cin\ bflag\ rt_{31} = (cin \Rightarrow \mathsf{NOT}\ rt_{31} \mid \mathsf{NOT}\ bflag)
\end{array}
$$

**2.9.2 INVCIN_SRS_DEF** As mentioned in Section 2.1, the block labelled 'INVCIN_SRS' represented by the function of the same name in NODEN_HDL has been broken down, so we do not need to translate this into HOL.

**2.9.3 ZERO_LS16_DEF** The arguments to this function are two 4-bit words. They are the zero condition words returned by the 16-bit ALU units. Each bit indicates whether the operation result from a 4-bit slice is equal to zero. A $\mathsf{T}$ represents a zero result. The function $\mathsf{ZERO\_LS16}$ returns $\mathsf{T}$ if and only if the least significant sixteen bits are *not* all zero.

**HOL Definition** (ZERO_LS16_DEF)

$$
\begin{array}{l}
\vdash \forall lszeros\ mszeros. \\
\qquad \mathsf{ZERO\_LS16}\ lszeros\ mszeros = \mathsf{NOT}\ (lszeros = \mathsf{WORD}\ [\mathsf{T}; \mathsf{T}; \mathsf{T}; \mathsf{T}])
\end{array}
$$

**2.9.4 ZERO_MID4** The arguments to this function are the same as $\mathsf{ZERO\_LS16}$. The function $\mathsf{ZERO\_MID4}$ returns $\mathsf{T}$ if and only if the middle four bits, i.e., bit 16 to 19, are all zero.

**HOL Definition** (`ZERO_MID4_DEF`)

$$\begin{array}{l} \vdash \forall lszeros\; mszeros. \\ \quad \mathsf{ZERO\_MID4}\; lszeros\; mszeros = \mathsf{BIT}\; 0\; mszeros \end{array}$$

**2.9.5 ZERO_MS12** The arguments to this function are the same as $\mathsf{ZERO\_LS16}$. The function $\mathsf{ZERO\_MS12}$ returns $\mathsf{T}$ if and only if the most significant twelve bits are *not* all zero.

**HOL Definition** (`ZERO_MS12_DEF`)

$$\begin{array}{l} \vdash \forall lszeros\; mszeros. \\ \quad \mathsf{ZERO\_MS12}\; lszeros\; mszeros = \\ \quad\quad \mathsf{NOT}\,(\mathsf{SEG}\; 3\; 1\; mszeros = \mathsf{WORD}\,[\mathsf{T};\mathsf{T};\mathsf{T}]) \end{array}$$

**2.9.6 ZERO_GATES_DEF** This function combines the previous three functions to produce a triple of values indicating the zero condition of the operation result. It is used in the 32-bit ALU definition.

**HOL Definition** (`ZERO_GATES_DEF`)

$$\begin{array}{l} \vdash \forall lszeros\; mszeros. \\ \quad \mathsf{ZERO\_GATES}\; lszeros\; mszeros = \\ \quad\quad \mathsf{ZERO\_LS16}\; lszeros\; mszeros, \\ \quad\quad \mathsf{ZERO\_MID4}\; lszeros\; mszeros, \\ \quad\quad \mathsf{ZERO\_MS12}\; lszeros\; mszeros \end{array}$$

**2.9.7 Carry lookahead funtions** The carry look ahead functions $\mathsf{LOOKAHEAD\_}n$ compute the carry input for the $n$th slice in a 16-bit ALU unit. Each 4-bit slice produces a pair of boolean values indicating the carry generation and propagation status. If the generation bit is set, a carry is generated in this slice. Likewise, if the propagation bit is set, a carry input to this slice should be propagated to the next more significant slice. In the original NODEN_HDL description, this carry status is represented as a 2-bit word. In the HOL description, the type `:cla` is defined to represent this carry look ahead value to avoid confusion in bit ordering. This type has two fields: the propagation field and the generation field. Both are boolean. The type constructor is $\mathsf{CLA}$. Two projection operators $\mathsf{CPRO}$ and $\mathsf{CGEN}$ are defined to access these fields, respectively.

**HOL Definition** (`CPRO_DEF`)

$$\vdash \forall p\, g.\, \mathsf{CPRO}\,(\mathsf{CLA}\, p\, g) = p$$

**HOL Definition** (`CGEN_DEF`)

$$\vdash \forall p\, g.\, \mathsf{CGEN}\,(\mathsf{CLA}\, p\, g) = g$$

#### 2.9.8 LOOKAHEAD_0_DEF

This is the carry lookahead function for slice 0. It simply negates the carry input.

**HOL Definition** (`LOOKAHEAD_0_DEF`)

$$\vdash \forall c_0bar.\quad \mathsf{LOOKAHEAD\_0}\; c_0bar = \mathsf{NOT}\; c_0bar$$

#### 2.9.9 LOOKAHEAD_1_DEF

This is the carry lookahead function for slice 1. It takes the carry input to the unit and carry lookahead value from slice 0, and computes the carry input for slice 1.

**HOL Definition** (`LOOKAHEAD_1_DEF`)

$$\begin{array}{l}\vdash \forall c_0bar\; pbg_0.\\ \quad \mathsf{LOOKAHEAD\_1}\; c_0bar\; pbg_0 = \\ \quad\; \mathbf{let}\; p_0bar = \mathsf{CPRO}\; pbg_0\; \mathbf{and}\; g_0 = \mathsf{CGEN}\; pbg_0\; \mathbf{in} \\ \quad\;\; g_0\; \mathsf{OR}\; (\mathsf{NOT}\; p_0bar\; \mathsf{AND}\; \mathsf{NOT}\; c_0bar)\end{array}$$

#### 2.9.10 LOOKAHEAD_2_DEF

This is the carry lookahead function for slice 2. It takes the carry input to the unit and carry lookahead value from slice 0 and 1, and computes the carry input for slice 2.

**HOL Definition** (`LOOKAHEAD_2_DEF`)

$$\begin{array}{l}\vdash \forall c_0bar\; pbg_0\; pbg_1.\\ \quad \mathsf{LOOKAHEAD\_2}\; c_0bar\; pbg_0\; pbg_1 = \\ \quad\; \mathbf{let}\; p_0bar = \mathsf{CPRO}\; pbg_0\; \mathbf{and}\; g_0 = \mathsf{CGEN}\; pbg_0\; \mathbf{in} \\ \quad\; (\mathbf{let}\; p_1bar = \mathsf{CPRO}\; pbg_1\; \mathbf{and}\; g_1 = \mathsf{CGEN}\; pbg_1\; \mathbf{in} \\ \quad\; (\mathbf{let}\; c_1 = g_0\; \mathsf{OR}\; (\mathsf{NOT}\; p_0bar\; \mathsf{AND}\; \mathsf{NOT}\; c_0bar)\; \mathbf{in} \\ \quad\;\; g_1\; \mathsf{OR}\; (\mathsf{NOT}\; p_1bar\; \mathsf{AND}\; c_1)))\end{array}$$

#### 2.9.11 LOOKAHEAD_3_DEF

This is the carry lookahead function for slice 3. It takes the carry input to the unit and carry lookahead value from slice 0, 1 and 2, and computes the carry input for slice 3.

**HOL Definition** (`LOOKAHEAD_3_DEF`)

$$\begin{array}{l}\vdash \forall c_0bar\; pbg_0\; pbg_1\; pbg_2.\\ \quad \mathsf{LOOKAHEAD\_3}\; c_0bar\; pbg_0\; pbg_1\; pbg_2 = \\ \quad\; \mathbf{let}\; p_0bar = \mathsf{CPRO}\; pbg_0\; \mathbf{and}\; g_0 = \mathsf{CGEN}\; pbg_0\; \mathbf{in} \\ \quad\; (\mathbf{let}\; p_1bar = \mathsf{CPRO}\; pbg_1\; \mathbf{and}\; g_1 = \mathsf{CGEN}\; pbg_1\; \mathbf{in} \\ \quad\; (\mathbf{let}\; p_2bar = \mathsf{CPRO}\; pbg_2\; \mathbf{and}\; g_2 = \mathsf{CGEN}\; pbg_2\; \mathbf{in} \\ \quad\; (\mathbf{let}\; c_1 = g_0\; \mathsf{OR}\; (\mathsf{NOT}\; p_0bar\; \mathsf{AND}\; \mathsf{NOT}\; c_0bar)\; \mathbf{in} \\ \quad\; (\mathbf{let}\; c_2 = g_1\; \mathsf{OR}\; (\mathsf{NOT}\; p_1bar\; \mathsf{AND}\; c_1)\; \mathbf{in} \\ \quad\;\; g_2\; \mathsf{OR}\; (\mathsf{NOT}\; p_2bar\; \mathsf{AND}\; c_2)))))\end{array}$$

**2.9.12 LOOKAHEAD_4_DEF** This is the carry lookahead function for a 61-bit unit. It takes the carry input to the unit and carry lookahead value from all the slices, and computes the carry output of the whole unit.

**HOL Definition** (`LOOKAHEAD_4_DEF`)

$$
\begin{array}{l}
\vdash \forall c_0bar\ pbg_0\ pbg_1\ pbg_2\ pbg_3. \\
\quad \mathsf{LOOKAHEAD\_4}\ c_0bar\ pbg_0\ pbg_1\ pbg_2\ pbg_3 = \\
\quad\quad \mathbf{let}\ p_0bar = \mathsf{CPRO}\ pbg_0\ \mathbf{and}\ g_0 = \mathsf{CGEN}\ pbg_0\ \mathbf{in} \\
\quad\quad (\mathbf{let}\ p_1bar = \mathsf{CPRO}\ pbg_1\ \mathbf{and}\ g_1 = \mathsf{CGEN}\ pbg_1\ \mathbf{in} \\
\quad\quad (\mathbf{let}\ p_2bar = \mathsf{CPRO}\ pbg_2\ \mathbf{and}\ g_2 = \mathsf{CGEN}\ pbg_2\ \mathbf{in} \\
\quad\quad (\mathbf{let}\ p_3bar = \mathsf{CPRO}\ pbg_3\ \mathbf{and}\ g_3 = \mathsf{CGEN}\ pbg_3\ \mathbf{in} \\
\quad\quad (\mathbf{let}\ c_1 = g_0\ \mathsf{OR}\ (\mathsf{NOT}\ p_0bar\ \mathsf{AND}\ \mathsf{NOT}\ c_0bar)\ \mathbf{in} \\
\quad\quad (\mathbf{let}\ c_2 = g_1\ \mathsf{OR}\ (\mathsf{NOT}\ p_1bar\ \mathsf{AND}\ c_1)\ \mathbf{in} \\
\quad\quad (\mathbf{let}\ c_3 = g_2\ \mathsf{OR}\ (\mathsf{NOT}\ p_2bar\ \mathsf{AND}\ c_2)\ \mathbf{in} \\
\quad\quad\quad \mathsf{NOT}\ (g_3\ \mathsf{OR}\ (\mathsf{NOT}\ p_3bar\ \mathsf{AND}\ c_3))))))))
\end{array}
$$

**2.9.13 ADDER4_DEF** The function ADDER4 performs the 4-bit word addition operation. It is specified in terms of natural number addition '+'. The operands are converted to natural numbers first. If the carry input $cin$ is set, a 1 is added to the sum. The result is then converted back to a 4-bit word. Unlike the specification model (see definition of 2.8.1), this conversion will lose a possible carry bit. However, this does not matter since the carries are computed by the carry lookahead function PG4.

**HOL Definition** (`ADDER4_DEF`)

$$
\begin{array}{l}
\vdash \forall r\ m :: \mathsf{word4}. \forall cin. \\
\quad \mathsf{ADDER4}\ r\ m\ cin = \\
\quad\quad (cin \Rightarrow \mathsf{SEG}\ 4\ 0\ (\mathsf{NBWORD}\ 5\ (\mathsf{BNVAL}\ r + \mathsf{BNVAL}\ m + 1))\ | \\
\quad\quad\quad \mathsf{SEG}\ 4\ 0\ (\mathsf{NBWORD}\ 5\ (\mathsf{BNVAL}\ r + \mathsf{BNVAL}\ m)))
\end{array}
$$

**2.9.14 PG4_DEF** The function PG4 calculates the carry look ahead value for a 4-bit slice.

**HOL Definition** (`PG4_DEF`)

$$
\begin{array}{l}
\vdash \forall r\ m :: \mathsf{word4}. \\
\quad \mathsf{PG4}\ r\ m = \\
\quad\quad \mathbf{let}\ p = r\ \mathsf{WOR}\ m\ \mathbf{and}\ g = r\ \mathsf{WAND}\ m\ \mathbf{in} \\
\quad\quad (\mathbf{let}\ p_0 = \mathsf{BIT}\ 0\ p\ \mathbf{and}\ p_1 = \mathsf{BIT}\ 1\ p\ \mathbf{and}\ p_2 = \mathsf{BIT}\ 2\ p\ \mathbf{and}\ p_3 = \mathsf{BIT}\ 3\ p\ \mathbf{in} \\
\quad\quad (\mathbf{let}\ g_0 = \mathsf{BIT}\ 0\ g\ \mathbf{and}\ g_1 = \mathsf{BIT}\ 1\ g\ \mathbf{and}\ g_2 = \mathsf{BIT}\ 2\ g\ \mathbf{and}\ g_3 = \mathsf{BIT}\ 3\ g\ \mathbf{in} \\
\quad\quad\quad \mathsf{CLA}\ (\mathsf{NOT}\ (p_0\ \mathsf{AND}\ (p_1\ \mathsf{AND}\ (p_2\ \mathsf{AND}\ p_3)))) \\
\quad\quad\quad\quad (g_3\ \mathsf{OR}\ (p_3\ \mathsf{AND}\ (g_2\ \mathsf{OR}\ (p_2\ \mathsf{AND}\ (g_1\ \mathsf{OR}\ (p_1\ \mathsf{AND}\ g_0))))))))
\end{array}
$$

**2.9.15 ADD4BIT_DEF** The function ADD4BIT is the top level addition function for 4-bit words. It returns a pair: the first field is the sum and the second field is the carry.

**HOL Definition** (ADD4BIT_DEF)

$$\vdash \forall r\, m :: \mathsf{word4}. \forall cin.\\ \mathsf{ADD4BIT}\, r\, m\, cin = \mathsf{ADDER4}\, r\, m\, cin, \mathsf{PG4}\, r\, m$$

**2.9.16 FOURBITOP** This is the function specifying the actual operation performed by the ALU slice. The first two arguments are the operands. The third argument *srbar* supplies the bit to be padded onto the most significant bit when a shift-right operation is performed. This should be the least significant bit shifted out of the next more significant slice. The fourth argument *cin* is the carry input. The last one, *op*, specifies what operation is to be performed. It returns a pair: the first field is the main operation result and the second field is the carry output. In cases where the operation is bitwise, the carry output is irrelevant, it was a *don't care* value in NODEN_HDL. Since there is no such value in HOL, the value (CLA F F) is returned. Because this carry output will not be used in bitwise operation, this change should not affect the correctness of the model.

**HOL Definition** (FOURBITOP_DEF)

$$
\begin{array}{l}
\vdash \forall r\, m :: \mathsf{word4}. \forall srbar\, cin\, op.\\
\quad \mathsf{FOURBITOP}\, r\, m\, srbar\, cin\, op =\\
\quad\; \mathbf{let}\ mbar = \mathsf{WNOT}\, m\ \mathbf{and}\ rbar = \mathsf{WNOT}\, r\ \mathbf{in}\\
\quad\; (\mathbf{let}\ r_3 = \mathsf{BIT}\, 3\, r\ \mathbf{in}\\
\quad\; (\mathbf{let}\ shift\_right = \mathsf{FST}\,(\mathsf{SHR}\,\mathsf{F}\,(\mathsf{NOT}\, srbar)\, r)\ \mathbf{in}\\
\quad\; (\mathbf{let}\ shift\_left = \mathsf{SND}\,(\mathsf{SHL}\,\mathsf{F}\, r\, cin)\ \mathbf{in}\\
\quad\quad ((op = \mathsf{alu\_and}) \Rightarrow r\ \mathsf{WAND}\, m, \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_rmb}) \Rightarrow r\ \mathsf{WAND}\, mbar, \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_0}) \Rightarrow \mathsf{NBWORD}\, 4\, 0, \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_m}) \Rightarrow m, \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_com}) \Rightarrow mbar, \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_r}) \Rightarrow r, \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_sr}) \Rightarrow shift\_right, \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_xor}) \Rightarrow r\ \mathsf{WXOR}\, m, \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_nor}) \Rightarrow \mathsf{WNOT}\,(r\ \mathsf{WOR}\, m), \mathsf{CLA}\,\mathsf{F}\,\mathsf{F} \mid\\
\quad\quad ((op = \mathsf{alu\_sl}) \Rightarrow shift\_left, \mathsf{CLA}\,(\mathsf{NOT}\,(r = \mathsf{WORD}\,[\mathsf{T};\mathsf{T};\mathsf{T};\mathsf{T}]))\, r_3 \mid\\
\quad\quad ((op = \mathsf{alu\_add}) \Rightarrow \mathsf{ADD4BIT}\, r\, m\, cin \mid\\
\quad\quad ((op = \mathsf{alu\_sub}) \Rightarrow \mathsf{ADD4BIT}\, r\, mbar\, cin \mid\\
\quad\quad\quad \mathsf{ADD4BIT}\, r\,(\mathsf{NBWORD}\, 4\, 0)\, cin))))))))))))))
\end{array}
$$

**2.9.17 ALU_4BIT** The funcion ALU_4BIT represents the 4-bit slice. It takes the same arguments as FOURBITOP and returns an 8-tuple. The fields in the return value are listed in the following table:

| | | |
|---|---|---|
| pg4 | :cla | carry lookahead output |
| aout | :(bool)word | the main result (a 4-bit word) |
| zero | :bool | T if and only if the result equals zero |
| t4 | :bool | MSB of the second operand if addition<br>negated MSB of the second operand if subtraction<br>otherwise, this is F |
| r0 | :bool | LSM of the first operand |
| r2 | :bool | bit 2 of the first operand |
| r3 | :bool | MSB of the first operand |
| aout31 | :bool | MSB of the result |

**HOL Definition** (ALU_4BIT_DEF)

$$\begin{array}{l}
\vdash \forall r_4bar\ t_4reg :: \mathsf{word4}. \forall srbar\ cin_4bit\ alucon. \\
\quad \mathsf{ALU_4BIT}\ r_4bar\ t_4reg\ srbar\ cin_4bit\ alucon = \\
\quad\quad \textbf{let}\ r = \mathsf{WNOT}\ r_4bar\ \textbf{in} \\
\quad\quad (\textbf{let}\ aout_pg = \mathsf{FOURBITOP}\ r\ t_4reg\ srbar\ cin_4bit\ alucon\ \textbf{in} \\
\quad\quad (\textbf{let}\ aout = \mathsf{FST}\ aout_pg\ \textbf{in} \\
\quad\quad\quad \mathsf{SND}\ aout_pg, \mathsf{WNOT}\ aout, (aout = \mathsf{NBWORD}\ 4\ 0), \\
\quad\quad\quad \mathsf{GET_RM31}\ (\mathsf{BIT}\ 3\ t_4reg)\ alucon, \mathsf{BIT}\ 0\ r, \mathsf{BIT}\ 2\ r, \mathsf{BIT}\ 3\ r, \mathsf{BIT}\ 3\ aout))
\end{array}$$

**2.9.18 ALU_16BIT** The function ALU_16BIT represents a 16-bit unit. It specifies how to assemble four ALU_4BIT's to form such a unit. It takes eight arguments: the first two are the 16-bit operands; the third *srbar* is a bit to be used to pad the most significant bit when doing shift right; the fourth one is the carry input; and the remaining four specify the required operation for each slice. It returns a triple: the first field is the operation result which is a 16-bit word; the second is a 4-bit word indicating the zero status of the results from each slice; and the last is a 6-bit word indicating various conditions. The names and meanings of the condition word are listed in the following table:

| | |
|---|---|
| c4bar | carry output |
| rm31 | the output bit t4 from the most significant slice |
| rt0 | LSB of the first operand |
| rt30 | bit 14 of the first operand |
| rt31 | MSB of the first operand |
| aout31 | MSB of the result |

Here, we have some difficulty in handling feedback from a single block definition. The LOOK_AHEAD in the original NODEN_HDL description takes input from the bit slices and drives the slices as well. To overcome this, we unfold the LOOK_AHEAD block into a separate lookahead circuit for each individual slice. This is why the LOOK_AHEAD definition is not needed.

**HOL Definition** (ALU_16BIT_DEF)

$$\begin{array}{l}
\vdash \forall r_{16}bar\ t_{16}reg :: \mathsf{word16}. \forall srbar\ cin_{16}bar\ alucn_0\ alucn_1\ alucn_2\ alucn_3. \\
\quad \mathsf{ALU_16BIT}\ r_{16}bar\ t_{16}reg\ srbar\ cin_{16}bar\ alucn_0\ alucn_1\ alucn_2\ alucn_3 = \\
\quad\quad \textbf{let}\ rb_{00\text{-}03} = \mathsf{SEG}\ 4\ 0\ r_{16}bar\ \textbf{and}\ rb_{04\text{-}07} = \mathsf{SEG}\ 4\ 4\ r_{16}bar\ \textbf{in} \\
\quad\quad (\textbf{let}\ rb_{08\text{-}11} = \mathsf{SEG}\ 4\ 8\ r_{16}bar\ \textbf{and}\ rb_{12\text{-}15} = \mathsf{SEG}\ 4\ 12\ r_{16}bar\ \textbf{in}
\end{array}$$

$$
\begin{array}{l}
(\mathbf{let}\ m_{00\text{-}03} = \mathsf{SEG}\,4\,0\,t_{16}reg\ \mathbf{and}\ m_{04\text{-}07} = \mathsf{SEG}\,4\,4\,t_{16}reg\ \mathbf{in} \\
(\mathbf{let}\ m_{08\text{-}11} = \mathsf{SEG}\,4\,8\,t_{16}reg\ \mathbf{and}\ m_{12\text{-}15} = \mathsf{SEG}\,4\,12\,t_{16}reg\ \mathbf{in} \\
(\mathbf{let}\ rbar_{04} = \mathsf{BIT}\,4\,r_{16}bar\ \mathbf{and}\ rbar_{08} = \mathsf{BIT}\,8\,r_{16}bar\ \mathbf{in} \\
(\mathbf{let}\ rbar_{12} = \mathsf{BIT}\,12\,r_{16}bar\ \mathbf{in} \\
(\mathbf{let}\ c_0 = \mathsf{LOOKAHEAD\_0}\,cin_{16}bar\ \mathbf{in} \\
(\mathbf{let}\ alu_0 = \mathsf{ALU\_4BIT}\,rb_{00\text{-}03}\,m_{00\text{-}03}\,rbar_{04}\,c_0\,alucn_0\ \mathbf{in} \\
(\mathbf{let}\ pg_0 = \mathsf{FST}\,alu_0\ \mathbf{and} \\
\quad aoutbar_{\text{-}0} = \mathsf{FST}\,(\mathsf{SND}\,alu_0)\ \mathbf{and} \\
\quad zout_{\text{-}0} = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_0))\ \mathbf{and} \\
\quad rt_0 = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_0))))\ \mathbf{in} \\
(\mathbf{let}\ c_1 = \mathsf{LOOKAHEAD\_1}\,cin_{16}bar\,pg_0\ \mathbf{in} \\
(\mathbf{let}\ alu_1 = \mathsf{ALU\_4BIT}\,rb_{04\text{-}07}\,m_{04\text{-}07}\,rbar_{08}\,c_1\,alucn_1\ \mathbf{in} \\
(\mathbf{let}\ pg_1 = \mathsf{FST}\,alu_1\ \mathbf{and} \\
\quad aoutbar_{\text{-}1} = \mathsf{FST}\,(\mathsf{SND}\,alu_1)\ \mathbf{and} \\
\quad zout_{\text{-}1} = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_1))\ \mathbf{in} \\
(\mathbf{let}\ c_2 = \mathsf{LOOKAHEAD\_2}\,cin_{16}bar\,pg_0\,pg_1\ \mathbf{in} \\
(\mathbf{let}\ alu_2 = \mathsf{ALU\_4BIT}\,rb_{08\text{-}11}\,m_{08\text{-}11}\,rbar_{12}\,c_2\,alucn_2\ \mathbf{in} \\
(\mathbf{let}\ pg_2 = \mathsf{FST}\,alu_2\ \mathbf{and} \\
\quad aoutbar_{\text{-}2} = \mathsf{FST}\,(\mathsf{SND}\,alu_2)\ \mathbf{and} \\
\quad zout_{\text{-}2} = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_2))\ \mathbf{in} \\
(\mathbf{let}\ c_3 = \mathsf{LOOKAHEAD\_3}\,cin_{16}bar\,pg_0\,pg_1\,pg_2\ \mathbf{in} \\
(\mathbf{let}\ alu_3 = \mathsf{ALU\_4BIT}\,rb_{12\text{-}15}\,m_{12\text{-}15}\,srbar\,c_3\,alucn_3\ \mathbf{in} \\
(\mathbf{let}\ pg_3 = \mathsf{FST}\,alu_3\ \mathbf{and} \\
\quad aoutbar_{\text{-}3} = \mathsf{FST}\,(\mathsf{SND}\,alu_3)\ \mathbf{and} \\
\quad zout_{\text{-}3} = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_3))\ \mathbf{and} \\
\quad rm_{31} = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_3)))\ \mathbf{and} \\
\quad coutbar = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_3))))\ \mathbf{and} \\
\quad rt_{30} = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_3)))))\ \mathbf{and} \\
\quad rt_{31} = \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_3))))))\ \mathbf{and} \\
\quad aout_{31} = \mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,alu_3))))))\ \mathbf{in} \\
(\mathbf{let}\ c_4bar = \mathsf{LOOKAHEAD\_4}\,cin_{16}bar\,pg_0\,pg_1\,pg_2\,pg_3\ \mathbf{in} \\
\quad \mathsf{WCAT}\,(\mathsf{WCAT}\,(aoutbar_{\text{-}3}, aoutbar_{\text{-}2}), \mathsf{WCAT}\,(aoutbar_{\text{-}1}, aoutbar_{\text{-}0})), \\
\quad \mathsf{WORD}\,[zout_{\text{-}3}; zout_{\text{-}2}; zout_{\text{-}1}; zout_{\text{-}0}], \\
\quad \mathsf{WORD}\,[c_4bar; rm_{31}; rt_0; rt_{30}; rt_{31}; aout_{31}]))))))))))))))))))
\end{array}
$$

**2.9.19 ALU_32BIT_DEF** This is the 32-bit ALU implementation function. It specifies how to assemble two ALU_16BIT's to form the 32-bit ALU. It takes exactly the same arguments as the specification ALU and returns the same results. The name of this function in NODEN_HDL was `ALU_C`. It has been changed to ALU_32BIT so that it has the same format as the 4-bit slice ALU_4BIT and 16-bit units ALU_16BIT.

We have the same feedback problem here as in the 16-bit block. It occurs in the `INVCIN_SRS` block in the NODEN_HDL description. This block can be unfolded into two simpler blocks:

SRSELECT and cinbar. Even after unfolding, we still have the feedback in the SRSELECT block which takes one of the conditional bit output of the higher 16-bit block, namely rt31, and sends its output back to the same block as the srbar input. After some analysis, we find that rt31 is equivalent to the negated most significant bit of the first operand rbar. So, we added the local variable rb31 and equate it to $\mathsf{NOT}\,(\mathsf{BIT}\,31\,rbar)$.

**HOL Definition** (ALU_32BIT_DEF)

$$
\begin{array}{l}
\vdash \forall rbar\,treg :: \mathsf{word32}.\ \forall cin\,bflag\,alucon.\\
\quad \mathsf{ALU\_32BIT}\ rbar\,treg\,cin\,bflag\,alucon =\\
\quad\ \ \mathbf{let}\ (rb_{16\text{-}31}, rb_{00\text{-}15}) = \mathsf{WSPLIT}\ 16\ rbar\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ (m_{16\text{-}31}, m_{00\text{-}15}) = \mathsf{WSPLIT}\ 16\ treg\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ rbar_{16} = \mathsf{BIT}\ 16\ rbar\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ rb_{31} = \mathsf{NOT}\,(\mathsf{BIT}\ 31\ rbar)\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ srbar = \mathsf{SRSELECT}\ cin\,bflag\,rb_{31}\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ cinbar = \mathsf{NOT}\ cin\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ (aoutbar_{\text{-}0}, lszeros, cond_{\text{-}0}) =\\
\quad\quad\ \ \mathsf{ALU\_16BIT}\ rb_{00\text{-}15}\ m_{00\text{-}15}\ rbar_{16}\ cinbar\ alucon\ alucon\ alucon\ alucon\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ cin\_ms = \mathsf{BIT}\ 5\ cond_{\text{-}0}\ \mathbf{and}\ rt_0 = \mathsf{BIT}\ 3\ cond_{\text{-}0}\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ (aoutbar_{\text{-}1}, mszeros, cond_{\text{-}1}) =\\
\quad\quad\ \ \mathsf{ALU\_16BIT}\ rb_{16\text{-}31}\ m_{16\text{-}31}\ srbar\ cin\_ms\ alucon\ alucon\ alucon\ alucon\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ coutbar = \mathsf{BIT}\ 5\ cond_{\text{-}1}\ \mathbf{and}\ rm_{31} = \mathsf{BIT}\ 4\ cond_{\text{-}1}\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ rt_{30} = \mathsf{BIT}\ 2\ cond_{\text{-}1}\ \mathbf{and}\ rt_{31} = \mathsf{BIT}\ 1\ cond_{\text{-}1}\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ aout_{31} = \mathsf{BIT}\ 0\ cond_{\text{-}1}\ \mathbf{in}\\
\quad\ \ (\mathbf{let}\ (nz\_bot_{16}, z\_mid_4, nz\_top_{12}) = \mathsf{ZERO\_GATES}\ lszeros\ mszeros\ \mathbf{in}\\
\quad\quad\ \ \mathsf{WCAT}\ (aoutbar_{\text{-}1}, aoutbar_{\text{-}0}),\\
\quad\quad\ \ \mathsf{WORD}\ [aout_{31}; rt_0; rt_{30}; rt_{31}; rm_{31}; coutbar; nz\_top_{12}; z\_mid_4; nz\_bot_{16}])))))))))))))
\end{array}
$$

# 3 The goal

The goal of the verification is to prove that the two levels of the ALU formal model representing by the functions ALU and ALU_32BIT are functionally equivalent, i.e., both functions should deliver the same result for all possible combinations of inputs. This may be expressed in HOL as

$$\forall r\,t :: \mathsf{word32}.\ \forall c_{in}\,b\,alucon.\ \mathsf{ALU}\ r\,t\,c_{in}\,b\,alucon = \mathsf{ALU\_32BIT}\ r\,t\,c_{in}\,b\,alucon.$$

However, a goal of this form is unprovable since one of the condition bits has been arbitrarily assigned a return value of F which should really be *don't care* when the operation is bitwise. What needs to be proved is really that, *for all possible combinations of inputs, if the operation is not bitwise, these functions return identical results, otherwise, they return two results which are piecewise identical except for the* don't care *bit.*

Based on this analysis, a predicate named $\mathsf{IS\_ALU\_IMP}$ is defined. When applied to a function $f$ of the same type as $\mathsf{ALU}$, it returns $\mathsf{T}$ if and only if $f$ is an implementation of $\mathsf{ALU}$. Hence, the goal to be proved will be

$$?- \mathsf{IS\_ALU\_IMP}\ \mathsf{ALU\_32BIT}$$

The definition of this predicate and the details of the proof will be described in next section.

# 4 The proof

Since every ALU operation involves different combination of word and/or arithmetic operations and there are not too many operations, it is possible to carry out case analysis on the operations. Thus, for each ALU operation, i.e., for every possible value of the input *alucon* to the ALU functions, two expressions representing the operation result can be derived: one for the specification and one for the implementation. If these two expressions can be reduced to an identical form minus the field which has the *don't care* value, then the goal will be proved by combining all the cases.

Although every ALU operation is different, the underlying structure of the resulting expressions should be similar since they are derived from the same function definitions, namely the function $\mathsf{ALU}$ and $\mathsf{ALU\_32BIT}$. Therefore, it is possible to write programs to implement some proof procedures to carry out the proof mechanically.

The proof has been performed in three stages:

1. unfolding the definitions to derive two expressions for each ALU operation, such as

$$\begin{array}{rcl} \mathsf{ALU}\, r\, t\, c_{in}\, b\, op & = & E_{spec} \\ \mathsf{ALU\_32BIT}\, r\, t\, c_{in}\, b\, op & = & E_{imp} \end{array}$$

   where the expressions $E_{spec}$ and $E_{imp}$ on the right hand side of the equations consist of only basic word operations in the canonical set, bitwise word operations, word–natural number conversion and natural number addition '+';

2. manipulating $E_{spec}$ and $E_{imp}$ to unify them to an identical expression $E$ in cases of non-bitwise operations, or to derive two expressions $E'_{spec}$ and $E'_{imp}$ whose corresponding fields are identical except for the conditional bit `coutbar` in all bitwise cases;

3. proving the final goal by combining all cases.

The proof employs both forward and backward (goal-directed) proof techniques. The program implementing the proof procedure of Stage 1 is common to all thirteen cases. Details of this program will be described in Section 4.1.

In Stage 2, the ALU operations can be divided into two groups: the bitwise operations and non-bitwise operations. The latter consists of the following operations: alu_add, alu_sub, alu_inc and alu_sl. The last operation in this group is slightly different to the other three, it requires some special facts about the relation between addition and shift of binary words. In the bitwise group, operations can be further divided into four subgroups:

**binary** which consists of alu_and, alu_rmb, alu_nor and alu_xor;

**unary** which consists of alu_r, alu_m and alu_com;

**constant** which consists of a single operation alu_0;

**shift** which also consists of a single operation alu_sr.

As examples, three typical operations alu_and, alu_add and alu_sl will be described in detail in the following subsections. The result of this stage is a set of theories, one for each operation. For non-bitwise operations, there is only a single theorem stored in their theories. This theorem asserts the equivalence between the two ALU functions in the following form

$$\mathsf{ALU}\, r\, t\, c_{in}\, b\, op = \mathsf{ALU\_32BIT}\, r\, t\, c_{in}\, b\, op \tag{1}$$

where $op$ is one of the non-bitwise operations. For bitwise operations, two theorems will be derived and stored; one for each ALU function in the following form:

$$\begin{aligned} \mathsf{ALU}\, r\, t\, c_{in}\, b\, op &= E'_{spec} \qquad (2)\\ \mathsf{ALU\_32BIT}\, r\, t\, c_{in}\, b\, op &= E'_{imp} \qquad (3) \end{aligned}$$

These theories are then used in the last stage to derive the final theorem. The final result of the proof is stored in the theory `EQUIV` which will be described in Section 4.5. The complete theory hierarchy including the model definitions is shown in Figure 6.

## 4.1 The first stage

The task in the first stage is to unfold the ALU definitions to obtain the expressions $E_{spec}$ and $E_{imp}$. What needs to be done is to substitute instances of the lower level functions into the top level ALU functions ALU and ALU_32BIT. For the specification, the functions BITOP and FIND_SR are instantiated with a specific ALU operation $op$, and these are substituted into the ALU specification

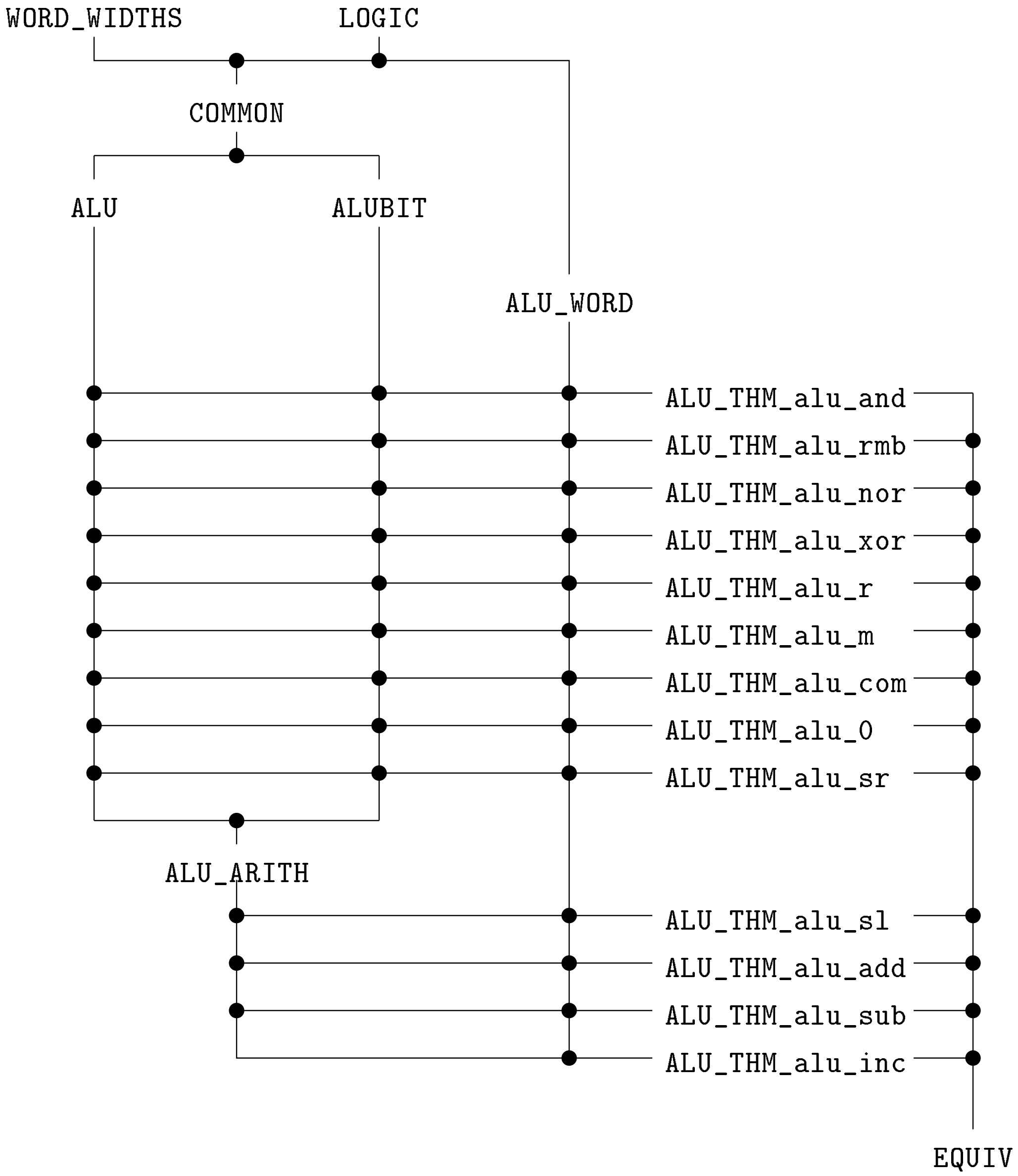


Figure 6: Theory hierarchy of the whole verification

ALU which is also instantiated with the same operation *op*. The manipulation is carried out by the function `ALU_CASE_RULE` which is listed in Section 4.1.1. The resulting theorem is bound to the ML identifier `ALU_SPEC`. The expression $E_{spec}$ is in the form

$$(w_{spec}, \mathsf{WORD}[B_8; B_7; B_6; B_5; B_4; B_3; B_2; B_1; B_0])$$

where $w_{spec}$ is the operation result and the $B_i$'s are the condition bits. Taking alu_and as an example,

$$\begin{aligned}
E_{spec}(\mathsf{alu_and}) \;=\; & (\mathsf{WNOT}\,(\mathsf{WNOT}\,rbar\;\mathsf{WAND}\,treg),\\
& \mathsf{WORD}\,[\mathsf{MSB}\,(\mathsf{WNOT}\,rbar\;\mathsf{WAND}\,treg);\\
& \quad \mathsf{LSB}\,(\mathsf{WNOT}\,rbar);\\
& \quad \mathsf{BIT}\,30\,(\mathsf{WNOT}\,rbar);\\
& \quad \mathsf{MSB}\,(\mathsf{WNOT}\,rbar);\\
& \quad \mathsf{F};\\
& \quad \mathsf{T};\\
& \quad \neg(\mathsf{BNVAL}\,(\mathsf{SEG}\,12\,20\,(\mathsf{WNOT}\,rbar\;\mathsf{WAND}\,treg)) = 0);\\
& \quad \mathsf{BNVAL}\,(\mathsf{SEG}\,4\,16\,(\mathsf{WNOT}\,rbar\;\mathsf{WAND}\,treg)) = 0;\\
& \quad \neg(\mathsf{BNVAL}\,(\mathsf{SEG}\,16\,0\,(\mathsf{WNOT}\,rbar\;\mathsf{WAND}\,treg)) = 0)])
\end{aligned}$$

For the implementation, the process of unfolding the definitions is more complicated. This is because there is much more detail. The process is carried out in three steps by three functions whose names are `ALU4BIT_CASES_RULE`, `ALU16BIT_CASES_RULE` and `ALU32BIT_CASES_RULE`. The first function produces an expression representing the output of a 4-bit ALU slice for the given operation *op*. This is used in the next step, in which the function `ALU16BIT_CASES_RULE` produces an expression representing the output of a 16-bit unit. This is in turn used in the final step by the function `ALU32BIT_CASES_RULE`. The theorem returned by this final step is bound to the ML identifier `ALU32BIT_THM`. The right hand side of this theorem is in the following form

$$(w_{imp}, \mathsf{WORD}[b_8; b_7; b_6; b_5; b_4; b_3; b_2; b_1; b_0])$$

which is similar to $E_{spec}$ in structure, but it is much more complicated. The main operation result $w_{imp}$ has the following tree-like structure in general:

$$\begin{aligned}
&\mathsf{WCAT}(\\
&\quad \mathsf{WCAT}(\\
&\quad\quad \mathsf{WCAT}((op\, r_{4,28}\, t_{4,28}), (op\, r_{4,24}\, t_{4,24})),\\
&\quad\quad \mathsf{WCAT}((op\, r_{4,20}\, t_{4,20}), (op\, r_{4,16}\, t_{4,16}))),\\
&\quad \mathsf{WCAT}(\\
&\quad\quad \mathsf{WCAT}((op\, r_{4,12}\, t_{4,12}), (op\, r_{4,8}\, t_{4,8})),\\
&\quad\quad \mathsf{WCAT}((op\, r_{4,4}\, t_{4,4}), (op\, r_{4,0}\, t_{4,0}))))
\end{aligned} \tag{4}$$

where $x_{i,j}$ is an $i$-bit segment of the word $x$ starting from $j$th bit and $op$ is a combination of word operations. Since the ALU is constructed in two tiers, the operand segments are actually expressed in a nested SEG term, for example, $r_{4,20}$ is SEG 4 4(SEG 16 16 $r$). It may be prefixed by a word negation operator WNOT. For the alu_and operation, $w_{imp}$(alu_and) is the following expression:

```
WCAT
 (WCAT
 (WCAT
   (WNOT
   (WNOT (SEG 4 12 (SEG 16 16 rbar)) WAND SEG 4 12 (SEG 16 16 treg)),
   WNOT
   (WNOT (SEG 4 8 (SEG 16 16 rbar)) WAND SEG 4 8 (SEG 16 16 treg))),
  WCAT
   (WNOT
   (WNOT (SEG 4 4 (SEG 16 16 rbar)) WAND SEG 4 4 (SEG 16 16 treg)),
   WNOT
   (WNOT (SEG 4 0 (SEG 16 16 rbar)) WAND SEG 4 0 (SEG 16 16 treg)))),
 WCAT
  (WCAT
   (WNOT
   (WNOT (SEG 4 12 (SEG 16 0 rbar)) WAND SEG 4 12 (SEG 16 0 treg)),
   WNOT
   (WNOT (SEG 4 8 (SEG 16 0 rbar)) WAND SEG 4 8 (SEG 16 0 treg))),
  WCAT
   (WNOT
   (WNOT (SEG 4 4 (SEG 16 0 rbar)) WAND SEG 4 4 (SEG 16 0 treg)),
   WNOT
   (WNOT (SEG 4 0 (SEG 16 0 rbar)) WAND SEG 4 0 (SEG 16 0 treg)))))
```

Similarly, many condition bits $b_i$ are also involved with very large expressions.

The ML functions implementing the proof procedures described in this section are common to all operations, they are stored in the file `alu_funs.m`. This file is loaded when the proof of each operation is performed. The subsections below are the contents of this file.

#### 4.1.1 Function for unfolding the ALU specification

The function `ALU_CASES_RULE` unfolds the ALU specification for each operation. It takes a list of theorems and a string $op$ as its arguments. The string should be the name of one of the ALU operations. The theorems in the list are used in rewriting. It returns a theorem in the form:

$$\begin{array}{l}\mathsf{word32}\ rbar, \mathsf{word32}\ treg \\ \vdash \mathsf{ALU}\ rbar\ treg\ cin\ bflag\ op = E_{spec}\end{array}$$

where $E_{spec}$ consists of only basic word operations and possibly the function ADD32 in arithmetic operations.

```
let ALU_CASES_RULE =
   let get_defs = map (definition `COMMON`) [`GET_AOUT_DEF`;`GET_COUT_DEF`] in
   let defthm = ALU_THM1 in
   let ALUCON_RULE =  \opthms th.
     (CONV_RULE (DEPTH_CONV COND_CONV)
     (PURE_ONCE_REWRITE_RULE [REFL_CLAUSE] (SUBS opthms th))) in
   let notths = tl (CONJUNCTS NOT_CLAUSES) in
   let opconst  =
       let lst = CONJUNCTS (theorem `COMMON` `alu_op_const_dist`) in
       (map (\t. MATCH_MP NOT_F t) (lst @ (map NOT_EQ_SYM lst))) in
 \thms op.
  (let tm = mk_const(op, ":alu_op") in
   let opthms = filter (\t. ((lhs o lhs) (concl t)) = tm) opconst in
   let bthm = ALUCON_RULE opthms(SPEC tm BITOP_THM1) in
   let rmthm = RIGHT_CONV_RULE (TOP_DEPTH_CONV COND_CONV)
     (PURE_ONCE_REWRITE_RULE [REFL_CLAUSE]
 (SUBS opthms (SPECL ["MSB(treg:(bool)word)";tm]
 (definition `COMMON` `GET_RM31_DEF`)))) in
   (PURE_REWRITE_RULE (rmthm . thms)
   (repeatl 8 (RIGHT_CONV_RULE let_CONV)
   (PURE_ONCE_REWRITE_RULE get_defs
   (RIGHT_CONV_RULE ((RAND_CONV (REWR_CONV bthm)) THENC let_CONV)
    (SPEC tm ALU_THM1))))));;
```

#### 4.1.2 Function for unfolding the 4-bit ALU slice

The function `ALU4BIT_CASES_RULE` unfolds the 4-bit ALU implementation for each operation. It takes two arguments: the first is a list of theorems to be used in rewriting, and the second is a string specifying the operation.

```
let ALU4BIT_CASES_RULE =
   let ALUCON_RULE =  \opthms th.
     (CONV_RULE (DEPTH_CONV COND_CONV)
     (PURE_ONCE_REWRITE_RULE [REFL_CLAUSE] (SUBS opthms th))) in
   let def_thm =  (GEN_RR_RULE (definition `ALUBIT` `ALU_4BIT_DEF`)) in
   let notths = tl (CONJUNCTS NOT_CLAUSES) in
   let opconst =
       let lst = CONJUNCTS (theorem `COMMON` `alu_op_const_dist`) in
       (map (\t. MATCH_MP NOT_F t) (lst @ (map NOT_EQ_SYM lst))) in
 \ths op.  (
   let tm = mk_const(op, ":alu_op") in
   let opthms = filter (\t. ((lhs o lhs) (concl t)) = tm) opconst in
   let rmth = GEN_ALL(ALUCON_RULE opthms (SPECL ["t31:bool";tm]
      (definition `COMMON` `GET_RM31_DEF`))) in
   let bthm = ALUCON_RULE opthms(SPEC tm FOURBITOP_THM1) in
```

```
let wthm = GEN_ALL (SYM (AP_THM
(definition `WORD_WIDTHS` `word4_def`)"w:(bool)word")) in
let thm1 = UNDISCH_ALL (SPEC "r4bar:(bool)word"
   (PURE_ONCE_REWRITE_RULE[wthm](RESQ_HALF_SPEC
    (SPEC "4" (MATCH_MP (RESQ_HALF_SPEC PBITOP_PWORDLEN) PBITOP_WNOT))))) in
(GEN_ALL (RESQ_GEN_ALL (PURE_REWRITE_RULE (rmth . ths) (LET_LAM_RULE
(RIGHT_CONV_RULE (RAND_CONV (RESQ_REWRITE1_CONV [thm1] bthm))
(RIGHT_CONV_RULE let_CONV
   (SPEC_ALL(RESQ_SPEC_ALL(SPEC tm def_thm)))))))));;
```

**4.1.3 Function for unfolding the 16-bit ALU definition** This function unfolds the 16-bit ALU implementation for each operation. It takes three arguments: the first is a theorem returned by the function ALU4BIT_CASES_RULE of the same operation; the second is a list of theorems to be used in rewriting, and the last is a string specifying the operation.

```
let ALU16BIT_CASES_RULE =
    let defthm = GEN_ROR_BLOCK_RULE 4 (definition `ALUBIT` `ALU_16BIT_DEF`) in
  \th4 thms op.
   (let tm = mk_const(op, ":alu_op") in
    let thm1 = (SPECL (replicate tm 4)  defthm) in
    let thm2 = (repeatl 7 (RIGHT_CONV_RULE let_CONV)
       (SPEC_ALL(RESQ_SPEC_ALL thm1))) in
    let unfold4 = \k th.
    (repeatl 2 (RIGHT_CONV_RULE let_CONV)
      (PURE_REWRITE_RULE [FST;SND] (RIGHT_CONV_RULE let_CONV
       (RIGHT_CONV_RULE (RAND_CONV
    (RESQ_REWRITE1_CONV
       (map (SEG_PWL_CONV "4" k "16")
        ["r16bar:(bool)word"; "t16reg:(bool)word"]) th4)) th)))) in
    GEN_ALL(RESQ_GEN_ALL(PURE_REWRITE_RULE thms
    (itlist unfold4 ["12"; "8"; "4"; "0"] thm2))));;
```

**4.1.4 Function for unfolding the 32-bit ALU implementation** This function unfolds the 32-bit ALU implementation for each operation. It takes three arguments: the first is a theorem returned by the function ALU16BIT_CASES_RULE of the same operation; the second is a list of theorems to be used in rewriting, and the last is a string specifying the operation.

```
let ALU32BIT_CASES_RULE =
  \thm16 thms op.
    let OP = mk_const(op, ":alu_op") in
    let unfold16 = \k th.
  (repeatl 2 (RIGHT_CONV_RULE let_CONV)
     (RIGHT_CONV_RULE (RAND_CONV (RESQ_REWRITE1_CONV
        (map(SEG_PWL_CONV "16" k "32")
      ["rbar:(bool)word"; "treg:(bool)word"]) thm16)) th)) in
    (PURE_REWRITE_RULE (thms)
    (RIGHT_CONV_RULE (let_CONV THENC (DEPTH_CONV BIT_CONV))
      (PURE_ONCE_REWRITE_RULE zero_thms
    (PURE_ONCE_REWRITE_RULE[(definition `ALUBIT` `ZERO_GATES_DEF`)]
     (repeatl 2 (RIGHT_CONV_RULE let_CONV)
```

```
(itlist unfold16 ["16"; "0"] (SPEC OP ALU_32BIT_THM1)))))));;
```

## 4.2 Stage 2: bitwise operations

This section describes Stage 2 of the the proof for bitwise operations. As a typical operation of this group, the results of the bitwise AND operation `alu_and` is shown as an example. At the end of Stage 1, as described in Section 4.1, two theorems stating the operation results have been derived. The expression $E_{imp}$ representing the result at the implementation level is very large. The aim of this stage is to simplify this expression using the properties of word operations so that each corresponding fields between $E_{imp}$ and $E_{spec}$ will be equivalent, i.e., $w_{imp}$ becomes $w_{spec}$ and $b_i$ becomes $B_i$ for $i = 0, 1, 2, 4, 5, 6, 7, 8$. In essence, the difference between $E_{spec}$ and $E_{imp}$ is that the bitwise operators are applied to their 32-bit operands directly in the former expression while the same combination of operators are applied to a number of 4-bit segments and the results are concatenated to form a 32-bit word in the latter. The strategy for proving the equivalence of these two methods of performing the same operations relies on the basic properties of word operations. The `word` library provides all the required theorems which assert these general properties of words and operations on words.

The strategy for unifying these two expressions is common to all bitwise operations. It is implemented as an ML function `simp_bitop` which will be listed in Section 4.2.1, but before looking at the details of the function, the strategy is explained first. This can be divided into five steps:

1. Reduce the nested SEG introduced by the two tier structure of the ALU implementation using the theorem `SEG_SEG`.

   **HOL Theorem** (`SEG_SEG`)

   $$\begin{array}{l} \vdash \forall n.\, \forall w :: \mathsf{PWORDLEN}\, n.\, \forall m_1\, k_1\, m_2\, k_2. \\ \quad k_1 + m_1 \leq n \wedge k_2 + m_2 \leq m_1 \supset \\ \quad\quad (\mathsf{SEG}\, m_2\, k_2\, (\mathsf{SEG}\, m_1\, k_1\, w) = \mathsf{SEG}\, m_2\, (k_1 + k_2)\, w) \end{array}$$

2. Move the bitwise operators, such as WNOT, WAND and so on, inside SEG. This is possible because they all satisfy the predicate PBITOP or PBITBOP and have the property stated in the theorems `PBITOP_SEG` and `PBITBOP_SEG`.

   **HOL Theorem** (`PBITOP_SEG`)

   $$\begin{array}{l} \vdash \forall op :: \mathsf{PBITOP}. \\ \quad \forall n.\, \forall w :: \mathsf{PWORDLEN}\, n.\, \forall k\, m. \\ \quad\quad k + m \leq n \supset (op\, (\mathsf{SEG}\, m\, k\, w) = \mathsf{SEG}\, m\, k\, (op\, w)) \end{array}$$

**HOL Theorem** (PBITBOP_SEG)

$$\vdash \forall op :: \mathsf{PBITBOP}.\ \forall n.\ \forall w_1\, w_2 :: \mathsf{PWORDLEN}\, n.\ \forall k\, m.\ k + m \leq n \supset (op\,(\mathsf{SEG}\, m\, k\, w_1)\,(\mathsf{SEG}\, m\, k\, w_2) = \mathsf{SEG}\, m\, k\,(op\, w_1\, w_2))$$

3. Remove WCAT at three nested levels using the theorem WCAT_SEG_SEG.

**HOL Theorem** (WCAT_SEG_SEG)

$$\vdash \forall n.\ \forall w :: \mathsf{PWORDLEN}\, n.\ \forall k\, m_1\, m_2.\ k + m_1 + m_2 \leq n \supset (\mathsf{WCAT}\,(\mathsf{SEG}\, m_2\,(k + m_1)\, w, \mathsf{SEG}\, m_1\, k\, w) = \mathsf{SEG}\,(m_1 + m_2)\, k\, w)$$

4. The result of the above three steps is that the expression representing the main operation result $w_{imp}$ becomes

$$\mathsf{SEG}\, 32\, 0\, w$$

where $w$ involves only bitwise operators and the operands. More specifically, the expression for alu_and operation is

$$\mathsf{SEG}\, 32\, 0\, \mathsf{WNOT}\,(\mathsf{WNOT}\, rbar\ \mathsf{WAND}\ treg).$$

According to the theorem SEG_WORD_LENGTH, this can be reduced to $w$ which should be identical to $w_{spec}$.

**HOL Theorem** (SEG_WORD_LENGTH)

$$\vdash \forall n.\ \forall w :: \mathsf{PWORDLEN}\, n.\ \mathsf{SEG}\, n\, 0\, w = w$$

As the result of this step, $w_{imp}$ has been reduced to equal $w_{spec}$. At the same time, the expressions representing the conditional bits $b_i$'s have also been simplified to certain extent. The remaining steps work only on the condition bits.

5. use the theorem BIT_SEG to simplify the condition bits aout31, r0, r30 and r31.

**HOL Theorem** (BIT_SEG)

$$\vdash \forall n.\ \forall w :: \mathsf{PWORDLEN}\, n.\ \forall j\, k\, m.\ k + m \leq n \supset (j < m \supset (\mathsf{BIT}\, j\,(\mathsf{SEG}\, m\, k\, w) = \mathsf{BIT}\,(j + k)\, w))$$

The expression representing aout31 for alu_and before this step is

$$\mathsf{BIT}\,0(\mathsf{SEG}\,4\,0\,w_{spec}(\mathsf{alu\_and}))$$

which will be simplified to $\mathsf{BIT}\,0\,w_{spec}(\mathsf{alu\_and})$.

6. The remaining condition bits, namely nztop12, zmid4 and nzbot16, indicates the zero condition of the operation result. On the specification level, they are expressed in terms of equality test between the value of the appropriate segments of the result and 0(zero). The expressions for alu_and are:

$$\begin{aligned} B_0 &= \neg(\mathsf{BNVAL}(\mathsf{SEG}\,16\,0\,((\mathsf{WNOT}\,rbar)\ \mathsf{WAND}\,treg)) = 0) \\ B_1 &= \mathsf{BNVAL}(\mathsf{SEG}\,4\,16\,((\mathsf{WNOT}\,rbar)\ \mathsf{WAND}\,treg)) = 0) \\ B_2 &= \neg(\mathsf{BNVAL}(\mathsf{SEG}\,12\,20\,((\mathsf{WNOT}\,rbar)\ \mathsf{WAND}\,treg)) = 0) \end{aligned}$$

Whilst on the implementation level, they are expressed in terms of equality test between segments of the results and a zero valued word $\mathsf{NBWORD}\,4\,0$. The expression for alu_and are:

$$\begin{aligned} b_0 &= \neg(\mathsf{WORD}[\mathsf{SEG}\,4\,12\,w = \mathsf{NBWORD}\,4\,0; \\ &\qquad \mathsf{SEG}\,4\,8\,w = \mathsf{NBWORD}\,4\,0; \\ &\qquad \mathsf{SEG}\,4\,4\,w = \mathsf{NBWORD}\,4\,0; \\ &\qquad \mathsf{SEG}\,4\,0\,w = \mathsf{NBWORD}\,4\,0] = \mathsf{WORD}[\mathsf{T};\mathsf{T};\mathsf{T};\mathsf{T}]) \\ b_1 &= \mathsf{SEG}\,4\,16\,w = \mathsf{NBWORD}\,4\,0 \\ b_2 &= \neg(\mathsf{WORD}[\mathsf{SEG}\,4\,28\,w = \mathsf{NBWORD}\,4\,0; \\ &\qquad \mathsf{SEG}\,4\,24\,w = \mathsf{NBWORD}\,4\,0; \\ &\qquad \mathsf{SEG}\,4\,20\,w = \mathsf{NBWORD}\,4\,0] = \mathsf{WORD}[\mathsf{T};\mathsf{T};\mathsf{T}]) \end{aligned}$$

where $w$ is the expression $((\mathsf{WNOT}\,rbar)\ \mathsf{WAND}\,treg)$. The general theorems required to unify the corresponding bits between the two levels are ZERO_WORD_VAL and EQ_NBWORD0_SPLIT.

**HOL Theorem** (ZERO_WORD_VAL)

$$\begin{aligned} &\vdash \forall n.\,\forall w :: \mathsf{PWORDLEN}\,n. \\ &\qquad (w = \mathsf{NBWORD}\,n\,0) = (\mathsf{BNVAL}\,w = 0) \end{aligned}$$

**HOL Theorem** (EQ_NBWORD0_SPLIT)

$$\begin{aligned} &\vdash \forall n.\,\forall w :: \mathsf{PWORDLEN}\,n. \\ &\quad \forall m.\, m \leq n \supset \\ &\qquad ((w = \mathsf{NBWORD}\,n\,0) = \\ &\qquad\quad (\mathsf{SEG}\,(n-m)\,m\,w = \mathsf{NBWORD}\,(n-m)\,0) \wedge \\ &\qquad\quad (\mathsf{SEG}\,m\,0\,w = \mathsf{NBWORD}\,m\,0)) \end{aligned}$$

The first theorem states that an equality test between an $n$-bit word and a zero valued word of the same size is equal to a test between the value of the word and zero. The second theorem states that an equality test between an $n$-bit word and a zero valued word of the same size can be split into two parts. Since the indexes in the expressions $b_i$'s and $B_i$'s are all constants, three theorems matching these expressions are obtained by instantiating the general theorems. They are stored in the theory `ALU_WORD` so they can be used in all proofs for bitwise operations.

The proof script for the bitwise group is organized into several files. ML functions implementing proof procedures common to all bitwise operations and theorems required for all bitwise operations are stored in the file `alu_bitop_funs.m`. There is a separate file, the driver file, for each operation. They are named by prefixing the operation name to '`_thm.m`'. For instance, the file for the `alu_and` operation is `alu_and_thm.m`. These driver files contain function calls and instances of theorems specific to each operation. The first subsection below shows the function `simp_bitop` in the file `alu_bitop_funs.m`. The remaining subsections lists the file `alu_and_thm.m`.

**4.2.1 The function `simp_bitop`** This is the function for simplifying the result of the bitwise operations. It takes three arguments: the first $thm1$ should be the theorem derived from unfolding the ALU implementation, i.e., `ALU_32BIT_THM`; the second $sthms$ is a list of theorems to be substituted into $thm1$ to simplify the latter, the last argument $tm$ is the expression representing the main result specified in the `ALU_SPEC`, i.e., the term $w_{spec}$ which is the target of the simplification.

```
let simp_binop =
   let eqT = GEN_ALL (el 2 (CONJ_LIST 4 (SPEC_ALL EQ_CLAUSES))) in
   let cond1_lem = GEN_ALL (DISCH_ALL (PURE_REWRITE_RULE[IMP_CLAUSES]
      (CONV_RULE ((ONCE_DEPTH_CONV ADD_CONV) THENC
   (LHS_CONV LESS_EQ_CONV) THENC ((RAND_CONV o LHS_CONV) LESS_CONV))
       (GQSPECL ["32"; "w:bool word";"3";"28";"4"] BIT_SEG)) )) in
   let mk_wcat_seg_seg th ts = MP
      (CONV_RULE ((ONCE_DEPTH_CONV ADD_CONV) THENC
      (ONCE_DEPTH_CONV ADD_CONV) THENC (LHS_CONV LESS_EQ_CONV))
      (SPECL ts th)) TRUTH in
  \thm1 sthms tm.
   let thm3 = PWORDLEN_bitop_CONV ("PWORDLEN 32 ^tm") in
   let thm31 = RESQ_MATCH_MP (SPEC "32" WCAT_SEG_SEG) thm3 in
   let sub31 = map (mk_wcat_seg_seg thm31)
   [ ["24"; "4"; "4"]; ["16"; "4"; "4"];
     ["8" ; "4"; "4"]; ["0";  "4"; "4"]] in
   let sub32 = map (mk_wcat_seg_seg thm31)
   [ ["16"; "8"; "8"]; ["0"; "8"; "8"]] in
   let sub33 =  map (mk_wcat_seg_seg thm31) [["0"; "16"; "16"]] in
   let sub4 = [RESQ_MATCH_MP (SPEC "32" SEG_WORD_LENGTH) thm3] in
   let subl = [sub4; sub33;sub32;sub31] @ sthms @ [(seg_seg_r @ seg_seg_t)] in
   let tm' = (snd (dest_comb tm)) in
```

```
let lem5 = PWORDLEN_bitop_CONV "PWORDLEN 32 ^tm'" in
let cond1_lem2 = MP (SPEC tm' cond1_lem) lem5 in
let thm6 = PURE_REWRITE_RULE[WORD_11;CONS_11;eqT;REFL_CLAUSE;AND_CLAUSES]
 (SUBS (cond1_lem2 . cond_lems2) (itlist SUBS subl thm1)) in
itlist PROVE_HYP (lem5 . operands)
(RIGHT_CONV_RULE (EVERY_CONV
 (map (\th. RESQ_REWRITE1_CONV [] (GSYM (theorem `ALU_WORD` th)))
  [`COND_NZTOP12`;`COND_ZMID4`;`COND_NZBOT16`])) thm6);;
```

**4.2.2 Define the current operation** The name of the current operation, as a string, is bound to the identifier `alu_op`. It is used in all calls to proof functions.

```
letref alu_op = `alu_and`;;
```

**4.2.3 Stage 1:specification** The function `ALU_CASES_RULE` is called to unfold the specification.

```
let ALU_SPEC =
   let rwthms = [NOT_CLAUSES; NOT_THM1] in
   ALU_CASES_RULE rwthms alu_op;;
```

**4.2.4 Stage 1:implementation** The functions for unfolding the implementation are called here.

```
let ALU32BIT_THM =
   let rwthms = (NOT_CLAUSES . LOGIC_THMS) in
   let rwthms2 = [FST;SND] in
   let ALU4BIT_THM =
      ALU4BIT_CASES_RULE rwthms2 alu_op in
   let ALU16BIT_THM =
      GEN_ALL(RESQ_GEN_ALL (PURE_REWRITE_RULE rwthms2
       (RESQ_SPEC_ALL(SPEC_ALL
     (ALU16BIT_CASES_RULE ALU4BIT_THM
       (LOOKAHEAD_4_THMF . rwthms2) alu_op))))) in
   ALU32BIT_CASES_RULE ALU16BIT_THM rwthms alu_op;;
```

**4.2.5 Simplify the theorems—stage 2** After stage 1, we have obtained two theorems stating the operation results of the two ALU models: the specification — `ALU_SPEC` and the implementation `ALU32BIT_THM` (See Section 4.1). We can now work on these theorems to reduce their right hand sides to the same expression by simplifying the implementation theorem and/or expanding the specification theorem.

**4.2.6** Terms and theorems specific to each operation are bound to some ML identifiers so that they can be passed to the proof functions. The identifier `tm` is bound to the term representing the main result of the operation as given on the specificaation level. This is the target toward which the implementation is simplified. `rands` is bound to a list of terms whose elements are the operands or the negation of the operands depending on the operation. `operands` is bound to a list of theorems which asserts that the size of operands ia 32 bits.

```
let tm = (fst o dest_pair o snd o dest_eq)(concl ALU_SPEC);;

%  ["WNOT (rbar:(bool)word)";" (treg:(bool)word)"] %

let rands = (snd o strip_comb o snd)(dest_comb tm);;

let operands =
   map (\t. PWORDLEN_bitop_CONV "PWORDLEN 32 ^t") rands;;
```

**4.2.7 The implementation theorem** Here, we instantiate some theorems to match specific operations and pass them to the proof function `simp_bitop`. The theorem returned by the function is `alu_imp_thm`.

**HOL Theorem** (`alu_imp_thm`)

$$
\begin{array}{l}
\mathsf{PWORDLEN}\,32\,rbar, \mathsf{PWORDLEN}\,32\,treg \\
\quad \vdash \mathsf{ALU\_32BIT}\,rbar\,treg\,cin\,bflag\,\mathsf{alu\_and} = \\
\quad\quad \mathsf{WNOT}\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg), \\
\quad\quad \mathsf{WORD}\,[\mathsf{BIT}\,31\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg); \\
\quad\quad\quad \mathsf{BIT}\,0\,(\mathsf{WNOT}\,rbar); \\
\quad\quad\quad \mathsf{BIT}\,30\,(\mathsf{WNOT}\,rbar); \\
\quad\quad\quad \mathsf{BIT}\,31\,(\mathsf{WNOT}\,rbar); \\
\quad\quad\quad \mathsf{F}; \\
\quad\quad\quad \neg cin; \\
\quad\quad\quad \neg(\mathsf{BNVAL}\,(\mathsf{SEG}\,12\,20\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg)) = 0); \\
\quad\quad\quad \mathsf{BNVAL}\,(\mathsf{SEG}\,4\,16\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg)) = 0; \\
\quad\quad\quad \neg(\mathsf{BNVAL}\,(\mathsf{SEG}\,16\,0\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg)) = 0)]
\end{array}
$$

```
let alu_imp_thm =
   let thms2 =
      let lem3 = itlist PROVE_HYP operands (RESQ_SPECL rands SEG_WAND32) in
```

```
      mk_SEG_thms lem3 in
   let thms3 =
      let lem3 = PWORDLEN_bitop_CONV
       (mk_comb("(PWORDLEN:num->bool word->bool) 32",(snd(dest_comb tm)))) in
   mk_SEG_thms (RESQ_MATCH_MP SEG_WNOT32 lem3) in
   let sthms = [thms3; thms2; wnot_seg_thms_r] in
   let alu32bit_thm = itlist PROVE_HYP word32_thms ALU32BIT_THM in
   (simp_binop alu32bit_thm sthms tm);;
```

**4.2.8 The specification theorem** Since the result of unfolding the specification of this operation is already quite simple, what remains to be done is to convert a few basic word operations to the canonical form.

**HOL Theorem** (alu_spec_thm)

$$
\begin{array}{l}
\mathsf{PWORDLEN}\,32\,rbar, \mathsf{PWORDLEN}\,32\,treg \\
\quad \vdash \mathsf{ALU}\,rbar\,treg\,cin\,bflag\,\mathsf{alu\_and} = \\
\quad\quad \mathsf{WNOT}\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg), \\
\quad\quad \mathsf{WORD}\,[\mathsf{BIT}\,31\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg); \\
\quad\quad\quad \mathsf{BIT}\,0\,(\mathsf{WNOT}\,rbar); \\
\quad\quad\quad \mathsf{BIT}\,30\,(\mathsf{WNOT}\,rbar); \\
\quad\quad\quad \mathsf{BIT}\,31\,(\mathsf{WNOT}\,rbar); \\
\quad\quad\quad \mathsf{F}; \\
\quad\quad\quad \mathsf{T}; \\
\quad\quad\quad \neg(\mathsf{BNVAL}\,(\mathsf{SEG}\,12\,20\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg)) = 0); \\
\quad\quad\quad \mathsf{BNVAL}\,(\mathsf{SEG}\,4\,16\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg)) = 0; \\
\quad\quad\quad \neg(\mathsf{BNVAL}\,(\mathsf{SEG}\,16\,0\,(\mathsf{WNOT}\,rbar\,\mathsf{WAND}\,treg)) = 0)]
\end{array}
$$

```
let alu_spec_thm =
   let alu_spec = itlist PROVE_HYP word32_thms ALU_SPEC in
   HYP_CONV_RULE PWORDLEN_bitop_CONV
    (CONV_RULE (EVERY_CONV
     (map (\th. RESQ_REWRITE1_CONV [] (theorem `ALU_WORD` th))
    [`COND_R00`; `COND_AOUT31`])) alu_spec);;
```

**4.2.9 Equivalence theorems** Unlike for non-bitwise operations, we cannot prove an equivalence theorem for bitwise operations since one of the condition bits is undefined, so we save the theorems derived above. They can be used in the final equivalence proof.

```
map save_thm
   [(`alu_spec_thm`, alu_spec_thm); (`alu_imp_thm`, alu_imp_thm)];;
```

## 4.3 Stage 2: arithmetic operations

The procedures in Stage 2 for the arithmetic operations alu_add, alu_sub and alu_inc are similar to each other. The operation alu_add is described in detail as an example. The proof for these operations calls for more facts about words and addition. Some of the more general facts are supplied as theorems by the `word` library. Others are more specific to the ALU. They are proved and stored in the theory `ALU_ARITH`.

After unfolding the definition, two expressions representing the main operation results at the two levels, namely $w_{spec}(\mathsf{alu\_add})$ and $w_{imp}(\mathsf{alu\_add})$ are obtained. They are unified to an expression $W$ by both expanding $w_{spec}(\mathsf{alu\_add})$ and simplifying $w_{imp}(\mathsf{alu\_add})$.

The result at the specification level is

$$w_{spec}(\mathsf{alu\_add}) \quad = \quad \mathsf{WNOT}\,(\mathsf{SEG}\,32\,0\,(\mathsf{ADD32}\,(\mathsf{WNOT}\,rbar)\,treg\,cin))$$

Using the theorem `ADD32_THM`, this can be rewritten as

$$w_{spec}(\mathsf{alu\_add}) \quad = \quad \mathsf{WNOT}(\mathsf{NBWORD}\,32\,(\mathsf{BNVAL}\,rbar + \mathsf{BNVAL}\,treg + \mathsf{BV}\,cin))$$

which specifies that the main result of alu_add is obtained by converting the operands into natural numbers, performing the addition and converting the sum back to a word. The expression $w_{imp}(\mathsf{alu\_add})$ is very large and complicated. Nevertheless, it still has the same structure as shown in equation (4) on page 31. The expression $op\,r_{i,j}\,t_{i,j}$ representing the operation performed by each slice has an extra value $c_j$: the carry input from the next less significant slice. The operation $op$ is the function ADDER4. For instance, the expression for the second least significant slice is

$$
\begin{array}{l}
op\,r_{4,4}\,t_{4,4}\,c_4 = \\
\quad \mathsf{WNOT} \\
\quad (\mathsf{ADDER4} \\
\quad\quad (\mathsf{WNOT}\,(\mathsf{SEG}\,4\,4\,(\mathsf{SEG}\,16\,0\,rbar))) \\
\quad\quad (\mathsf{SEG}\,4\,4\,(\mathsf{SEG}\,16\,0\,treg)) \\
\quad\quad (\mathsf{LOOKAHEAD\_1}\,(\neg cin) \\
\quad\quad\quad (\mathsf{PG4}\,(\mathsf{WNOT}\,(\mathsf{SEG}\,4\,0\,(\mathsf{SEG}\,16\,0\,rbar)))\,(\mathsf{SEG}\,4\,0\,(\mathsf{SEG}\,16\,0\,treg)))))
\end{array}
$$

The carry input $c_4$ is generated by the carry look ahead function, which is expressed in terms of the function PG4. In summary, the implementation performs addition by adding the corresponding segments of the operands and taking care of the carry propagation.

In essence, the task of unifying the expressions denoting the main results of the specification and the implementation, $w_{spec}(\mathsf{alu\_add})$ and $w_{imp}(\mathsf{alu\_add})$ is to prove the two methods of performing addition give the same result.

The first two steps are exactly the same as those for bitwise operations described in Section 4.2. Step 3 will be to convert the carry inputs $c_j$ into an expression in terms of the carry function $\mathsf{ICARRY}$. This function has been defined in the `word` library.

**HOL Definition** (`ICARRY_DEF`)

$$\begin{array}{l}\vdash (\forall w_1\, w_2\, cin.\\ \quad \mathsf{ICARRY}\, 0\, w_1\, w_2\, cin = cin) \wedge \\ \quad (\forall n\, w_1\, w_2\, cin.\\ \quad\quad \mathsf{ICARRY}\,(\mathsf{SUC}\, n)\, w_1\, w_2\, cin = \\ \quad\quad\quad \mathsf{BIT}\, n\, w_1 \wedge \mathsf{BIT}\, n\, w_2 \vee \\ \quad\quad\quad (\mathsf{BIT}\, n\, w_1 \vee \mathsf{BIT}\, n\, w_2) \wedge \mathsf{ICARRY}\, n\, w_1\, w_2\, cin)\end{array}$$

$\mathsf{ICARRY}\, j\, r\, t\, c_{in}$ evaluates to the carry input to the $j$-th bit for any $j$ less than the size of the operand. The next step, step 4, moves the bitwise NOT operator $\mathsf{WNOT}$ out of the nested $\mathsf{WCAT}$s. Step 5 is to unfold with the definition of $\mathsf{ADDER4}$. After these steps, the expression $op\, r_{4,4}\, t_{4,4}\, c_4$ will be simplified to

$$\begin{array}{l}\mathsf{NBWORD}\, 4\, (\mathsf{BNVAL}\, (\mathsf{SEG}\, 4\, 4\, (\mathsf{WNOT}\, rbar)) + \mathsf{BNVAL}\, (\mathsf{SEG}\, 4\, 4\, treg) + \\ \quad\quad \mathsf{BV}\, (\mathsf{ICARRY}\, 4\, (\mathsf{WNOT}\, rbar)\, treg\, cin)).\end{array}$$

Instead of simplifying this further, the expression representing the specification can be expanded towards this. Using the theorem `ADD_WORD_SPLIT` recursively, the expression $w_{spec}(\mathsf{alu\_add})$ can be expanded to be the same as the implementation.

**HOL Theorem** (`ADD_WORD_SPLIT`)

$$\begin{array}{l}\vdash \forall n_1\, n_2.\, \forall w_1\, w_2 :: \mathsf{PWORDLEN}\, (n_1 + n_2).\, \forall cin.\\ \quad \mathsf{NBWORD}\, (n_1 + n_2)\, (\mathsf{BNVAL}\, w_1 + \mathsf{BNVAL}\, w_2 + \mathsf{BV}\, cin) = \\ \quad\quad \mathsf{WCAT} \\ \quad\quad (\mathsf{NBWORD}\, n_1 \\ \quad\quad\quad (\mathsf{BNVAL}\, (\mathsf{SEG}\, n_1\, n_2\, w_1) + \mathsf{BNVAL}\, (\mathsf{SEG}\, n_1\, n_2\, w_2) + \\ \quad\quad\quad \mathsf{BV}\, (\mathsf{ACARRY}\, n_2\, w_1\, w_2\, cin)), \\ \quad\quad \mathsf{NBWORD}\, n_2 \\ \quad\quad\quad (\mathsf{BNVAL}\, (\mathsf{SEG}\, n_2\, 0\, w_1) + \mathsf{BNVAL}\, (\mathsf{SEG}\, n_2\, 0\, w_2) + \mathsf{BV}\, cin))\end{array}$$

The results of this manipulation are two theorems, namely `alu_spec_thm` and `alu_imp_thm`.

Unlike the bitwise operations, the arithmetic operations do not contain *don't care* value in their results. A theorem equating the two ALU functions in the form of equation (1) can be derived. Goal-directed proof is used in this derivation. The following subsections list the contents of the driver file for the addition operation (`alu_add_thm.m`).

**4.3.1 Define the current operation** The name of the current operation, as a string, is bound to the identifier `alu_op`. It is used in all calls to proof functions.

```
letref alu_op = `alu_add`;;
```

**4.3.2 Stage 1:specification** The function `ALU_CASES_RULE` is called to unfold the specification.

```
let ALU_SPEC =
   let rwthms = [NOT_CLAUSES; NOT_THM1] in
   ALU_CASES_RULE rwthms alu_op;;
```

**4.3.3 Stage 1:implementation** The functions for unfolding the implementation are called here.

```
let ALU32BIT_THM =
   let rwthms = (NOT_CLAUSES . LOGIC_THMS) in
   let ALU4BIT_THM =
      ALU4BIT_CASES_RULE [ADD4BIT_DEF;FST;SND] alu_op in
   let ALU16BIT_THM =
      GEN_ALL(RESQ_GEN_ALL (PURE_REWRITE_RULE(thm_add_bits @ [FST;SND])
       (RESQ_SPEC_ALL(SPEC_ALL
     (ALU16BIT_CASES_RULE ALU4BIT_THM
       [FST;SND;LOOKAHEAD_4_THMF] alu_op))))) in
   ALU32BIT_CASES_RULE ALU16BIT_THM rwthms alu_op;;
```

**4.3.4 Simplify the theorems—stage 2** After stage 1, we have obtained two theorems stating the operation results of the two ALU models: the specification — `ALU_SPEC` and the implementation `ALU32BIT_THM` (See Section 4.1). We can now work on these theorems to reduce their right hand sides to the same expression by simplifying the implementation theorem and/or expanding the specification theorem.

**4.3.5** Terms and theorems specific to each operation are bound to some ML identifiers so that they can be passed to the proof functions. The identifier `operands` is bound to a list of theorems which asserts that the size of operands are 32-bit.

```
let operands = [pw_wnot_rbar; pw_treg];;
```

**4.3.6** We also need a list of theorems in the following form:

$$[[\mathsf{PWORDLEN}\,32\,r \vdash \mathsf{PWORDLEN}\,4(\mathsf{SEG}\,4\,k\,r); \\ [\mathsf{PWORDLEN}\,32\,t \vdash \mathsf{PWORDLEN}\,4(\mathsf{SEG}\,4\,k\,t)]\,;\ldots]$$

where $r$ and $t$ are the operands which may be prefixed with $\mathsf{WNOT}$ and $k$ equals one of the following indexes 0, 4, 8, 12, 16, 20, 24, 28. This list of theorems is bound to the name `word4_seg_thms`.

```
let mk_sthm1 =
   let th = (SPEC "4"(RESQ_SPEC_ALL(SPEC "32" SEG_PWORDLEN))) in
   (\k. PURE_REWRITE_RULE[IMP_CLAUSES]
   (CONV_RULE (((LHS_CONV o LHS_CONV) ADD_CONV) THENC
         (LHS_CONV LESS_EQ_CONV)) (SPEC k th))) ;;

let mk_sthm2 = (\th. map (\th2. RESQ_MATCH_MP th th2) operands);;

let word4_seg_thms = map mk_sthm2 (map (RESQ_GEN_ALL o mk_sthm1) indexes);;
```

**4.3.7 Step 3** The theorems `icarry_thms` are used to simplify the lookahead expressions.

```
let icarry_thms =
   let ICARRY_THMS = map
   (\tm. (theorem `ALU_ARITH`
         (`ICARRY_`^(string_of_int(4 + (int_of_term tm)))))) indexes in
   (map (GSYM o (rev_itlist (\th1 th2. RESQ_MATCH_MP th2 th1) operands))
    ICARRY_THMS);;

let LOOKAHEAD_0_THM = REWRITE_RULE LOGIC_THMS
   (definition `ALUBIT` `LOOKAHEAD_0_DEF`);;
```

**4.3.8 Step 4** This section defines three conversions to be used in step 4 to move the $\mathsf{WNOT}$ out over the nested $\mathsf{WCAT}$s.

```
let conv1 =
   let wcat_wnot_CONV1 =
     let t1 = (lhs o snd o strip_resq_forall) (concl wcat_wnot4) in
     let inst_a4 = \t. GQSPECL (snd(strip_comb t)) word4_ADDER4 in
   \tm.
```

```
      let mlist = fst(match t1 tm) in
      let alist = (rev(map fst mlist)) in
    (RESQ_SPECL alist wcat_wnot4) in
  (RAND_CONV o FST_CONV o RAND_CONV o pair_CONV)
     ((RAND_CONV o pair_CONV) wcat_wnot_CONV1);;

let conv2 =
   let wcat_wnot_CONV2 =
      let t1 = (lhs o snd o strip_resq_forall) (concl wcat_wnot8) in
      let inst_wcat = \t. PWORDLEN_CONV ["4";"4"]
        (mk_comb("(PWORDLEN:num->bool word -> bool) 8", t)) in
   \tm.
      let mlist = fst(match t1 tm) in
      let alist = (rev(map fst mlist)) in
      (itlist PROVE_HYP (map inst_wcat alist) (RESQ_SPECL alist wcat_wnot8)) in
   (RAND_CONV o FST_CONV o RAND_CONV o pair_CONV) wcat_wnot_CONV2;;

let conv3 =
   let wcat_wnot_CONV3 =
      let t1 = (lhs o snd o strip_resq_forall) (concl wcat_wnot16) in
      let inst_wcat = \t.
     let th1 = PWORDLEN_CONV ["8";"8"]
      (mk_comb("(PWORDLEN:num->bool word -> bool) 16", t)) in
   (itlist PROVE_HYP (map (PWORDLEN_CONV ["4";"4"]) (hyp th1)) th1) in
   \tm.
      let mlist = fst(match t1 tm) in
      let alist = (rev(map fst mlist)) in
      (itlist PROVE_HYP (map inst_wcat alist)(RESQ_SPECL alist wcat_wnot16)) in
   (RAND_CONV o FST_CONV) wcat_wnot_CONV3;;
```

#### 4.3.9 pw4_adder_thm

This theorem is used in step 5 to get rid of ADDER4.

**HOL Theorem** (pw4_adder_thm)

$$
\begin{array}{l}
\forall r\, m\, cin. \\
\quad \mathsf{PWORDLEN}\,4\,r \supset (\mathsf{PWORDLEN}\,4\,m \supset \\
\quad\quad (\mathsf{ADDER4}\,r\,m\,cin = \\
\quad\quad\quad \mathsf{NBWORD}\,4\,(\mathsf{BNVAL}\,r + \mathsf{BNVAL}\,m + \mathsf{BV}\,cin)))
\end{array}
$$

```
let pw4_adder_thm =
   let pw4_ADDER4_THM = REWRITE_RULE [word4_def]
   (theorem `ALU_ARITH` `ADDER4_THM`) in
   CONV_RULE ((TOP_DEPTH_CONV RESQ_FORALL_CONV)
      THENC (TOP_DEPTH_CONV RIGHT_IMP_FORALL_CONV)) pw4_ADDER4_THM;;
```

**4.3.10 Simplify the implementation** There are five steps required to simplify the right hand side of the theorem ALU32BIT_THM. The first two are the same as for the bitwise operations. The result of these two steps is thm2. Step 3 simplifies the carries, its result is thm3. Step 4 is performed by the conversions conv1 to conv3. The last step is performed by the conditional rewriting conversion.

```
let alu_imp_thm =
   let thm2 = SUBS wnot_seg_thms (SUBS (seg_seg_r @ seg_seg_t)
     (RESQ_SPEC_ALL (SUBS[word32_def](RESQ_GEN_ALL ALU32BIT_THM)))) in
   let thm3 = SUBS icarry_thms
    (PURE_REWRITE_RULE[LOOKAHEAD_0_THM;NOT_CLAUSES] thm2) in
   (itlist PROVE_HYP (flat word4_seg_thms)
    (RIGHT_CONV_RULE(COND_REWRITE1_CONV [] pw4_adder_thm)
     (HYP_CONV_RULE pw_ADDER4_CONV
      (CONV_RULE (conv1 THENC conv2 THENC conv3) thm3))));;
```

**4.3.11 ac_thms** These theorems are used to replace ACARRY by ICARRY. They are just instances of the theorem ACARRY_EQ_ICARRY.

```
let mk_ac_thm op1 op2 =
   let wnot_thm = hd operands in
   let th = (GQSPECL
    ["32"; op1; op2; "cin:bool"] ACARRY_EQ_ICARRY) in
   \k. itlist PROVE_HYP operands
   (GEN "cin:bool"(MP (SPEC k th) (EQT_ELIM (LESS_EQ_CONV "^k <= 32"))));;

let ac_thms = map (mk_ac_thm(mk_comb("WNOT",rbar)) treg) indexes;;
```

**4.3.12 Expand the specification** We expand the expressions representing the sum using the theorem ADD_WORD_THM which states that the addition of two 32-bit words can be split into eight 4-bit segments. ADD32_THM is used to eliminate the function ADD32.

```
let ADD32_THM = (theorem `ALU_ARITH` `ADD32_THM`);;

let alu_spec_thm =
   let ADD_WORD_THM = theorem `ALU_ARITH` `ADD_WORD_THM` in
   let add32_thm = (GEN_ALL o DISCH_ALL)(SPEC_ALL (RESQ_SPEC_ALL ADD32_THM)) in
   let thm10 =
     (RIGHT_CONV_RULE (COND_REWRITE1_CONV [] add32_thm)
   (RESQ_SPEC_ALL (SUBS[word32_def](RESQ_GEN_ALL ALU_SPEC)))) in
   REWRITE_RULE ac_thms (itlist PROVE_HYP operands
   (RIGHT_CONV_RULE ((FST_CONV o RAND_CONV)
     (RESQ_REWRITE1_CONV [] ADD_WORD_THM)) thm10));;
```

**4.3.13 The equivalence theorem** We can now attempt to prove that the specification is equal to the implementation for the addition operation. We use goal directed proof here. The main result of the operation at the right hand sides of the theorems `alu_imp_thm` and `alu_spec_thm` should be identical. There are theorems saved in the theories `ALU_WORD` and `ALU_ARITH` which should match the condition bits. They are fetched to solve the goal. The resulting theorem is

**HOL Theorem** (`ALU_EQ_THM`)

$$\vdash \forall rbar\, treg :: \mathsf{PWORDLEN}\,32.\, \forall cin\, bflag.$$
$$\mathsf{ALU}\; rbar\; treg\; cin\; bflag\; \mathsf{alu\_add} = \mathsf{ALU\_32BIT}\; rbar\; treg\; cin\; bflag\; \mathsf{alu\_add}$$

```
let ALU_EQ_THM =
   let op = mk_const(alu_op, ":alu_op") in
   prove_thm(`ALU_EQ_THM`,
   "!rbar treg::PWORDLEN 32. !cin bflag.
   ALU rbar treg cin bflag ôp = ALU_32BIT rbar treg cin bflag ôp",
   let cond_thms = map (\(thy, thm). theorem thy thm)
    [ ( `ALU_ARITH`, `COND_AOUT31_ADD`);
      ( `ALU_WORD`,  `COND_R0`);
      ( `ALU_WORD`,  `COND_R30`);
      ( `ALU_ARITH`, `COND_R31_ADD`);
      ( `ALU_ARITH`, `COND_R31_ADD`);
      ( `ALU_ARITH`, `COND_COUTBAR_ADD`);
      ( `ALU_ARITH`, `COND_NZTOP12_ADD`);
      ( `ALU_ARITH`, `COND_ZMID4_ADD`);
      ( `ALU_ARITH`, `COND_NZBOT16_ADD`) ] in
   REPEAT RESQ_GEN_TAC THEN REPEAT GEN_TAC
   THEN SUBST_TAC[alu_imp_thm;alu_spec_thm]
   THEN PURE_ONCE_REWRITE_TAC[PAIR_EQ] THEN CONJ_TAC THENL[
    REFL_TAC;
    MAP_EVERY ASSUME_TAC operands THEN PURE_ONCE_REWRITE_TAC[WORD_11]
    THEN REWRITE_TAC[CONS_11] THEN REPEAT CONJ_TAC
    THENL (map RESQ_REWRITE1_TAC cond_thms)
    THEN REFL_TAC]);;
```

## 4.4 Stage 2: shift left operations

The difficulty occurring with the shift left operation $\mathsf{alu\_sl}$ is due to the fact that the different approaches taken to perform the operation at the two levels are very different. The specification model specifies that the result of this operation should be the sum of adding the operand $r$ to itself plus the carry input $c_{in}$. whilst the implementation model specifies that the segments of the operand $r$ are actually shifted to the left one bit and the most significant bit of each segment is passed to the next more significant slice as the carry input. This carry input is padded to the end of that segment.[2]

In the formal implementation model, shift left is expressed in terms of the generic shift operator SHL defined as below:

**HOL Definition** (SHL_DEF)

$$
\begin{array}{l}
\vdash \forall f\, w\, b. \\
\quad \mathsf{SHL}\, f\, w\, b = \\
\quad\quad \mathsf{BIT}\,(\mathsf{PRE}\,(\mathsf{WORDLEN}\, w))\, w, \\
\quad\quad \mathsf{WCAT}(\mathsf{SEG}\,(\mathsf{PRE}\,(\mathsf{WORDLEN}\, w))\, 0\, w, (f \Rightarrow \mathsf{SEG}\, 1\, 0\, w \mid \mathsf{WORD}\,[b]))
\end{array}
$$

Depending on the value of the arguments $f$ and $b$, SHL can perform either logical shift, arithmetic shift or rotation. The behaviour required in the implementation is obtained by setting the first argument $f$ to F, the last argument $b$ to $\mathsf{BIT}\,(j-1)\, w$ when performing a shift on the segment $\mathsf{SEG}\, 4\, j\, w$. The theorem SHL_SEG_NF asserts that this is the desired operation.

**HOL Theorem** (SHL_SEG_NF)

$$
\begin{array}{l}
\vdash \forall n. \forall w :: \mathsf{PWORDLEN}\, n. \forall k\, m. \\
\quad k + m \leq n \supset (0 < m \supset (0 < k \supset \\
\quad\quad (\mathsf{SHL}\,\mathsf{F}\,(\mathsf{SEG}\, m\, k\, w)\,(\mathsf{BIT}\,(k-1)\, w) = \\
\quad\quad\quad \mathsf{BIT}\,(k + (m-1))\, w, \mathsf{SEG}\, m\,(k-1)\, w)))
\end{array}
$$

For binary words, these different approaches do produce the same results. The theorem DOUBL_EQ_SHL asserts this fact.

**HOL Theorem** (DOUBL_EQ_SHL)

$$
\begin{array}{l}
\vdash \forall n. 0 < n \supset \\
\quad (\forall w :: \mathsf{PWORDLEN}\, n. \forall b. \\
\quad\quad \mathsf{NBWORD}\, n\,(\mathsf{BNVAL}\, w + \mathsf{BNVAL}\, w + \mathsf{BV}\, b) = \mathsf{SND}\,(\mathsf{SHL}\,\mathsf{F}\, w\, b))
\end{array}
$$

The proof procedures for this operation follow roughly the same approach as in the addition operation. The following subsections list the contents of the driver file for the operation (alu_sl_thm.m).

[2]The reason of this divergence is probably historical (speculated by the author). In early versions of the HOL system, there was not shift function for word types.

**4.4.1 Define the current operation** The name of the current operation, as a string, is bound to the identifier `alu_op`. It is used in all calls to proof functions.

```
letref alu_op = `alu_sl`;;
```

**4.4.2 Stage 1:specification** The function `ALU_CASES_RULE` is called to unfold the specification. After unfolding the specification, we obtain the following expression for the main result:

$$\mathsf{WNOT}\,(\mathsf{SEG}\,32\,0\,(\mathsf{ADD32}\,(\mathsf{WNOT}\,rbar)\,(\mathsf{WNOT}\,rbar)\,cin))$$

```
let ALU_SPEC =
   let rwthms = [NOT_CLAUSES; NOT_THM1] in
   ALU_CASES_RULE rwthms alu_op;;
```

**4.4.3 Stage 1:implementation** The functions for unfolding the implementation are called here. The expression representing the main result in the theorem `ALU32BIT_THM` is:

```
WCAT
(WCAT
 (WCAT
  (WNOT
   (SND (SHL F
     (WNOT(SEG 4 12(SEG 16 16 rbar)))
     (BIT 3(WNOT(SEG 4 8(SEG 16 16 rbar)))))),
   WNOT
   (SND (SHL F
     (WNOT(SEG 4 8(SEG 16 16 rbar)))
     (BIT 3(WNOT(SEG 4 4(SEG 16 16 rbar))))))),
  WCAT
  (WNOT
   (SND (SHL F
     (WNOT(SEG 4 4(SEG 16 16 rbar)))
     (BIT 3(WNOT(SEG 4 0(SEG 16 16 rbar)))))),
   WNOT
   (SND (SHL F
     (WNOT(SEG 4 0(SEG 16 16 rbar)))
     (BIT 3(WNOT(SEG 4 12(SEG 16 0 rbar)))))))),
 WCAT
 (WCAT
  (WNOT
   (SND (SHL F
     (WNOT(SEG 4 12(SEG 16 0 rbar)))
     (BIT 3(WNOT(SEG 4 8(SEG 16 0 rbar)))))),
   WNOT
   (SND (SHL F
```

```
      (WNOT(SEG 4 8(SEG 16 0 rbar)))
      (BIT 3(WNOT(SEG 4 4(SEG 16 0 rbar))))))),
  WCAT
  (WNOT
   (SND (SHL F
      (WNOT(SEG 4 4(SEG 16 0 rbar)))
      (BIT 3(WNOT(SEG 4 0(SEG 16 0 rbar)))))),
   WNOT
   (SND (SHL F(WNOT(SEG 4 0(SEG 16 0 rbar)))cin)))))
```

```
let ALU32BIT_THM =
   let ADD4BIT_DEF = (definition `ALUBIT` `ADD4BIT_DEF`) in
   let rwthms = (NOT_CLAUSES . LOGIC_THMS) in
   let ALU4BIT_THM =
      ALU4BIT_CASES_RULE [ADD4BIT_DEF;FST;SND] alu_op in
   let ALU16BIT_THM =
   GEN_ALL(RESQ_GEN_ALL(SUBS[sl_la10;sl_la11;sl_la12;sl_la13;sl_la14]
       (RESQ_SPEC_ALL(SPEC_ALL
        (ALU16BIT_CASES_RULE ALU4BIT_THM
     [FST;SND;LOOKAHEAD_4_THMF] alu_op))))) in
   ALU32BIT_CASES_RULE ALU16BIT_THM rwthms alu_op;;
```

**4.4.4 Simplify the theorems—stage 2** After stage 1, we have obtained two theorems stating the operation results of the two ALU models: the specification — `ALU_SPEC` and the implementation `ALU32BIT_THM` (See Section 4.1). We can now work on these theorems to reduce their right hand sides to the same expression by simplifying the implementation theorem and/or expanding the specification theorem.

**4.4.5** Terms and theorems specific to each operation are bound to some ML identifiers so that they can be passed to the proof functions. The identifier `operands` is bound to a list of theorems which asserts that the size of operands are 32-bit.

```
let operands = [pw_wnot_rbar];;
```

**4.4.6** We also need a list of theorems in the following form:

$$[[\mathsf{PWORDLEN}\,32\,r \vdash \mathsf{PWORDLEN}\,4(\mathsf{SEG}\,4\,k\,r)]\,;\ldots]$$

where $r$ is the operands which may be prefixed by $\mathsf{WNOT}$ and $k = 0, 4, 8, 12, 16, 20, 24, 28$. This list is bound to the name `word4_seg_thms`.

```
let mk_sthm1 =
   let th = (SPEC "4"(RESQ_SPEC_ALL(SPEC "32" SEG_PWORDLEN))) in
   (\k. PURE_REWRITE_RULE[IMP_CLAUSES]
   (CONV_RULE (((LHS_CONV o LHS_CONV) ADD_CONV) THENC
         (LHS_CONV LESS_EQ_CONV)) (SPEC k th))) ;;

let mk_sthm2 =
   (\th. RESQ_MATCH_MP th (hd operands));;

let word4_seg_thms = map mk_sthm2 (map (RESQ_GEN_ALL o mk_sthm1) indexes);;
```

#### 4.4.7 Step 3

The theorem `bit_seg_thms` are used to simplify expressions of the following form: $\mathsf{BIT}\,3(\mathsf{SEG}\,4\,j\,w)$. The theorems `shl_thms` are used to simplify the $\mathsf{SHL}$ expressions. They are in the following form:

$$\mathsf{SND}\,(\mathsf{SHL}\,\mathsf{F}\,(\mathsf{SEG}\,4\,4\,(\mathsf{WNOT}\,rbar))\,(\mathsf{BIT}\,3\,(\mathsf{WNOT}\,rbar))) = \\ \mathsf{SEG}\,4\,3\,(\mathsf{WNOT}\,rbar)$$

```
let bit_seg_thms =
   let lem = GEN_ALL(REWRITE_RULE[LESS_CONV "3 < 4"](SPECL["3";"k:num";"4"]
   (RESQ_MATCH_MP (SPEC "32" BIT_SEG) (hd operands)))) in
   map
   (\k. let lm3 = EQT_ELIM
     (((LHS_CONV ADD_CONV) THENC LESS_EQ_CONV) "(^k+4)<=32") in
     RIGHT_CONV_RULE ((RATOR_CONV o RAND_CONV)ADD_CONV)(MATCH_MP lem lm3))
   indexes;;

let shl_thms =
   let lem1 = GEN "k:num" (DISCH_ALL(PROVE_HYP (EQT_ELIM(LESS_CONV "0<4"))
    (REWRITE_RULE[SND]
    (AP_TERM "SND:(bool # (bool)word)-> (bool)word" (UNDISCH_ALL
     (SPECL["k:num";"4"]
   (RESQ_MATCH_MP (SPEC "32" SHL_SEG_NF) (hd operands)))))))) in
   map
   (\k. let lm1 = num_CONV k in let k' = term_of_int((int_of_term k) -1) in
    let lm2 = SUBS[SYM lm1] (SPEC k' LESS_0) in
    let lm3 = EQT_ELIM
      (((LHS_CONV ADD_CONV) THENC LESS_EQ_CONV) "(^k+4)<=32") in
    SUBS[SYM lm1](REWRITE_RULE[SUC_SUB1](SUBS[lm1]
    (UNDISCH_ALL(LIST_MP [lm2; lm3] (SPEC k lem1)))))) (tl indexes);;
```

#### 4.4.8 Simplify the implementation

We substitute the theorems derived above into the implementation theorem `ALU32BIT_THM` to simplify the right hand side and obtain the following theorem:

```
PWORDLEN 32 rbar, PWORDLEN 32 treg
|- ALU_32BIT rbar treg cin bflag alu_sl =
   WCAT
   (WCAT
    (WCAT(WNOT(SEG 4 27(WNOT rbar)),WNOT(SEG 4 23(WNOT rbar))),
     WCAT(WNOT(SEG 4 19(WNOT rbar)),WNOT(SEG 4 15(WNOT rbar)))),
    WCAT
    (WCAT(WNOT(SEG 4 11(WNOT rbar)),WNOT(SEG 4 7(WNOT rbar))),
     WCAT(WNOT(SEG 4 3(WNOT rbar)),WNOT(SND(SHL F(SEG 4 0(WNOT rbar))cin))))),
   WORD
   [BIT 3(SEG 4 27(WNOT rbar));BIT 0(SEG 4 0(WNOT rbar));
    BIT 2(SEG 4 28(WNOT rbar));BIT 31(WNOT rbar);
    F;~BIT 31(WNOT rbar);
    ~(WORD
      [SEG 4 27(WNOT rbar) = NBWORD 4 0;
       SEG 4 23(WNOT rbar) = NBWORD 4 0;
       SEG 4 19(WNOT rbar) = NBWORD 4 0] =
      WORD[T;T;T]);SEG 4 15(WNOT rbar) = NBWORD 4 0;
    ~(WORD
      [SEG 4 11(WNOT rbar) = NBWORD 4 0;
       SEG 4 7(WNOT rbar) = NBWORD 4 0;
       SEG 4 3(WNOT rbar) = NBWORD 4 0;
       SND(SHL F(SEG 4 0(WNOT rbar))cin) = NBWORD 4 0] =
      WORD[T;T;T;T])]
```

```
let alu_imp_thm =
   itlist SUBS [shl_thms; bit_seg_thms; wnot_seg_thms; seg_seg_r]
     (RESQ_SPEC_ALL (SUBS[word32_def](RESQ_GEN_ALL ALU32BIT_THM)));;
```

**4.4.9 Expand the specification** We use the theorem DOUBL_EQ_SHL to rewrite the specification to eliminate the addition so that every thing is in terms of the shift operation.

```
let ADD32_THM = theorem `ALU_ARITH` `ADD32_THM`;;

let alu_spec_thm =
   let dshl = MATCH_MP DOUBL_EQ_SHL
   (SUBS[SYM(num_CONV "32")](SPEC "31" LESS_0)) in
   let add32_thm = (GEN_ALL o DISCH_ALL)(SPEC_ALL (RESQ_SPEC_ALL ADD32_THM)) in
   let thm10 =
     (RIGHT_CONV_RULE (COND_REWRITE1_CONV [] add32_thm)
   (RESQ_SPEC_ALL (SUBS[word32_def](RESQ_GEN_ALL ALU_SPEC)))) in
   (itlist PROVE_HYP operands
    (RIGHT_CONV_RULE (RESQ_REWRITE1_CONV [] dshl) thm10));;
```

**4.4.10 Equivalence theorem** We can now attempt to prove the equivalence between the specification and the implementation for the alu_sl operation. The proof is lengthy. Basically, we attempt to prove the corresponding fields of these two levels are equal.

For the main result field, we need to prove that shifting a 32-bit word as a whole to the left is equivalent to shifting the segments and then concatenating them. This can be proved using properties of basic word operations.

For the condition bit `aout31`, we need to show the most significant bit of the result of shifting the whole 32-bit word is the same as the most significant bit of the result of shifting the most significant 4-bit segment.

The condition bits `r0`, `r30` and `r31` can be proved as in other operations since they are just specific bits of the operand $r$. The condition bit `rm31` is trivial since both levels return the constant $\mathsf{F}$.

The condition bit `coutbar` in the implementation is the 31st bit of the operand while in the specification it is the most significant bit of the sum of adding the operand to itself. The theorem `MSB_DOUBLE` asserts this fact.

What needs to be proved in the remaining three condition bits is whether some specific segment of the operand is equal to zero. For instance, the bit `zmid4` at the specification level is the following expression

$$\mathsf{BNVAL}\,(\mathsf{SEG}\,4\,16\,(\mathsf{SND}\,(\mathsf{SHL}\,\mathsf{F}\,(\mathsf{WNOT}\,rbar)\,cin))) = 0$$

while the corresponding expression in the implementation is

$$\mathsf{SEG}\,4\,15\,(\mathsf{WNOT}\,rbar) = \mathsf{NBWORD}\,4\,0.$$

The former can be simplified using the theorems about $\mathsf{SHL}$ and basic word operation $\mathsf{SEG}$. Then they can be unified using the theorem `ZERO_WORD_VAL`. The resulting theorem is saved in the theory `alu_sl_thm` under the name `ALU_EQ_THM`.

**HOL Theorem** (`ALU_EQ_THM`)

$$\begin{array}{l} \vdash \forall rbar\, treg :: \mathsf{PWORDLEN}\,32.\ \forall cin\, bflag.\\ \quad \mathsf{ALU}\,rbar\,treg\,cin\,bflag\,\mathsf{alu\_sl} = \mathsf{ALU\_32BIT}\,rbar\,treg\,cin\,bflag\,\mathsf{alu\_sl} \end{array}$$

## 4.5 The final theorem

After the case analysis which results in a set of theorems stating the results of every ALU operation, it is not too difficult to prove the goal stated in Section 3. This section lists the contents of the file `equiv.m` which contains the proof of the final theorem.

**4.5.1 IS_ALU_IMP_DEF** First of all, a predicate is defined to state what is meant by an ALU implementation. A function $f$ is an implementation of the ALU specified by $\mathsf{ALU}$ if and only if it satisfies the predicate $\mathsf{IS\_ALU\_IMP}$.

**HOL Definition** (`IS_ALU_IMP_DEF`)

$$\begin{array}{l} \vdash \forall f.\ \mathsf{IS\_ALU\_IMP}\,f =\\ \quad (\forall r\,t :: \mathsf{word32}.\ \forall cin\,bflag\,alucon. \end{array}$$

$$\begin{array}{l}
\neg \mathsf{IS_BITOP}\, alucon \wedge \\
\quad (\mathsf{ALU}\, r\, t\, cin\, bflag\, alucon = f\, r\, t\, cin\, bflag\, alucon) \vee \\
(\mathsf{FST}\, (\mathsf{ALU}\, r\, t\, cin\, bflag\, alucon) = \mathsf{FST}\, (f\, r\, t\, cin\, bflag\, alucon)) \wedge \\
\quad \textbf{let}\ cond' = \mathsf{SND}\, (f\, r\, t\, cin\, bflag\, alucon) \\
\quad \textbf{and}\ cond = \mathsf{SND}\, (\mathsf{ALU}\, r\, t\, cin\, bflag\, alucon)\ \textbf{in} \\
\qquad ((\mathsf{COND_AOUT31}\, cond = \mathsf{COND_AOUT31}\, cond') \wedge \\
\qquad (\mathsf{COND_R0}\, cond = \mathsf{COND_R0}\, cond') \wedge \\
\qquad (\mathsf{COND_R30}\, cond = \mathsf{COND_R30}\, cond') \wedge \\
\qquad (\mathsf{COND_R31}\, cond = \mathsf{COND_R31}\, cond') \wedge \\
\qquad (\mathsf{COND_RM31}\, cond = \mathsf{COND_RM31}\, cond') \wedge \\
\qquad (\mathsf{COND_NZTOP}\, cond = \mathsf{COND_NZTOP}\, cond') \wedge \\
\qquad (\mathsf{COND_ZMID}\, cond = \mathsf{COND_ZMID}\, cond') \wedge \\
\qquad (\mathsf{COND_NZBOT}\, cond = \mathsf{COND_NZBOT}\, cond')))
\end{array}$$

The definition specifies that the function $f$ should return the same value as ALU if the operation is non-bitwise, otherwise, all fields except `COUTBAR` of its return value should be equal to the corresponding field of ALU.

```
let IS_ALU_IMP_DEF = new_definition(`IS_ALU_IMP_DEF`,
  "IS_ALU_IMP (f:(bool)word->(bool)word->bool->bool->alu_op->
  (bool)word#(bool)word) =
    !r t::word32. !cin bflag alucon.
   ((~(IS_BITOP alucon) /\
     ((ALU r t cin bflag alucon) = (f r t cin bflag alucon))) \/
    ((FST(ALU r t cin bflag alucon) = FST(f r t cin bflag alucon)) /\
     (let cond'= SND(f   r t cin bflag alucon) and
          cond = SND(ALU r t cin bflag alucon) in
      ((COND_AOUT31 cond  = COND_AOUT31 cond') /\
       (COND_R0 cond      = COND_R0 cond') /\
       (COND_R30 cond     = COND_R30 cond') /\
       (COND_R31 cond     = COND_R31 cond') /\
       (COND_RM31 cond    = COND_RM31 cond') /\
       (COND_NZTOP12 cond = COND_NZTOP12 cond') /\
       (COND_ZMID4 cond   = COND_ZMID4 cond') /\
       (COND_NZBOT16 cond = COND_NZBOT16 cond'))))))");;
```

**4.5.2 ALU_EQ_TAC** This tactic solves a goal in one of the following two forms:

1.
```
"((ALU r t cin bflag alucon) = (f r t cin bflag alucon)) \/
 ((FST(ALU r t cin bflag alucon) = FST(f r t cin bflag alucon)) /\
  (let cond'= SND(f   r t cin bflag alucon) and
       cond = SND(ALU r t cin bflag alucon) in
   ((COND_AOUT31 cond  = COND_AOUT31 cond') /\
    (COND_R0 cond      = COND_R0 cond') /\
    (COND_R30 cond     = COND_R30 cond') /\
    (COND_R31 cond     = COND_R31 cond') /\
    (COND_RM31 cond    = COND_RM31 cond') /\
```

```
    (COND_NZTOP12 cond = COND_NZTOP12 cond') /\
    (COND_ZMID4 cond   = COND_ZMID4 cond') /\
    (COND_NZBOT16 cond = COND_NZBOT16 cond'))))
```

2.

```
"((FST(ALU r t cin bflag alucon) = FST(f r t cin bflag alucon)) /\
 (let cond'= SND(f   r t cin bflag alucon) and
      cond = SND(ALU r t cin bflag alucon) in
   ((COND_AOUT31 cond  = COND_AOUT31 cond') /\
    (COND_R0 cond      = COND_R0 cond') /\
    (COND_R30 cond     = COND_R30 cond') /\
    (COND_R31 cond     = COND_R31 cond') /\
    (COND_RM31 cond    = COND_RM31 cond') /\
    (COND_NZTOP12 cond = COND_NZTOP12 cond') /\
    (COND_ZMID4 cond   = COND_ZMID4 cond') /\
    (COND_NZBOT16 cond = COND_NZBOT16 cond'))))"
```

where `alucon` should be a constant representing a specific ALU operation. It checks the goal to see what operation is being dealt with. It then opens the appropriate theory to fetch the required theorem(s). For non-bitwise operation, i.e., the goal is in form 1, the first disjunct will be proved. The theorem fetched should match it. For bitwise operations, the theorems fetched are $\alpha$-converted, and the results are substituted into the goal. The applications of projection operators COND_$XXX$ are reduced. The terms on either sides of the equation should match each other.

```
let ALU_EQ_TAC =
  \(asm,gl).
    let op,[tml;tmr] = strip_comb gl in
    if (op = "$\/")
    then % arith ops %
       (let args = (snd o strip_comb o snd)(dest_eq tml) in
        let vars = butlast args and alu_op = fst (dest_const(last args)) in
        let thm = GQSPECL vars (theorem (`ALU_THM_`âlu_op) `ALU_EQ_THM`) in
        ((DISJ1_TAC THEN ACCEPT_TAC thm) (asm, gl)))
    else if (op = "$/\") then % bitwise ops %
       (let args = (snd o strip_comb o snd o dest_comb o snd)(dest_eq tml) in
        let vars = butlast args and alu_op = fst (dest_const(last args)) in
        let P = "PWORDLEN 32:bool word -> bool" in
        let rbar = "rbar:bool word" and treg = "treg:bool word" in
        let thms = map (\s. GQSPECL vars
        (RESQ_GENL[(rbar,P); (treg,P)](GENL["cin:bool"; "bflag:bool"]
      (theorem (`ALU_THM_`âlu_op) s))))
    [`alu_spec_thm`; `alu_imp_thm`] in
        ((SUBST_TAC thms THEN PURE_ONCE_REWRITE_TAC[FST;SND]
          THEN CONJ_TAC THENL[
    REFL_TAC;
    CONV_TAC let_CONV THEN PURE_ONCE_REWRITE_TAC cond_bit_thms
       THEN REPEAT CONJ_TAC THEN REFL_TAC
           ])(asm,gl)))
    else failwith `ALU_EQ_TAC` ;;
```

**4.5.3 The final theorem** We now prove the final theorem using goal-dircted proof technique. We set up the goal `?- IS_ALU_IMP ALU_32BIT`. We first rewrite the goal with the definition of the predicate and get rid of the quantifiers. We then do a case split on the ALU operations *alucon* using the tactic `STRUCT_CASES_TAC` with the constructor cases theorem `alu_op_cases`. The resulting subgoals can be simplified according to whether they are bitwise operation. Using the tactic `ALU_EQ_TAC`, all subgoals can be solved, the final theorem is returned.

**HOL Theorem** (`EQUIV_THM`)

$$\vdash \mathsf{IS\_ALU\_IMP\ ALU\_32BIT}$$

```
let EQUIV_THM = prove_thm(`EQUIV_THM`,
   "IS_ALU_IMP ALU_32BIT",
   PURE_ONCE_REWRITE_TAC[IS_ALU_IMP_DEF]
   THEN PURE_ONCE_REWRITE_TAC[word32_def]
   THEN REPEAT RESQ_GEN_TAC THEN REPEAT GEN_TAC
   THEN STRUCT_CASES_TAC (SPEC "alucon:alu_op" alu_op_cases)
   THEN REWRITE_TAC[(definition `COMMON` `IS_BITOP_DEF`)]
   THEN ALU_EQ_TAC);;
```

# 5 Benchmark

As the ALU is a realistic hardware design, it is very interesting to examine the magnitude of the proof and the time required to complete it. The benchmark data were obtained by executing the proof on a Sun Sparc Server 10 with 90 Mbytes of physical memory at a time it was not heavily loaded. The time taken to create each theory(RUN TIME), the garbage collection time (GC TIME) and the number of intermediate theorems are listed in Table 3. The total machine time reported by HOL (Run time + GC time) is 3211.2 seconds which is 53.52 minutes. The real elapsed time reported by the operating system is one hour and 26 minutes (1:26:36).

The total number of intermediate theorems is about twice as many as the first level VIPER proof, and is one order of magnitude smaller than the second level VIPER proof. (The figures for these proofs are $230,036$ and $7,153,000$, respectively.[1] [2]) Some key factors which keep the magnitude of the proof reasonably small and manageable are:

1. choose a suitable model for the data and structure,
2. use general facts wherever possible,
3. use more specific proof strategies.

| **THEORY** | **RUN TIME** | **GC TIME** | **THEOREMS** |
|---|---|---|---|
| `logic` | 1.0 | 0.0 | 91 |
| `word_widths` | 21.0 | 4.9 | 441 |
| `common` | 45.3 | 11.2 | 8402 |
| `alu` | 24.7 | 6.4 | 744 |
| `alubit` | 32.2 | 9.1 | 1456 |
| `alu_word` | 39.8 | 10.6 | 7819 |
| `alu_arith` | 188.7 | 70.6 | 61494 |
| `alu_add_thm` | 151.9 | 95.8 | 29122 |
| `alu_sub_thm` | 156.2 | 99.9 | 29386 |
| `alu_inc_thm` | 158.8 | 88.7 | 33298 |
| `alu_sl_thm` | 193.7 | 132.0 | 49503 |
| `alu_0_thm` | 96.8 | 49.2 | 23479 |
| `alu_sr_thm` | 175.1 | 111.6 | 47880 |
| `alu_m_thm` | 98.6 | 50.9 | 23776 |
| `alu_com_thm` | 105.3 | 54.5 | 25782 |
| `alu_r_thm` | 105.2 | 54.6 | 25769 |
| `alu_and_thm` | 113.7 | 61.8 | 27816 |
| `alu_rmb_thm` | 112.4 | 60.0 | 27878 |
| `alu_xor_thm` | 111.1 | 59.9 | 27830 |
| `alu_nor_thm` | 117.7 | 63.8 | 29857 |
| `equiv` | 46.7 | 19.8 | 6937 |
| **Total** | 2095.9 | 1115.3 | 488760 |

Table 3: Timing results

The primary data objects involved in hardware verification is word. The model used in this project, namely a generic word type and dependent types simulated using restricted quantifiers for specific word sizes, proved very suitable for the problem. The `word` library provides many general facts about words which can easily be instantiated for specific application. Throughout the proof, more primitive proof procedures, such as instantiation(`SPEC`) and substitution(`SUBST`) are used instead of powerful rewriting tools wherever this is practical. This helps to reduce the number of intermediate theorems as well as the run time considerably.

It is very difficult to compare the time with the VIPER verification projects since the machine in which the benchmark is obtained is much more powerful than those in the old projects. However, the key point is that it is practical to verify realistic design of similar magnitude providing a proper infrastructure for reasoning about hardware is developed.

# 6 Conclusion

This report has shown that a formal machine-assisted proof of the functional equivalence between two levels of the ALU formal model has been completed successfully. This confirms the result of the manual proof carried out previously. Providing that the formal models of the ALU are faithful to the original description in NODEN_HDL, the formal proof verifies that the bit-slice ALU correctly implements the operations described in the specification. By carefully comparing the HOL description with the original NODEN_HDL description, one can be very confident that the formal model is a faithful representation of the ALU design.

The ALU formal model and the proof are based on an assumption that the circuitry is purely combinational, and that the outputs will eventually become stable after the inputs are stable. The models describe only the functional behaviour. Therefore, no time variables have been used.

As has been pointed out by many researchers, there is always a problem in ensuring the accuracy of the formal models in relation to the actual design and the actual device being fabricated. The physical device will never be verified in a mathematical sense. Nevertheless, past experience has confirmed that the process of creating formal models and the subsequent proof of certain properties, such as functional correctness, *does* help to discover many errors. Therefore, formal verification is not a burden but a real benefit to the designers.

In comparison with the original VIPER verification project, the verification of the ALU is relatively easy. This is because

- the size of the proof is smaller;
- there is now much better support in the HOL system; and

- lessons have been learnt from previous experience of VIPER and other hardware verification projects.

As part of the achievement of the project, the `word` library was created which provides a better infrastructure for reasoning about words or bit vectors in this project. Future projects involving words will also benefit.

# References

[1] A. Cohn. A proof of correctness of the VIPER microprocessor: the first level. Technical Report 104, University of Cambridge Computer Laboratory, January 1988.

[2] A. Cohn. A proof of correctness of the VIPER microprocessor: the second level. Technical Report 134, University of Cambridge Computer Laboratory, May 1990.

[3] W. J. Cullyer. VIPER microprocessor: Formal specification. Technical Report 85013, Royal Signals and Radar Establishment, October 1985.

[4] W. J. Cullyer. *Implementing Safety Critical Systems: The VIPER microprocessor*, pages 1–26. Kluwer Academic Publishers, 1987.

[5] D. E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, May 1984.

[6] C. H. Pygott. Verification of VIPER's ALU. Technical report, Devisional Memo (Draft), the Royal Signals and Radar Establishment, 1991.

# A Notes on proof management and document production

1. All proof script files have been written in a special format which combines the ML source and documentation in a master file. A couple of utility programs `mtangle` and `mweave` convert the master files into ML files and LaTeX files, respectively. Figure 7 illustrates this process. Although these utility programs have been developed as part of the project, they will be very useful to the entire HOL user community at large. This approach is inspired by the *literate programming* method developed by D. Knuth[5].

2. All files are managed using the `make` utility. They are arranged into several directories:

   `alu` —the proof directory which contains all the proof script master files, the ML files and the theory files: proof is carried out in this directory.

   `doc` —the document directory which contains all the LaTeX files: reports are produced in this directory.

   `web` —the utility directory which contains the source and executable of the utility programs `mtangle` and `mweave`.

   `theories` —the tagged theory directory which contains files generated using the `latex-hol` library. These files contain pretty printed HOL definitions and theorems in LaTeX format. The LaTeX source of all HOL definitions and theorems in this report was generated automatically using the `latex-hol` library with small amounts of manual editing.

   The dependency of the files are specified in `Makefiles` in the directory `alu` and `doc`. The proof is performed by typing the command

   ```
   make EQUIV.th
   ```

   in the `alu` directory while the document in PostScript is produced by typing the command

   ```
   make report.ps
   ```

   in the `doc` directory.

3. The typographical conventions used in this report are as follows:

   - HOL expressions are typeset in math mode with constants set in sans serif font, variables set in *italic* font;
   - HOL built-in special constants such as $\wedge$, $\forall$ and so on, are typeset with conventional mathematical symbols;

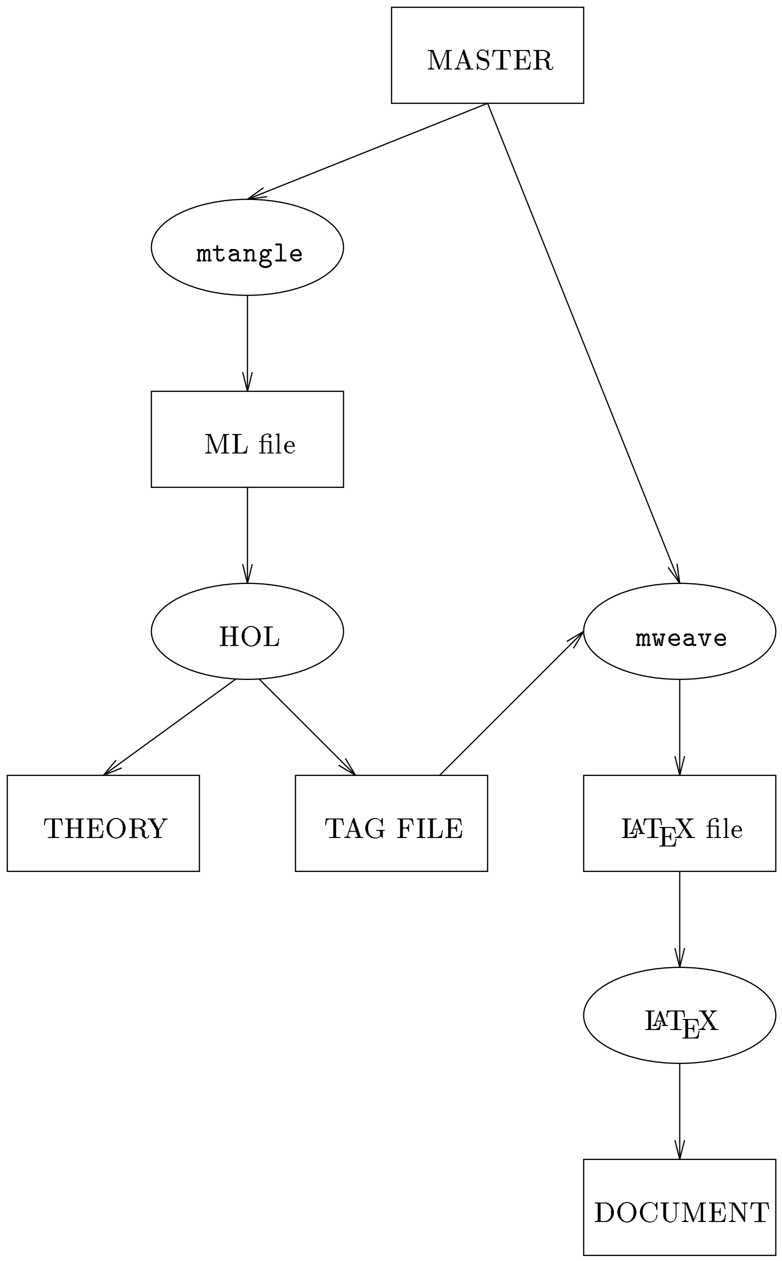


Figure 7: Processing the proof script master file

- ML text and identifiers, file names and theory names are typeset in `typewriter` font;
- some sections of the report are produced by including the proof script file generated by the utility `mweave`. Text in these sections are set in a smaller size.

# B ALU models in NODEN

This appendix lists the original ALU description in NODEN_HDL. This appeared in the report on the manual proof by Pygott[6]. It is divided into two parts: the specification and the implementation. They are listed in separate subsections. In fact, the NODEN_HDL specification is based on the ALU function in HOL in the original VIPER verification (see Appendix C). The notes at the beginning of the specification can be read in a reverse sense when comparing this description with the formal models presented in the report.

## B.1 ALU specification in NODEN_HDL

The following is a transliteration of the HOL description of the 32-bit ALU (from Annex A of [6]) into NODEN_HDL. A number of minor changes should be noted.

- The ALU operation codes, which in HOL were 7-bit boolean vectors (such as `#0011001`) have been replaced by an enumerated type `alu_op`.
- NODEN_HDL doesn't have built-in operators corresponding to HOL's NOT32, AND32 and OR32. So equivalent NODEN_HDL functions are defined at the start of the description.
- HOL's selector construct ($a \Rightarrow b \mid c$) is replaced either by NODEN_HDL's `IF` statement `IF a THEN b ELSE c FI` (as in `FIND_SR`), or by a `CASE` construct (as in `BITOP`).
- The HOL function ADD32 has been incorporated into the NODEN_HDL function `ADD32BIT` (in the definition of `sum33`).
- The HOL indexing operation EL is replaced by a NODEN_HDL index, and the HOL slice operation SEG by a NODEN_HDL slice. Note that the indices in NODEN_HDL are one greater than those in HOL, as HOL indexes structures from element 0, whilst NODEN_HDL indexes from 1.
- In `BITOP`, no definition is required for `cout_x`, as the *don't-care* value `(?bool)` can be used in its place.
- The HOL functions GET_AOUT and GET_COUT are redundant. They are replaced in NODEN_HDL's ALU by indexing of `a_c`.

- In ALU, NODEN_HDL doesn't need to construct a list for conditions.

```
[2,4,5,7,9,16,32,33]

TYPE alu_op = NEW word7 ( alu_and | alu_rmb | alu_0   | alu_m   | alu_com |
                      alu_r   | alu_sr  | alu_xor | alu_nor | alu_sl  |
                      alu_add | alu_sub | alu_inc
                     ).


\ **** The ALU in NODEN **** \

FN NOT_32 = (word32: a  ) -> word32: [FOR k = 1 TO 32] NOT  a[k].
FN  OR_32 = (word32: a b) -> word32: [FOR k = 1 TO 32] OR  (a[k],b[k]).
FN AND_32 = (word32: a b) -> word32: [FOR k = 1 TO 32] AND (a[k],b[k]).

FN FIND_SR = (bool: cin bflag r31) -> bool: IF cin THEN r31 ELSE bflag FI.

FN ADD32BIT = (word32: rin tin, bool: cin) -> (word32,bool):
  BEGIN LET sum33 = IF cin
                    THEN WORD33(((VAL32 rin) + (VAL32 tin)) + 1)
                    ELSE WORD33 ((VAL32 rin) + (VAL32 tin))
                  FI,
          aout  = sum33[1..32],
          carry = sum33[33].
        OUTPUT (aout, carry)
    END.


FN BITOP = (word32: rin tin, bool: bflag r31 cin, alu_op: op) -> (word32,bool):
  BEGIN LET tbar = NOT_32 tin,
          rbar = NOT_32 rin,
          sr   = FIND_SR(cin, bflag, r31),
          r_xor_t = OR_32(AND_32(rin, tbar), AND_32(rbar, tin)),
          shift_right = (rin[2..32]) CONC sr.
       OUTPUT CASE op OF
                alu_and: (AND_32(rin, tin),      ?bool),
                alu_rmb: (AND_32(rin, tbar),     ?bool),
                alu_0:   ([32]f,                ?bool),
                alu_m:   (tin,                  ?bool),
                alu_com: (tbar,                 ?bool),
                alu_r:   (rin,                  ?bool),
```

```
            alu_sr:  (shift_right,          ?bool),
            alu_xor: (r_xor_t,              ?bool),
            alu_nor: (NOT_32(OR_32(rin,tin)), ?bool),
            alu_sl:  ADD32BIT(rin, rin,   cin),
            alu_add: ADD32BIT(rin, tin,   cin),
            alu_sub: ADD32BIT(rin, tbar,  cin),
            alu_inc: ADD32BIT(rin, [32]f, cin)
        ESAC
    END.


FN GET_RM31 = (bool: t31, alu_op: alucon) -> bool:
  CASE alucon OF
     alu_sub: NOT t31,
     alu_add:    t31
    ELSE ?bool
  ESAC.


\ Note lower bounds of vectors 1 (not 0) \
FN ALU = (word32: rbar treg, bool: cin bflag, alu_op: alucon) -> (word32,word9):
     BEGIN LET r = NOT_32 rbar,
             t31 = treg[32],
             rm31 = GET_RM31(t31, alucon),
             r0 = r[1],
             r30 = r[31],
             r31 = r[32],
             a_c = BITOP(r, treg, bflag, r31, cin, alucon),
             aout = a_c[1],
             aoutbar = NOT_32 aout,
             coutbar = NOT(a_c[2]),
             aout31 = aout[32],
             nzbot16 = NOT((aout[ 1..16]) == ([16]f)),
             zmid4   =    ((aout[17..20]) == ([ 4]f)),
             nztop12 = NOT((aout[21..32]) == ([12]f)),
             conditions = (aout31, r0, r30, r31, rm31, coutbar,
                           nztop12, zmid4, nzbot16).
          OUTPUT (aoutbar, conditions)
       END.
```

## B.2 ALU implementation

This is the description of the ALU as constructed from 4-bit slices used in the 'lower levels' of verification.

```
[2,4,5,7,9,16,32]

TYPE alu_op = NEW word7 ( alu_and | alu_rmb | alu_0   | alu_m   | alu_com |
                          alu_r   | alu_sr  | alu_xor | alu_nor | alu_sl  |
                          alu_add | alu_sub | alu_inc
                        ).


\ **** The ALU in NODEN **** \

FN NOT_4 = (word4: a  ) -> word4: [FOR k = 1 TO 4] NOT  a[k].
FN  OR_4 = (word4: a b) -> word4: [FOR k = 1 TO 4] OR  (a[k],b[k]).
FN AND_4 = (word4: a b) -> word4: [FOR k = 1 TO 4] AND (a[k],b[k]).


\ **** BUFFER function **** \

FN BUFFER_AC = (alu_op: alucon) -> [8]alu_op:  \generates 8 buffered copies\
   (alucon, alucon, alucon, alucon, alucon, alucon, alucon, alucon).


\ **** INV_CIN and SRSELECT, combined as a single function INVCIN_SRS **** \

FN SRSELECT = (bool: cin bflag rt31) -> bool:       \srbar\
   IF cin  THEN NOT rt31  ELSE NOT bflag  FI.


FN INVCIN_SRS = (bool: cin bflag rt31) -> [2]bool:
    (NOT cin, SRSELECT(cin, bflag, rt31)).


\ **** ZERO GATING, combined as ZERO_GATES **** \

FN ZERO_LS16 = (word4: lszeros mszeros) -> bool:    \nz_bot16\
   (NOT (lszeros == (t,t,t,t))).

FN ZERO_MID4 = (word4: lszeros mszeros) -> bool:    \ z_mid4\
   mszeros[1].
```

```
FN ZERO_MS12 = (word4: lszeros mszeros) -> bool:     \nz_top12\
   (NOT (mszeros[2..4] == (t,t,t))).


FN ZERO_GATES = (word4: lszeros mszeros) -> [3]bool:
   (ZERO_LS16(lszeros, mszeros), ZERO_MID4(lszeros, mszeros),
    ZERO_MS12(lszeros, mszeros)).


\ **** LOOK_AHEAD = combined LOOKAHEAD_n functions **** \

FN LOOKAHEAD_0 = (bool: c0bar)  ->  bool:                          \c0\
   (NOT c0bar).

FN LOOKAHEAD_1 = (bool: c0bar, [2]bool: pbg0) -> bool:              \c1\
   BEGIN LET p0bar = pbg0[1], g0 = pbg0[2],  p0 = NOT p0bar.
        LET c0 = NOT c0bar.
        OUTPUT  (g0 OR (p0 AND c0))
     END.

FN LOOKAHEAD_2 = (bool: c0bar, [2]bool: pbg0 pbg1) -> bool:         \c2\
   BEGIN LET p0bar = pbg0[1], g0 = pbg0[2], p0 = NOT p0bar.
        LET p1bar = pbg1[1], g1 = pbg1[2], p1 = NOT p1bar.
        LET c0 = NOT c0bar,  c1 = g0 OR (p0 AND c0).
        OUTPUT  (g1 OR (p1 AND c1))
     END.

FN LOOKAHEAD_3 = (bool: c0bar, [2]bool: pbg0 pbg1 pbg2) -> bool:     \c3\
   BEGIN LET p0bar = pbg0[1], g0 = pbg0[2], p0 = NOT p0bar.
        LET p1bar = pbg1[1], g1 = pbg1[2], p1 = NOT p1bar.
        LET p2bar = pbg2[1], g2 = pbg2[2], p2 = NOT p2bar.
        LET c0 = NOT c0bar,  c1 = g0 OR (p0 AND c0).
        LET c2 = g1 OR (p1 AND c1).
        OUTPUT  (g2 OR (p2 AND c2))
     END.

FN LOOKAHEAD_4 = (bool: c0bar, [2]bool: pbg0 pbg1 pbg2 pbg3) -> bool: \NOT c4\
   BEGIN LET p0bar = pbg0[1], g0 = pbg0[2], p0 = NOT p0bar.
        LET p1bar = pbg1[1], g1 = pbg1[2], p1 = NOT p1bar.
        LET p2bar = pbg2[1], g2 = pbg2[2], p2 = NOT p2bar.
        LET p3bar = pbg3[1], g3 = pbg3[2], p3 = NOT p3bar.
```

```
      LET c0 = NOT c0bar,  c1 = g0 OR (p0 AND c0).
      LET c2 = g1 OR (p1 AND c1).
      LET c3 = g2 OR (p2 AND c2).
      OUTPUT (NOT (g3 OR (p3 AND c3) ))
   END.


FN LOOK_AHEAD = (bool: c0bar, [2]bool: pbg0 pbg1 pbg2 pbg3) -> [5]bool:
  (LOOKAHEAD_0(c0bar),
   LOOKAHEAD_1(c0bar, pbg0),
   LOOKAHEAD_2(c0bar, pbg0, pbg1),
   LOOKAHEAD_3(c0bar, pbg0, pbg1, pbg2),
   LOOKAHEAD_4(c0bar, pbg0, pbg1, pbg2, pbg3)
  ).


\ **** A 4-bit slice of the ALU **** \

FN ADDER4 = (word4: r m, bool: cin) -> word4:
   IF cin
     THEN ( WORD5( (VAL4 r) + (VAL4 m) + 1) )[1..4]
     ELSE ( WORD5( (VAL4 r) + (VAL4 m)    ) )[1..4]
   FI.


FN PG4 = (word4: r m) -> (bool,bool):  \PBAR and G\
   BEGIN LET  p = OR_4 (r, m).
        LET  g = AND_4(r, m).
        LET p0 = p[1],  p1 = p[2],  p2 = p[3],  p3 = p[4].
        LET g0 = g[1],  g1 = g[2],  g2 = g[3],  g3 = g[4].
        OUTPUT(NOT (p0 AND p1 AND p2 AND p3),
              (g3 OR (p3 AND (g2 OR (p2 AND (g1 OR (p1 AND g0))))))
             )
   END.


FN ADD4BIT = (word4: r m, bool: cin) -> (word4,(bool,bool)):
   (ADDER4(r, m, cin), PG4(r, m)).


FN FOURBITOP = (word4: r m, bool: srbar cin, alu_op: op) -> (word4,(bool,bool)):
   BEGIN LET mbar = NOT_4 m,  rbar = NOT_4 r.
```

```
      LET  r0 = r[1],      r1 = r[2].
      LET  r2 = r[3],      r3 = r[4].

      LET    r_xor_m = OR_4( AND_4(r, mbar), AND_4(rbar, m)).
      LET shift_right = (r[2..4]) CONC (NOT srbar).
      LET  shift_left = (cin, r0, r1, r2).

      OUTPUT CASE op OF
              alu_and: (AND_4(r, m),     word2),       \r AND m\
              alu_rmb: (AND_4(r, mbar),  word2),       \r AND NOT m\
              alu_0:   (WORD4 0,        word2),       \zero\
              alu_m:   (m,              word2),       \m\
              alu_com: (mbar,           word2),       \NOT m\
              alu_r:   ( r,             word2),       \r\
              alu_sr:  ( shift_right,    word2),       \r SR 1\
              alu_xor: ( r_xor_m,       word2),       \r XOR m\
              alu_nor: (NOT_4(OR_4(r, m)),word2),       \r NOR m\
              alu_sl:  ( shift_left, (NOT(r == (t,t,t,t)), r3)),\SL\
              alu_add: ADD4BIT(r, m, cin),             \r + m\
              alu_sub: ADD4BIT(r, mbar, cin),          \r - m\
              alu_inc: ADD4BIT(r, (WORD4 0), cin)       \r + 1\
          ESAC
    END.


FN GET_RM31 = (bool: m31, alu_op: op) -> bool:
  CASE  op OF
     alu_sub: NOT m31,
     alu_add:    m31
    ELSE ?bool
  ESAC.


FN ALU_4BIT = (word4: r4bar t4reg, bool: srbar cin4bit, alu_op: alucon) ->
           (word2, word4, bool, bool, bool, bool, bool, bool):
  BEGIN LET      r = NOT_4 r4bar.
      LET aout_pg = FOURBITOP(r, t4reg, srbar, cin4bit, alucon).
      LET   aout = aout_pg[1].
      OUTPUT  (aout_pg[2],
             NOT_4 aout,
             (aout == (f,f,f,f)),
             GET_RM31(t4reg[4], alucon),
```

```
              r[1],
              r[3],
              r[4],
              aout[4]
             )
    END.


\ **** Assembly of four ALU_4BIT's to form an ALU_16BIT **** \

FN ALU_16BIT = (word16: r16bar t16reg, bool: srbar cin16bar,
             alu_op: alucn0 alucn1 alucn2 alucn3) ->
            (word16,word4,bool,bool,bool,bool,bool,bool):
\delivers aoutbar, zout, c16outbar, rm31, rt0, rt30, rt31, aout31\
  BEGIN LET rb00_03 = r16bar[ 1.. 4], rb04_07 = r16bar[ 5.. 8].
       LET rb08_11 = r16bar[ 9..12], rb12_15 = r16bar[13..16].
       LET  m00_03 = t16reg[ 1.. 4], m04_07 = t16reg[ 5.. 8].
       LET  m08_11 = t16reg[ 9..12], m12_15 = t16reg[13..16].
       LET  rbar04 = r16bar[5],     rbar08 = r16bar[9].
       LET  rbar12 = r16bar[13].

       MAKE LOOK_AHEAD: lookahead.
           LET   c0 = lookahead[1],  c1 = lookahead[2].
           LET   c2 = lookahead[3],  c3 = lookahead[4].
           LET c4bar = lookahead[5].

       MAKE ALU_4BIT: alu4_00_03.
           LET      pg0 = alu4_00_03[1],  aoutbar_0 = alu4_00_03[2].
           LET   zout_0 = alu4_00_03[3],        rt0 = alu4_00_03[5].


       MAKE ALU_4BIT: alu4_04_07.
           LET      pg1 = alu4_04_07[1],  aoutbar_1 = alu4_04_07[2].
           LET   zout_1 = alu4_04_07[3].

       MAKE ALU_4BIT: alu4_08_11.
           LET      pg2 = alu4_08_11[1]   aoutbar_2 = alu4_08_11[2].
           LET   zout_2 = alu4_08_11[3].

       MAKE ALU_4BIT: alu4_12_15.
           LET      pg3 = alu4_12_15[1],  aoutbar_3 = alu4_12_15[2].
           LET   zout_3 = alu4_12_15[3],       rm31 = alu4_12_15[4].
```

```
        LET    rt30 = alu4_12_15[6],     rt31 = alu4_12_15[7].
        LET   aout31 = alu4_12_15[8].

     JOIN (cin16bar, pg0, pg1, pg2, pg3)      -> lookahead,
          (rb00_03, m00_03, rbar04, c0, alucn0) -> alu4_00_03,
          (rb04_07, m04_07, rbar08, c1, alucn1) -> alu4_04_07,
          (rb08_11, m08_11, rbar12, c2, alucn2) -> alu4_08_11,
          (rb12_15, m12_15,  srbar, c3, alucn3) -> alu4_12_15.

     OUTPUT (((aoutbar_0 CONC aoutbar_1) CONC (aoutbar_2 CONC aoutbar_3)),
            (zout_0, zout_1, zout_2, zout_3),
            c4bar, rm31, rt0, rt30, rt31, aout31
           )
 END.


\ **** Assembly of two ALU_16BIT's to form the ALU **** \

\ Note order of conditions reversed to agree with spec \
FN ALU_C = (word32: rbar treg, bool: cin bflag, alu_op: alucon)
      -> (word32,word9):
\aoutbar, nz_bot16, z_mid4, nz_top12, coutbar, rm31, rt31, rt30, rt0, aout31\
 BEGIN LET rb00_15 = rbar[ 1..16],  rb16_31 = rbar[17..32].
     LET  rbar16 = rbar[17].

     LET  m00_15 = treg[ 1..16],   m16_31 = treg[17..32].

     MAKE ALU_16BIT: alu00_15 alu16_31.
     LET aoutbar_0 = alu00_15[1],  cin_ms = alu00_15[3].
     LET   lszeros = alu00_15[2],     rt0 = alu00_15[5].

     LET aoutbar_1 = alu16_31[1], coutbar = alu16_31[3].
     LET   mszeros = alu16_31[2],    rm31 = alu16_31[4].
     LET      rt30 = alu16_31[6],    rt31 = alu16_31[7].
     LET    aout31 = alu16_31[8].

     MAKE INVCIN_SRS: cin_sr.
       LET   srbar = cin_sr[2],   cinbar = cin_sr[1].

     MAKE BUFFER_AC: ac_n.
       LET ac_0 = ac_n[1],  ac_1 = ac_n[2],  ac_2 = ac_n[3].
       LET ac_3 = ac_n[4],  ac_4 = ac_n[5],  ac_5 = ac_n[6].
```

```
        LET ac_6 = ac_n[7],  ac_7 = ac_n[8].

      MAKE ZERO_GATES: zero_g.
        LET nz_bot16 = zero_g[1],  z_mid4 = zero_g[2].
        LET nz_top12 = zero_g[3].

      JOIN (rb00_15, m00_15, rbar16, cinbar, ac_0, ac_1, ac_2, ac_3)
                                              -> alu00_15,
         (rb16_31, m16_31,  srbar, cin_ms, ac_4, ac_5, ac_6, ac_7)
                                              -> alu16_31,
         (cin, bflag, rt31) -> cin_sr,
                   alucon -> ac_n,
         (lszeros, mszeros) -> zero_g.

        LET  aoutbar = (aoutbar_0 CONC aoutbar_1).
        OUTPUT (aoutbar, (aout31, rt0, rt30, rt31, rm31, coutbar,
                       nz_top12, z_mid4, nz_bot16
                      )
               )
   END.
```

# C ALU specification in the VIPER project

Listed in this appendix are the HOL definitions of the ALU used at the electronic block level in the VIPER verification project (see [2]).

```
new_definition(`ADD32_DEF`,
  "(ADD32:word32#word32#bool->word33)(r,t,cin) =
   (cin => WORD33((VAL32 r) + (VAL32 t) + 1) |
         WORD33((VAL32 r) + (VAL32 t)))");;

new_definition(`ADD32BIT_DEF`,
  "(ADD32BIT:word32#word32#bool->word32#bool)(r,t,cin) =
   let sum33 = ADD32(r,t,cin) in
   let aout = WORD32(V(SEG(0,31)(BITS33 sum33))) in
   let carry = EL 32(BITS33 sum33) in (aout,carry)");;

new_definition(`FIND_SR_DEF`,
  "(FIND_SR:bool#bool#bool->bool)(cin,bflag,r31) =
   (cin => r31 | bflag)");;

new_definition(`BITOP_DEF`,
```

```
 "(BITOP:word32#word32#bool#bool#bool#word7->word32#bool)
  (r,t,bflag,r31,cin,op) =
  let cout_x = F in
  let tbar = NOT32 t in
  let rbar = NOT32 r in
  let sr = FIND_SR(cin,bflag,r31) in
  let r_xor_t = (r AND32 tbar) OR32 (rbar AND32 t) in
  let shift_right = WORD32(V(CONS sr(SEG(1,31)(BITS32 r)))) in
  ((op = #1101001) => ((r AND32 t),cout_x) |
   (op = #1111001) => ((r AND32 tbar),cout_x) |
   (op = #1100000) => (WORD32 0,cout_x) |
   (op = #1101000) => (t,cout_x) |
   (op = #1111000) => (tbar,cout_x) |
   (op = #1110001) => (r,cout_x) |
   (op = #1110010) => (shift_right,cout_x) |
   (op = #0111001) => (r_xor_t,cout_x) |
   (op = #0111100) => (NOT32(r OR32 t),cout_x) |
   (op = #0000101) => ADD32BIT(r,r,cin) |
   (op = #0001001) => ADD32BIT(r,t,cin) |
   (op = #0011001) => ADD32BIT(r,tbar,cin) |
   (op = #0000001) => ADD32BIT(r,WORD32 0,cin) | ARB)");;

new_definition(`GET_RM31_DEF`,
 "(GET_RM31:bool#word7->bool)(t31,alucon) =
  ((alucon = #0011001) => NOT t31 |
   (alucon = #0001001) => t31 | F)");;

new_definition(`GET_AOUT_DEF`,
 "(GET_AOUT:word32#bool->word32)(aout,cout) = aout");;

new_definition(`GET_COUT_DEF`,
 "(GET_COUT:word32#bool->bool)(aout,cout) = cout");;

new_definition(`ALU_DEF`,
 "(ALU:word32#word32#bool#bool#word7->word32#word9)
  (rbar,treg,cin,bflag,alucon) =
 (let r = NOT32 rbar in
  let t31 = EL 31(BITS32 treg) in
  let rm31 = GET_RM31(t31,alucon) in
  let r0 = EL 0(BITS32 r) in
  let r30 = EL 30(BITS32 r) in
  let r31 = EL 31(BITS32 r) in
```

```
let a_c = BITOP(r,treg,bflag,r31,cin,alucon) in
let aout = GET_AOUT a_c in
let aoutbar = NOT32 aout in
let coutbar = NOT(GET_COUT a_c) in
let aout31 = EL 31(BITS32 aout) in
let nzbot16 = NOT((V(SEG(0,15)(BITS32 aout))) = 0) in
let zmid4 = V(SEG(0,15)(BITS32 aout)) = 0 in
let nztop12 = NOT((V(SEG(20,31)(BITS32 aout))) = 0) in
let conditions = WORD9(V(CONS aout31(CONS r0(CONS r30(
             CONS r31(CONS rm31(CONS coutbar(CONS nztop12(
             CONS zmid4(CONS nzbot16 []))))))))))) in
(aoutbar,conditions))");;
```

# Index