(2)

vh

UNLIMITED

Report No. 90009

Report No. 90009

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

FORMAL SPECIFICATION OF THE VIPER MICROPROCESSOR IN HOL.

Author: C H Pygott

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

June 1990

UNLIMITED

CONDITIONS OF RELEASE

0108583 304494

************************* DRIC U

************************* DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 90009

TITLE: FORMAL SPECIFICATION OF THE VIPER MICROPROCESSOR IN HOL

AUTHOR: C H Pygott

DATE: June 1990

SUMMARY

This report provides a mathematically rigorous specification of the required behaviour of the VIPER microprocessor in the HOL notation (Higher Order Logic) of Cambridge University. This specification has been used as the starting point for a chain of proofs, in an attempt to show that a number of implementations of this specification are indeed correct.

This report replaces the early RSRE report 85013, which describes VIPER in the language LCF-LSM (a precursor to HOL).

FORMAL SPECIFICATION OF THE VIPER MICROPROCESSOR IN HOL

C H Pygott

June 1990

CONTENTS

1. INTRODUCTION

The VIPER (1, 2) microprocessor was invented at RSRE to satisfy the need for a highly trusted 32-bit computer which can be used in safety critical applications. The need for such a chip, has arisen in areas such as the arming and fuzing of weapons, "fly by wire" control systems in high performance military and civil aircraft and in the instrumentation of nuclear reactors. The majority of widely available microprocessors are regarded as unsatisfactory for safety critical applications because they have instruction sets that are too rich and that lead to programmer confusion and problems with formal verification of the software to run on them. Also, they are documented in natural language (with its inherent ambiguities) and their designs are validated by simulation (a process that cannot give a 100% guarantee of correctness). The aim of the VIPER project has therefore been to design a processor architecture well matched to critical applications, to define it rigorously (this document) and to attempt to prove by mathematical means (5,6) the correctness of circuits designed to meet this specification (in addition to their conventional validation by simulation).

This Report specifies the VIPER architecture in two ways:-

a) Informally, using a conventional description of the instruction set

b) Formally, using the notation of the HOL system (Higher Order Logic),(3) developed at the University of Cambridge

The purpose of this Report is to provide an unambiguous description of the functional behaviour of VIPER. This is usually referred to as the top-level specification and corresponds to the programmer's view of the processor. This view excludes such considerations as electrical properties and detailed timings, which can be found in reference 2. A number of VLSI technologies have been used to realise machines which respect the specification given in this document, using in the region of 4000-5000 logic array cells (which illustrates t e inherent simplicity of the architecture).

This report replaces the earlier RSRE report 85013 (9) which provided the formal description of VIPER in the LCF-LSM language (4). This was the precursor of HOL, but as the proofs of correctness to show the validity of VIPER's major state and block level implementations (5,6) were done in HOL, this new specification should be regarded as the definitive description. There are only minor syntactic differences between LCF-LSM and the sub-set of the HOL logic used here, so this description and the previous report are almost identical. However, there is one substantive change in the function NEXT. This will be described in the appropriate section. This HOL description was derived from the earlier LCF-LSM description by Dr Avra Cohn of Cambridge University.

2. INFORMAL DESCRIPTION OF ARCHITECTURE

The rationale for the VIPER architecture is given in reference 1. As shown in Fig 1, the conceptual machine has an accumulator, A, of 32-bits, two index registers, X and Y also of 32-bits and a register for the program counter P, of width 20-bits. VIPER's main memory is 1 Mega-word. This is used as the source of all instructions and the source and destination of most data operations. However VIPER has a separate 1 Mega-word memory space available for peripheral devices, which is only accessed by the INPUT and OUTPUT instructions. In both cases the data path is 32-bits wide. Selection between the domain of the main memory and the input/output space is achieved by a 1-bit signal. From the point of view of this specification, all of the main memory and the input/output space can be viewed as Random Access Memory (RAM) addressed by 21-bits in total, i.e. a memory/io control bit concatenated with a 20-bit address generated by the rest of the machine.

In addition to the above registers, the architecture has a single 1-bit flag register, B, which holds the results of comparison instructions and carry bits from arithmetic or shift operations. The final key feature is a single Boolean, STOP, which becomes true if any logical error occurs in the execution of a program in VIPER, such as arithmetic overflow or generation of an offset address larger than 20-bits. In such circumstances the machine must halt. If the real time application requires continued operation, this must be achieved by external means, such as redundant processing capability or by switching to an alternative program.

From the two paragraphs above it will be seen that a 'state' in which the machine rests momentarily between instructions can be written down as the vector:-

(RAM, P, A, X, Y, B, STOP)

where P, A, X, Y, B imply the current contents of these registers and STOP the current setting of the stop condition. The element RAM implies the current contents of both address spaces. The essence of the specification given in this document is to define rigorously all transitions from one such state to another for all possible instructions stored in the main memory.

Instructions are stored as groups of fields occupying the highest 12-bits of each word. The remaining 20-bits represent either an address or a constant.

Throughout this specification the bits of a word are numbered from 0 at the least significant end. For the purposes of definition the machine is assumed to have an Arithmetic and Logic Unit (ALU) with two 32-bit inputs, denoted by convention as R and M, but these values are not directly accessible to the user and are not part of the primary state of the VIPER machine.

It should be noted that events such as reset which are caused by the 'environment' in which VIPER is operating are regarded as being outside the programmer's view of normal operation, and so are not formally defined in this document. Informally, a reset may occur at any time and causes all the registers (A, X, Y, P, B and STOP) to be cleared. The other similar event that may occur is a forced error, when for example hardware external to the processor detects a parity fault in the memory and forces the processor into the stopped state. The effect of a forced error is to set the STOP flag, thus preventing further instructions being executed, until cleared by reset.

The fields formed from the 32-bits of each VIPER instruction can be defined as shown in Table 1:-

TABLE 1. INSTRUCTION DECODING

Field	Identifier	High Bit	Low Bit	Length Bits	Defining
Register select	rsf	31	30	2	READ: source of R input to ALU WRITE: register to be written
Memory select	msf	29	28	2	READ: source of M input to ALU WRITE: memory or io address SHIFT: shift op
Destination select	dsf	27	25	3	Destination of result
Comparison select	csf	24	24	1	1=compare 0=arithmetic op
Function select	fsf	23	20	4	Comparison or ALU function
Address	addr	19	0	20	Address or 20 bit constant

The next level of decoding is illustrated in the following tables, which indicate the coding of each field. The R input of the ALU or the register to be written into RAM is selected by the Register Select Field as shown in Table 2:-

TABLE 2. REGISTER SELECT FIELD

Value of rsf	R input to ALU or value to be written to memory
0	A
1	X
2	Y
3	P, padded to 32-bits with leading zeros

The memory select field has several roles, depending on the values of the other fields of the instruction, these are illustrated in Table 3:-

Case 1: (csf = 0) AND (dsf > 5) is a WRITE operation, msf indicates the address

Case 2: (csf = 1) OR ((dsf <= 5) AND (fsf /= 12)) is a comparison or arithmetic operation, with either an operand being read from memory or the 20-bit tail of the instruction being allocated as a constant to the M input of the ALU

Case 3: (csf = 0) AND ((dsf <= 5) AND (fsf = 12)) is a shift operation, with msf defining which of the four possible shift instructions is to be performed

TABLE 3. MEMORY SELECT FIELD

msf	(csf = 0) AND (dsf > 5)	(csf = 1) OR ((dsf <= 5) AND (fsf /= 12))	(csf = 0) AND (dsf <= 5) AND (fsf = 12)
0	Illegal stop = TRUE	Assign constant Padded to 32 bits to M	No WRITE or READ. Defines one of four shift instructions, as listed in Table 6
1	Write source (rsf) to addr	Read from addr and assign result to M	
2	IF (addr + X) is <= 20-bits write to this location ELSE stop	IF (addr + X) is <= 20 bits read from this location and assign result to M, ELSE stop	
3	IF (addr + Y) is <= 20-bits write to this location ELSE stop	IF (addr + Y) is <= 20 bits read from this location and assign result to M, ELSE stop	

As indicated above, the destination select field controls the read/write operations, subject to conditionals such as indexed addresses being within the 20-bit range of the machine. The definition of the coding of dsf is given in Tables 4A & 4B, in which the values of the predicates are indicated by 1, 0 or X (for either value). Each column in such a table defines a combination of conditions and the actions which must be performed if these circumstances are encountered in the execution of a program. The 'actions' in this specification of VIPER are assignments to the elements of the state vector (RAM, A, X, Y, P, B, STOP).

TABLE 4A. DESTINATION SELECT FIELD LOGIC

Column number	1	2	3	4	5	6	7
STOP	1	0	0	0	0	0	0
invalid address or illegal operation, excluding illegal calls (see col 16)	X	1	0	0	0	0	0
compare (csf = 1)	X	X	1	0	0	0	0
value of dsf	any	any	any	7,6	5	5	5
b	X	X	X	X	1	0	0
call (fsf = 1)	X	X	X	X	X	1	0
CHANGES VALUE OF:-							
RAM (memory + i/o)	-	-	-	regval	-	-	-
A	-	-	-	-	-	-	-
X	-	-	-	-	-	-	-
Y	-	-	-	-	-	P+1	-
P	-	P + 1	P + 1	P + 1	P + 1	RES	RES
B	-	-	comparison	-	-	BVAL	BVAL
STOP	-	TRUE	-	-	-	SVAL	SVAL

Note: The ALU is specified (table 6) to deliver a triple (RES, BVAL, SVAL), where RES is a 32-bit answer, Boolean BVAL is the new assignment to register B and Boolean SVAL is the new value of the STOP flag.

Notes on each column, with (RES, BVAL, SVAL) delivered by ALU:-

1. Processor has halted
2. Invalid address or illegal operation, which must cause processor to halt.
3. Comparison functions, see Table 5 for which function is required.
4. Write to memory (dsf=7) or io (dsf=6)
   * regval is the contents of the register defined by rsf written into the RAM
5. No operation, (dsf=5) AND B
6. Conditional CALL,
   * P loaded with bottom 20-bits of RES and Y loaded with P+1 padded to 32-bits
7. Conditional GOTO,
   * P loaded with bottom 20-bits of RES

TABLE 4B. DESTINATION SELECT FIELD LOGIC

Column number	8	9	10	11	12	13	14	15	16
STOP	0	0	0	0	0	0	0	0	0
invalid address or illegal operation, excluding illegal calls (see col 16)	0	0	0	0	0	0	0	0	0
compare (csf = 1)	0	0	0	0	0	0	0	0	0
value of dsf	4	4	4	3	3	2	1	0	<3
b	0	1	1	X	X	X	X	X	X
call (fsf = 1)	X	1	0	1	0	0	0	0	1
CHANGES VALUE OF:-									
RAM (memory + i/o)	-	-	-	-	-	-	-	-	-
A	-	-	-	-	-	-	-	RES	-
X	-	-	-	-	-	-	RES	-	-
Y	-	P+1	-	P+1	-	RES	-	-	P+1
P	P+1	RES	RES	RES	RES	P+1	P+1	P+1	P+1
B	-	BVAL	BVAL	BVAL	BVAL	BVAL	BVAL	BVAL	-
STOP	-	SVAL	SVAL	SVAL	SVAL	SVAL	SVAL	SVAL	TRUE

Notes on columns.
8. No operation, (dsf=4) AND (NOT B)
9. Conditional CALL
   * P loaded with bottom 20-bits of RES, Y loaded with P+1 padded to 32-bits
1). Conditional GOTO: * P loaded with bottom 20-bits of RES
11. Unconditional CALL
   * P loaded with bottom 20-bits of RES, Y loaded with P+1 padded to 32-bits
12. Unconditional GOTO: * P loaded with bottom 20-bits of RES
13..15 Assignments to Y, X or A
16. Illegal CALL instructions

TABLE 5. COMPARISON FUNCTIONS

fsf	comparator
0	R < M
1	R >= M
2	R = M
3	R /= M
4	R <= M
5	R > M
6	unsigned R < M
7	unsigned R >= M
8 to 15	As above, but with the the result ORed with B eg: 8 => (R < M) OR B

The description of the arithmetic and logic functions of the ALU calls for two further definitions to describe the error conditions:-

pwrite a Boolean which is TRUE if the destination of the result of the operation is the P register i.e. value of dsf = 3, 4 or 5. Many of the ALU operations cannot be used for manipulating the program counter, since potentially dangerous effects could be produced. Hence pwrite is the error condition for "barred on P register" (note that the CALL instruction can only be used with destination P).

INVALID An operator applied to the 32-bit result of an ALU operation which delivers TRUE if this value has any one of the top 12-bits set i.e. represents an impossible address or value of the program counter

The following tables also contain the values 'carry', 'borrow' and 'overflow', which are defined later.

TABLE 6A. ALU FUNCTIONS 0 - 11

fsf	msf	function	output		
			RES	BVAL	SVAL
0	any	NEGATE m	NOT M	B	pwrite
1	any	CALL m	M	B	NOT pwrite OR INVALID M
2	any	READ from peripheral	M	B	pwrite
3	any	READ from memory	M	B	pwrite AND INVALID M
4	any	ADD, no overflow detected	R + M	carry	pwrite
5	any	ADD, stop on overflow	R + M	B	overflow OR (pwrite AND INVALID(R+M))
6	any	SUB, no overflow detected	R - M	borrow	pwrite
7	any	SUB, stop on overflow	R - M	B	overflow OR (pwrite AND INVALID(R-M))
8	any	XOR	R XOR M	B	pwrite
9	any	AND	R AND M	B	pwrite
10	any	NOR	R NOR M	B	pwrite
11	any	AND NOT	R AND NOT M	B	pwrite

TABLE 6B. ALU FUNCTIONS 12 - 15

fsf	msf	function	output		
			RES	BVAL	SVAL
12	0	SHIFT RIGHT copy sign bit	R31.R31..R1	B	pvrite
12	1	SHIFT RIGHT through B	B.R31..R1	R0	pvrite
12	2	SHIFT LEFT stop on overflov	R + R	B	pvrite OR overflov
12	3	SHIFT LEFT through B	R30..R0.B	R31	pvrite
13	any	Illegal	R	B	TRUE
14	any	Illegal	R	B	TRUE
15	any	Illegal	R	B	TRUE

Note:-
In the notation used for the shifts, Rm..Rn denotes a slice of the bits in the R input to the ALU and . (full stop) denotes concatenation.

3. FORMAL SPECIFICATION IN HOL

The HOL system has been devised by the Computing Laboratory of the University of Cambridge, using the interactive programming language ML (Meta Language), and is a development of LCF-LSM. The origins of this vork are described in a book by Gordon, Milner and Vadsvorth (7).

In this Section VIPER is specified in the style proposed by Gordon (8), using the primitive functions defined in the present Cambridge HOL system. Annex A gives a very brief introduction to the HOL constructs used and the reader should turn to those pages next to gain an initial understanding.

The formal specification is presented on the folloving pairs of pages, each page of HOL text having a facing page of commentary. Inevitably any such specification needs a number of auxiliary functions, to enable the primary axiom for the next state of the machine to be defined in a concise manner.

The specification of VIPER begins with two declarations, to create types for words of fixed numbers of bits and to create a hypothetical address space:-

```
declare_word_widths[1;2;3;4;20;21;32;33;34]
declare_memories[(21,32)]
```

This introduces the types word1, word2, ...word34 and the standard functions for converting these types to positive integers (of type num) and to lists of booleans:-

```
VAL1,  VAL2,  VAL3  ......VAL34     wordn -> num
WORD1, WORD2, WORD3 ......WORD34    num   -> wordn
BITS1, BITS2, BITS3 ......BITS34    wordn -> bool_list
NOT1,  NOT2,  NOT3  ......NOT34     wordn -> wordn
```

Writing to and reading from the address space created by declare_memories is achieved using the pair of functions:-

```
STORE21: word21 -> word32 -> mem21_32 -> mem21_32
FETCH21: mem21_32 -> word21 -> word32
```

The first functions VALUE, CARRY, OFLO, BVAL and SVAL exist solely to extract a single field from a structured value.

A number of conversions between words of differing lengths are required and this is the role of the three functions TRIM32TO20, TRIM34TO32 and PAD20TO32. The trim functions make use of the concept of lists and HOL functions such as SEG, EL, V and TL (see Annex A). As defined, trimming is 'blind' in the sense that no checks are performed to see if significant bits are lost in the trimming.

SIGNEXT performs sign extension, ie increases the length of a word by duplicating the most significant bit. Much use of this will be made in the later definition of arithmetic operations.

RIGHT and LEFT shift a word W in the appropriate direction, losing the right/left most bit and adding B as the left/right most bit.

RIGHTARITH provides a divide by two operation for a 2's complement value. That is, the value is shifted one place right with the most significant bit being duplicated.

```
declare_word_widths[1;2;3;4;20;21;32;33;34]
declare_memories[(21,32)]


   VALUE: word32£bool£bool -> word32
|- VALUE (result,carry,overflow) = result


   CARRY: word32£bool£bool -> bool
|- CARRY (result,carry,overflow) = carry


   OFLO:  word32£bool£bool -> bool
|- OFLO  (result,carry,overflow) = overflow


   BVAL: word32£bool£bool -> bool
|- BVAL (result, b, abort) = b


   SVAL: word32£bool£bool -> bool
|- SVAL (result, b, abort) = abort


   TRIM32TO20: word32 -> word20
|- TRIM32TO20  w = WORD20(V(SEG (0,19) (BITS3? w)))


   TRIM34TO32: word34 -> word32
|- TRIM34TO32  w = WORD32(V(TL(TL(BITS34 w))))


   PAD20TO32: word20 -> word32
|- PAD20TO32  w = WORD32(VAL20 w)


   SIGNEXT: word32 -> word33
|- SIGNEXT  w =
   (let bitlist = BITS32 w in WORD33(V(CONS(EL 31 bitlist) bitlist)))


   RIGHT: bool£word32 -> word32
|- RIGHT (b,r) = WORD32(V(CONS b (SEG (1,31) (BITS32 r))))


   LEFT: word32£bool -> word32
|- LEFT (r,b) = (let twice = V(TL(BITS32 r)) in
                 (b => WORD32(twice + twice) + 1 | WORD32(twice + twice))


   RIGHTARITH: word32 -> word32
|- RIGHTARITH  r = (let sign = EL 31 (BITS32 r) in
                    WORD32(V(CONS sign (SEG (1,31) (BITS32 r)))))
```

NEG provides a negate function for a 33-bit 2's complement value. It uses the usual invert and add 1 algorithm. Note that 0 is treated as a special case. If this were not removed by the initial test, 0 inverted and incremented would deliver a 34-bit result, but the use of NEG in SUB32 and COMPARE is such that only a 33-bit value is required.

ADD32 and SUB32 are the addition and subtraction operations for 32-bit values. Both deliver triples, a 32-bit result, a carry (or borrow) value and an overflow condition. The 32-bit result is the bottom 32-bits of the result of adding/subtracting the two operands regardless of whether they are 2's complement or unsigned values. The carry (or borrow for subtraction) value is used during unsigned operations only, whilst the overflow value is only significant during 2's complement arithmetic. An informal definition of overflow is that it is true if and only if the sum (or difference) of the two operands cannot be represented by a 32-bit 2's complement value. Similarly, carry or borrow are defined to be true if the sum (or difference) of the two operands cannot be represented by a 32-bit unsigned value. The relationship between these informal definitions of overflow and carry and their formal HOL descriptions is investigated in Annex B.

Given a definition of ADD32, the function to increment the program counter P, INCP32 is as shown opposite. A 32-bit value is delivered to cope with the situation when the last instruction was fetched from the top word in memory, leading to P overflowing into the 21st bit. By delivering a 32-bit value and checking that the top 12-bits are zero, it is possible to detect this unusual, but fatal condition. This check is done using the function INVALID, defined later.

The COMPARE function follows from Table 5. The values of 'dif' and 'borrow' are the same as delivered by SUB32 (although borrow is expressed slightly differently). 'Less' examines the most significant bit of the difference of the (sign extended) operands R and M. This is the sign of the result of R-M, and it is set if R is less than M. Again, this informal definition and its formal counterpart are investigated in Annex B.

```
   NEG: vord33 -> num
|- NEG  m = ((VAL33 m = 0) => 0 | (VAL33(NOT33 m) + 1)


   ADD32: vord32£vord32 -> vord32£bool£bool
|- ADD32 (r,m) =
  (let sum      = WORD34((VAL33(SIGNEXT r)) + (VAL33(SIGNEXT m))) in
   let opposite = (EL 31 (BITS32 r)) XOR (EL 31 (BITS32 m)) in
   TRIM34TO32 sum, (EL 32 (BITS34 sum)) XOR opposite,
                   (EL 32 (BITS34 sum)) XOR (EL 31 (BITS34 sum)))


   SUB32: vord32£vord32 -> vord32£bool£bool
|- SUB32 (r,m) =
  (let dif      = WORD34((VAL33(SIGNEXT r)) + (NEG(SIGNEXT m))) in
   let opposite = (EL 31 (BITS32 r)) XOR (EL 31 (BITS32 m)) in
   TRIM34TO32 dif, (EL 32 (BITS34 dif)) XOR opposite,
                   (EL 32 (BITS34 dif)) XOR (EL 31 (BITS34 dif)))


   INCP32: vord20 -> vord32
|- INCP32   p = VALUE(ADD32(PAD20TO32 p, WORD32 1))


   COMPARE: vord4£vord32£vord32£bool -> bool
|- COMPARE (fsf,r,m,b) =
  (let op     = VAL4 fsf in
   let dif    = WORD34((VAL33(SIGNEXT r)) + (NEG(SIGNEXT m))) in
   let equal  = r = m in
   let less   =  EL 32 (BITS34 dif) in
   let borrov =  (EL 32 (BITS34 dif)) XOR
                ((EL 31 (BITS32 r)) XOR (EL 31(BITS32 m))) in
    ((op =  0) => less                   |
    ((op =  1) => NOT less               |
    ((op =  2) => equal                  |
    ((op =  3) => NOT equal              |
    ((op =  4) => less OR equal          |
    ((op =  5) => NOT(less OR equal)     |
    ((op =  6) => borrov                 |
    ((op =  7) => NOT borrov              |
    ((op =  8) => less OR b                     |
    ((op =  9) => (NOT less) OR b               |
    ((op = 10) => equal OR b                    |
    ((op = 11) => (NOT equal) OR b              |
    ((op = 12) => (less OR equal) OR b          |
    ((op = 13) => (NOT(less OR equal)) OR b     |
    ((op = 14) => borrov OR b                   |
                 (NOT borrov) OR b)))))))))))))))
```

The next group of auxiliary functions is concerned with the VIPER architecture itself, rather than with manipulation of words and rows of booleans. From Table 2 it is clear that the R input to the ALU can be defined by the HOL function REG given opposite.

Generation of addresses for writing and reading is performed using the function OFFSET, to generate a 32-bit value which is checked by the predicate INVALID to make sure that none of the top 12-bits are set. If INVALID delivers FALSE it is certain that the value in question can be trimmed safely back to 20 bits and then used as a memory or input/output address. The expression:-

INVALID(OFFSET(msf,addr,x,y))

is used in the rest of the description for checking addresses in the VIPER high-level specification. Note that addition of a positive offset to a negative value in X or Y, generating a non-negative result, is perfectly legal.

Fetching instructions from main memory involves padding the 20-bit value of P with a leading zero and using the resulting 21-bit argument in the function INSTFETCH. This concatenation is achieved readily using the list constructor CONS.

Writing to and reading from the two contiguous address spaces involves the introduction of the boolean variable "io", which models the one-bit signal controlling the division between main memory and the input/output space. As will be seen from both MEMREAD and MEMWRITE this is regarded as an extra bit to be concatenated with the 20-bit address generated by the rest of the machine, to perform accesses to a 21-bit regime. These functions MEMREAD and MEMWRITE assume that the address generated by OFFSET(msf, addr, x, y) is valid, i.e. that INVALID delivers FALSE. The validity of this assumption is guaranteed by the use of INVALID to trap illegal addresses before MEMREAD and MEMWRITE are invoked (see NEXT). The generation of the M input to the ALU using MEMREAD involves one extra factor. If a shift instruction is invoked, (dsf <= 5 and fsf = 12), there is no read required, since the operation is on the R input only. In these circumstances, with the boolean variable "nil" set to TRUE, the M input is defined to be a 32-bit representation of zero. Also notice if msf=0 in MEMREAD, the value of the M input of the ALU is the addr field of the instruction padded to 32-bits. In MEMWRITE, msf=0 is illegal (and will actually be trapped in NEXT), so doesn't change the contents of RAM.

```
   REG: word2£word32£word32£word32£word20 -> word32
|- REG (rsf,a,x,y,p) =
  (let r = VAL2 rsf in
   ((r = 0) => a | ((r = 1) => x | ((r = 2) => y | PAD20TO32 p))))


   OFFSET: word2£word20£word32£word32 -> word32
|- OFFSET (msf,addr,x,y)
  (let mf     = VAL2 msf in
   let addr32 = PAD20TO32 addr in
   ((mf = 0) => addr32 |
   ((mf = 1) => addr32 |
   ((mf = 2) => VALUE(ADD32(addr32, x)) |
                VALUE(ADD32(addr32, y))))))


   INVALID: word32 -> bool
|- INVALID  value = NOT(value = PAD20TO32(TRIM32TO20 value))


   INSTFETCH: mem21_32£word20 -> word32
|- INSTFETCH (ram,p) = FETCH21 ram (WORD21(V(CONS F (BITS20 p))))


   MEMREAD: mem21_32£word2£word20£word32£word32£bool£bool -> word32
|- MEMREAD (ram,msf,addr,x,y,io,nil) =
  (let m = VAL2 msf in
   ( nil    => WORD32 0       |
   ((m = 0) => PAD20TO32 addr |
               FETCH21 ram
              (WORD21(V(CONS io (BITS20(TRIM32TO20(OFFSET(msf,addr,x,y)))))))))


   MEMWRITE: mem21_32£word32£word2£word20£word32£word32£bool -> mem21_32
|- MEMWRITE (ram,source,msf,addr,x,y,io) =
  (let m = VAL2 msf in
   ((m = 0) => ram |
               STORE21
                (WORD21(V(CONS io (BITS20(TRIM32TO20(OFFSET(msf,addr,x,y)))))))
                source ram))
```

The function for the ALU remains to be declared before moving to the definition of the permissible state transitions for VIPER. The ALU delivers a triple consisting of a 32-bit result, the next state of the B flag and a value for the STOP condition flag. As can be seen from the facing page, the ALU is very simple in concept, the most obvious feature being that most operations are barred on the P register. Only addition and subtraction with overflow protection, CALL instructions and reads from memory or manifest constants can be used to define the new contents of the P register. The definition of the ALU follows Table 6 in a natural manner.

```
   ALU: word4£word2£word3£word32£word32£bool -> word32£bool£bool
|- ALU (fsf,msf,dsf,r,m,b) =
   (let ff     = VAL4 fsf in
    let mf     = VAL2 msf in
    let df     = VAL3 dsf in
    let pwrite = (df = 3) OR ((df = 4) OR (df = 5)) in
     ((ff =  0) => (NOT32 m, b, pwrite) |
     ((ff =  1) => (m, b, (NOT pwrite) OR (INVALID m)) |
     ((ff =  2) => (m, b, pwrite) |
     ((ff =  3) => (m, b, pwrite AND (INVALID m)) |
     ((ff =  4) => let sum = ADD32(r,m) in
                   VALUE sum, CARRY sum, pwrite |
     ((ff =  5) => let sum = ADD32(r,m) in
                   VALUE sum,b, (OFLO sum) OR (pwrite AND (INVALID(VALUE sum)))|
     ((ff =  6) => let dif = SUB32(r,m) in
                   VALUE dif, CARRY dif, pwrite |
     ((ff =  7) => let dif = SUB32(r,m) in
                   VALUE dif,b, (OFLO dif) OR (pwrite AND (INVALID(VALUE dif)))|
     ((ff =  8) => ((r OR32 m) AND32 (NOT32(r AND32 m)), b, pwrite) |
     ((ff =  9) => (r AND32 m, b, pwrite) |
     ((ff = 10) => (NOT32(r OR32 m), b, pwrite)  |
     ((ff = 11) => (r AND32 (NOT32 m), b, pwrite) |
     ((ff = 12) => ((mf = 0) => (RIGHTARITH r, b, pwrite) |
                   ((mf = 1) => (RIGHT(b,r), EL 0 (BITS32 r), pwrite) |
                   ((mf = 2) => let double = ADD32(r,r) in
                                VALUE double, b, (OFLO double) OR pwrite |
                                (LEFT(r,b), EL 31(BITS32 r), pwrite))))  |
     ((ff = 13) => (r,b,T) |
     ((ff = 14) => (r,b,T) |
                   (r,b,T))))))))))))))))
```

To write a concise statement of all permissible transitions in VIPER, it is convenient in the HOL text to define a number of primary predicates derived from the fields of the current instruction and the current value of B:-

WRITE which is TRUE if the instruction involves writing to the main memory or the peripheral space

NILM which is TRUE if no M input is required to the ALU

NOOP which is TR"E if no operation is to be performed, ie SKIP

SPAREFUNC which becomes TRUE if any attempt is made to use ALU functions 13, 14 or 15

ILLEGALCALL which becomes TRUE if an illegal CALL instruction is attempted (with the destination defined as the A, X or Y registers)

ILLEGALPDEST which becomes TRUE if the destination is given as P but the specified function is an illegal way of deriving a new value of the program counter

ILLEGALWRITE which is TRUE if a WRITE instruction is attempted with the memory select field equal to 0

OUTPUT which is TRUE if data is to be written to an address in the IO space, ie NOT a comparison and df = 6

INPUT which is TRUE if data is being read from an address in the IO space, ie NOT a comparison, df <=5 and ff = 2.

```
   WRITE: word3£word1 -> bool
|- WRITE (dsf,csf) =
  (let df = VAL3 dsf in
   let cf = VAL1 csf in
    (cf = 0) AND ((df = 7) OR (df = 6)))

   NILM: word3£word1£word4 -> bool
|- NILM (dsf,csf,fsf) =
   (let df = VAL3 dsf in
    let cf = VAL1 csf in
    let ff = VAL4 fsf in
     (cf = 0) AND ((NOT((df = 7) OR (df = 6))) AND (ff = 12)))

   NOOP: word3£word1£bool -> bool
|- NOOP (dsf,csf,b) =
  (let df = VAL3 dsf in
   let cf = VAL1 csf in
    (cf = 0) AND (((df = 5) AND b) OR ((df = 4) AND (NOT b))))

   SPAREFUNC: word3£word1£word4 -> bool
|- SPAREFUNC (dsf,csf,fsf) =
  (let df = VAL3 dsf in
   let cf = VAL1 csf in
   let ff = VAL4 fsf in
    (cf=0) AND ((NOT((df=6) OR (df=7))) AND ((ff=13) OR ((ff=14) OR (ff=15)))))

   ILLEGALCALL: word3£word1£word4 -> bool
|- ILLEGALCALL (dsf,csf,fsf) =
  (let df = VAL3 dsf in
   let cf = VAL1 csf in
   let ff = VAL4 fsf in
    (cf = 0) AND ((ff = 1) AND ((df = 0) OR ((df = 1) OR (df = 2)))))

   ILLEGALPDEST: word3£word1£word4 -> bool
|- ILLEGALPDEST (dsf,csf,fsf) =
  (let df = VAL3 dsf in
   let cf = VAL1 csf in
   let ff = VAL4 fsf in
    (cf = 0) AND (((df = 3) OR ((df = 4) OR (df = 5))) AND
                 (NOT((ff = 1) OR ((ff = 3) OR ((ff = 5) OR (ff = 7)))))))

   ILLEGALWRITE: word3£word1£word2 -> bool
|- ILLEGALWRITE (dsf,csf,msf) =
  (let mf = VAL2 msf in (WRITE(dsf,csf)) AND (mf = 0))

   OUTPUT: word3£word1 -> bool
|- OUTPUT (dsf,csf) =
  (let df = VAL3 dsf in
   let cf = VAL1 csf in (cf = 0) AND (df = 6))

   INPUT: word3£word1£word4 -> bool
|- INPUT (dsf,csf,fsf) =
  (let df = VAL3 dsf in
   let cf = VAL1 csf in
   let ff = VAL4 fsf in
    (cf = 0) AND ((NOT((df = 7) OR (df = 6))) AND (ff = 2)))
```

VIPER must obey the transitions defined in the function NEXT on the opposite page. Table 4 gives the details of the new states to be achieved. As can be seen from the definition of NEXT, precise descriptions of the conditions in which the "io" signal is TRUE and for detection of invalid addresses are found in the HOL text and provide a rigorous definition of the looser statements in Section 2.

One issue which was not dealt with at all in the informal description of Section 2 is the problem of overflow of the program counter. If an instruction has been fetched from the top word of the main memory, it follows that the next increment of the program counter will cause an illegal value to be generated for P unless this last instruction is GOTO. Notice that if the instruction fetched from the top word is CALL, difficulties may be encountered later in the execution of the program, because an illegal return link will have been stored in the Y register. In view of the complexity this could introduce, any instruction in the top word of memory is illegal in VIPER and if encountered stops the processor.

The function NEXT contains the one substantive change between this report and Report 85013 (9). The expression "AND (NOT skip)" has been added to the definition of "illegaladdr". The reason for this is that, when the previous top-level specification (9) was compared with the first level of decomposition (the microprogram model described in reference 5), it was discovered that they differed when a conditional call or goto instruction delivered an illegal new value for the program counter. In the original description (9), the illegal result was detected before the B flag was examined to see if the instruction was to be performed. This led to the processor always stopping. The implementation (5) examined the B flag first and only generated the new value of the program counter (and hence only stopped if it was illegal) if the conditional operation was to be performed. The latter more closely reflected the designers' intended response to these circumstances and so the top-level specification has been changed to reflect this new requirement.

```
   NEXT: mem21_32£word20£word32£word32£word32£bool£bool ->
         mem21_32£word20£word32£word32£word32£bool£bool
|- NEXT (ram,p,a,x,y,b,stop) =
  (let instbits    = BITS32(INSTFETCH(ram,p)) in
   let newp        = TRIM32TO20(INCP32 p) in
   let rsf         = WORD2(V(SEG (30,31) instbits)) in
   let msf         = WORD2(V(SEG (28,29) instbits)) in
   let dsf         = WORD3(V(SEG (25,27) instbits)) in
   let csf         = WORD1(V(SEG (24,24) instbits)) in
   let fsf         = WORD4(V(SEG (20,23) instbits)) in
   let addr        = WORD20(V(SEG (0,19) instbits)) in
   let df          = VAL3 dsf in
   let cf          = VAL1 csf in
   let ff          = VAL4 fsf in
   let comp        = cf = 1 in
   let call        = (cf = 0) AND (ff = 1) in
   let output      = OUTPUT(dsf,csf) in
   let input       = INPUT(dsf,csf,fsf) in
   let io          = output OR input in
   let writeop     = WRITE(dsf,csf) in
   let skip        = NOOP(dsf,csf,b) in
   let noinc       = INVALID(INCP32 p) in
   let illegaladdr = (NOT(NILM(dsf,csf,fsf))) AND
                     ((INVALID(OFFSET(msf,addr,x,y))) AND (NOT skip)) in
   let illegalcl   = ILLEGALCALL(dsf,csf,fsf) in
   let illegalsp   = SPAREFUNC(dsf,csf,fsf) in
   let illegalonp  = ILLEGALPDEST(dsf,csf,fsf) in
   let illegalwr   = ILLEGALWRITE(dsf,csf,msf) in
   let source      = REG(rsf,a,x,y,newp) in

    ( stop    => (ram, p, a, x, y, b, T) |

    ((noinc OR illegaladdr) OR ((illegalcl OR illegalsp) OR
     (illegalonp OR illegalwr)) => (ram, newp, a, x, y, b, T) |

    ( comp    => (ram, newp, a, x, y,
                  COMPARE(fsf,source,MEMREAD(ram,msf,addr,x,y,io,F), b), F) |

    ( writeop => (MEMWRITE(ram,source,msf,addr,x,y,io), newp, a, x, y, b, F) |

    ( skip    => (ram, newp, a, x, y, b, F) |

     let m      = MEMREAD(ram,msf,addr,x,y,io,NILM(dsf,csf,fsf)) in
     let aluout = ALU(fsf,msf,dsf,source,m,b) in
      ((df = 0) => (ram, newp, VALUE aluout, x, y, BVAL aluout, SVAL aluout) |
      ((df = 1) => (ram, newp, a, VALUE aluout, y, BVAL aluout, SVAL aluout) |
      ((df = 2) => (ram, newp, a, x, VALUE aluout, BVAL aluout, SVAL aluout) |
      (call     => (ram, TRIM32TO20(VALUE aluout), a, x, INCP32 p,
                    BVAL aluout, SVAL aluout) |
                   (ram, TRIM32TO20(VALUE aluout), a, x, y,
                    BVAL aluout, SVAL aluout))))))))))
```

4. CONCLUSIONS

This document demonstrates that it is possible to write a specification for the functions of a powerful microprocessor, using simple concepts in first order logic. Experience has shown that HOL is a firm basis for the formal specification of VIPER.

5. ACKNOWLEDGEMENTS

VIPER has been developed by the High Integrity Systems Section of the Computing Divisions, by a team comprising Dr J Kershaw, Dr C H Pygott and Dr W J Cullyer. All members of the Section have contributed to this specification. Mr I F Currie and Dr J M Foster have made important contributions in suggesting formal methods for use in this environment of safety critical computing.

The author would also like to thank Dr A Cohn of Cambridge University, for her work on the VIPER proofs, and in particular for providing the HOL translation of the original LCF-LSM description.

6. REFERENCES.

1. KERSHAW, J.	"The VIPER microprocessor" RSRE REPORT 87014. November 1987
2. PYGOTT, C.H.	"Electrical, environmental and timing specification of VIPER microprocessor (issue 2)" RSRE REPORT 86006, June 1986
3. GORDON, M.J.	"HOL: a machine orientated formulation of higher-order logic" University of Cambridge Computing Laboratory Technical Report 68
4. GORDON, M.J.	"LCF-LSM" University of Cambridge Computing Laboratory Technical Report 41
5. COHN, A.	"A proof of correctness of the VIPER microprocessor: The first level" VLSI specification, verification and synthesis BIRTWISTLE G. & SABRAHMANYAM P.A.(ed), Kluwer 1987
6. COHN, A.	"Correctness properties of the VIPER block model: The second level" Current trends in hardware verification & automated deduction BIRTWISTLE G. & SABRAHMANYAM P.A.(ed), Springer-Verlag 1988
7. GORDON, M.J., MILNER, R. A., WADSWORTH, P.	"Edinburgh LCF" Lecture Notes in Computer Science, Springer-Verlag, 1979
8. GORDON, M.J.	"Proving a computer correct" University of Cambridge Computing Laboratory, Technical Report 42
9. CULLYER, W.J.	"Formal specification of the VIPER microprocessor" RSRE REPORT 85013, October 1985

Annex A: Short introduction to HOL

The material in this annex is a very brief, informal, digest of that presented by Gordon in reference 3. Hopefully it contains enough detail to enable the text of section 3 to be understood.

The description in section 3 assumes the existence of the following types:-

bool	the boolean type with members T and F
num	the non-negative integers
word<n>	a word of <n> bits (eg word1, word32 etc)
* list	a list of any other type "*" (eg bool list), the empty list is []

The description in section 3 also assumes the existence of certain operators and functions:-

=	equality between values	*£* -> bool
+	addition	num£num -> num
NOT	logical inversion	bool -> bool
OR	disjunction	bool£bool -> bool
AND	conjunction	bool£bool -> bool
XOR	exclusive OR	bool£bool -> bool
CONS	list constructor	* -> * list -> * list

This appends a value to the head of a list. Note that the form of the signature denotes a partially applied function (see 3), but for most purposes it can be regarded as being *£* list -> * list.
Note however that CONS is applied to two values 'a' and 'b' as "CONS a b", whilst a normal function 'C' would be applied as "C(a,b)"

HD	head of list	* list -> *
TL	tail of list	* list -> * list
EL	<n>th element of list	num -> * list -> *

(0 = first member, for a list of M elements EL (M-1) list = HD list)

SEG	select a slice from a list	(num£num) -> * list -> * list
V	the integer equivalent of a bool list (ie a list with M members delivers a value in the range 0 to 2**M -1)	bool list -> num
WORD<n>	converts an integer to a word<n>	num -> word<n>
VAL<n>	converts a word<n> to an integer	word<n> -> num
BITS<n>	converts a word<n> to a bool list	word<n> -> bool list

The main 'control' structure is the conditional expression:-
(a => b | c), which is read as "if a then b else c".

Annex B: VIPER arithmetic

This annex describes the arithmetic operations ADD32 and SUB32, and informally justifies the relationship between the informal descriptions of overflow, carry etc. given on page 11 and their formal ccunterparts on page 12.

Before these are considered some basic definitions are required. VIPER's (or any other computer's) integers are not the same as a mathematician's integers, in that any computer has a fixed word length whilst conceptual integers have an infinite range (actually a double infinite range, from -infinity to +infinity). In this annex, all 'computer words' of <n> bits will be regarded as positive values in the range 0 to two to the power <n> - 1. VIPER's 32-bit words can be interpreted as either a 2's complement signed value or an unsigned value. If 'pow<n>' is defined to be 2 to the power <n> (ie: pow4 = 16 etc), then an unsigned VIPER 32-bit word W has the equivalent integer range I as follows:-

```
For:-   0 <= W < pow32   then  I = W
 or:-   0 <= I < pow32   then  W = I
```

A 32-bit 2's complement VIPER word W, maps to an integer I as:-

```
For:-        0 <= W <  pow31    then  I = W
and:-    pow31 <= W <  pow32    then  I = W - pow32

 or:-        0 <= I <  pow31    then  W = I
and:-   -pow31 <= I <= -1       then  W = I + pow32
```

To avoid confusion, bit-<n> of a word will be said to correspond to the HOL statement "EL n ....". This means that the least significant bit is bit-0, rather than bit-1, but means that if a value is regarded as the sum of a series of powers of two, then bit-<n> corresponds to pow<n>.

Two theorems will be used frequently in the following discussions. If W represents a VIPER word, such that:- 0 <= W < pow<n>, then all the bits that are set in the word must be in the first <n> bits (ie bit-0 to bit-<n-1>), all other bits being clear. For example, if:- 0 <= W < 4 (pow2), then only the first two bits of the word may be set, all subsequent bits are known to be clear.

Also if:- pow<n> <= W < pow<n+1>, then bit-<n> of the word is set. For example, if:- 4 <= W < 8, then the third bit (bit-2) of the word is set.

Note that:- pow<n> + pow<n> = pow<n+1>.

1) The effect of SIGNEXT

```
   SIGNEXT: word32 -> word33
|- SIGNEXT  w =
   (let bitlist = BITS32 w in WORD33(V(CONS(EL 31 bitlist) bitlist)))
```

All the arithmetic operations work with 'sign extended' words. The effect of this function, in the realm of integers, depends upon whether the value being extended is considered as a signed or unsigned value.

1.a) Signed values: If the notional integer value is I, the VIPER word is W, and SXW is the effect of sign extension on W.

```
       0 <= I < pow31    then  W = I            and  SXW = I
  -pow31 <= I < 0        then  W = I + pow32    and  SXW = I + pow32 + pow32
```

1.b) Unsigned values: If the notional integer value is I, the VIPER word is W, and SXW is the effect of 'sign extension' on W.

```
    0 <= I < pov31   then  W = I    and  SXW = I
pov31 <= I < pov32   then  W = I    and  SXW = I + pov32
```

2) The addition function, ADD32

```
   ADD32: word32£word32 -> word32£bool£bool
|- ADD32 (r,m) =
  (let sum      = WORD34((VAL33(SIGNEXT r)) + (VAL33(SIGNEXT m))) in
   let opposite = (EL 31 (BITS32 r)) XOR (EL 31 (BITS32 m)) in
   TRIM34TO32 sum, (EL 32 (BITS34 sum)) XOR opposite,
                   (EL 32 (BITS34 sum)) XOR (EL 31 (BITS34 sum)))
```

As shown above, ADD32 delivers three values, the 32-bit sum, a carry condition and an overflow condition. The overflow condition is only of interest during 2's complement addition, whilst carry is only used by unsigned addition. These two signals will therefore be considered separately.

2.1) Overflow during addition

If I1 and I2 are two 32-bit signed integer values to be added, then the natural definition of overflow is any result of I1+I2 that cannot be represented as a 32-bit value. That is:-

overflow = ((I1+I2) < -pov31) OR ((I1+I2) >= pov31)

Unfortunately, when the VIPER specification was written, HOL did not support negative integers, so an alternative description in the regime of positive values was required. If I1 and I2 are represented by the two 32-bit 2's complement words R and M (as defined above), the definition of overflow given in the ADD32 function is such that an overflow is said to have occurred if bit-31 and bit-32 of the result of adding the two sign extended words together are different. This statement is to be justified in the next three sections.

Also it should be noted that the 32-bit value delivered from ADD32 is meant to be equal to the 2's complement sum of I1 and I2 in the absence of overflow. If an overflow has occurred this value has no significance.

2.1.a) Addition overflow when I1 and I2 both positive

Here R = I1, and M = I2, and the sign extension process doesn't change these values. So the sum of the sign extended words is:- SUM = I1 + I2.

Note that:- 0 <= I1 + I2 <= pov31 + pov31 - 2

There are two regions in the result space, if (I1+I2) < pov31, then no overflow has occurred, and SUM is also less than pov31, so bit-31 and bit-32 of the result are both clear. So no overflow and bit-31 and bit-32 of SUM are the same also the 32-bit result delivered = SUM.

An overflow can only occur if (I1+I2) >= pov31. This corresponds to SUM also being greater than pov31. However the maximum value of (I1+I2) is pov31-1 + pov31-1 = pov32-2. So if an overflow has occurred:-

pov31 <= SUM < pov32-1

This means that bit-31 is set but bit-32 is clear. So bit-31 and bit-32 of SUM are different when an 'overflow' has occurred.

2.1.b) Addition overflow when I1 and I2 both negative

Here R = I1 + pov32, and M = I2 + pov32. The sign extension process adds a further pov32 to both these values. The sum of the sign extended words is therefore:- SUM = I1 + I2 + pov32 + pov32 + pov32 + pov32.

Note that:- -pov32 <= I1 + I2 <= -2

There are two regions in the result space, if -pov31 <= (I1+I2) <= -2, then no overflow has occurred, and SUM is:-

SUM = pov33 + pov32 + (pov32 + I1 + I2)

Where (pov32 + I1 + I2) is in the range pov31 to pov32-2, that is the 32nd bit of the result is set, and as pov32 occurs in the definition of SUM, the 33rd is also set. So no overflow and the 32nd and 33rd bits of SUM are the same. Trimming SUM to 32-bits effectively subtracts pov33 and pov32 from SUM, so the 32-bit result delivered is (pov32 + I1 + I2), which is the 2's complement equivalent of the result.

An overflow can only occur if -pov32 <= (I1+I2) < -pov31, but SUM is:-

SUM = pov33 + pov32 + (pov32 + I1 + I2)

Where (pov32 + I1 + I2) is in the range 0 to pov31-1, that is bit-31 of the result is clear, and as pov32 occurs in the definition of SUM, bit-32 is set. So bit-31 and bit-32 of SUM are different and an 'overflow' has occurred.

2.1.c) Addition overflow when the signs of the operands are different

Under these circumstances the result of the addition can never overflow, as the range of the result is:-

-pov31 <= I1+I2 < pov31-1

The sign extension process adds a further pov32 to one value, so the sum of the sign extended words is therefore:- SUM = I1 + I2 + pov32 + pov32.

If I1+I2 is positive, its maximum value is pov31-2, so bit-31 and bit-32 of SUM are both clear. So bit-31 and bit-32 of SUM are the same and no overflow has occurred. Also trimming the result to 32-bits will deliver I1 + I2.

If I1+I2 is negative, it is in the range -pov31 to -1, so SUM is:- SUM = pov32 + pov31 + (pov31 + I1 + I2), where (pov31 + I1 + I2) is in the range 0 to pov31-1. This doesn't effect bit-31 and bit-32 of SUM which are both set. So bit-31 and bit-32 of SUM are the same and no overflow has occurred. Trimming to 32-bits will subtract the pov32 term, so the result is pov31+pov31+I1+I2, or pov32+I1+I2, the 2's complement form of the result.

So under all circumstances it has been (informally) shown that if an overflow has occurred bit-31 and bit-32 of the sign extended sum differ, but if the result is legal they are the same. Also if no overflow has occurred the result of the addition is the 2's complement form of the sum I1+I2.

2.2) Carry during unsigned addition:-

There is a natural definition of carry that could be used in HOL. That is:- CARRY = (I1+I2) >= pov32

where:- 0 <= I1 < pov32, and 0 <= I2 < pov32

Perversely, the VIPER specification doesn't use this definition, but as the proofs (5,6) were performed against a more complex definition, this will be justified here. The definition of carry in ADD32 is such that if the most significant bits of the operands are the same, then carry is the bit-32 of the 'sign extended' sum, otherwise it is the inverse of this bit. As in the case of overflow, the justification will be given in three parts.

It should be noted that the 32-bit result of ADD32 for unsigned addition is always (I1+I2) modulo pov32.

2.2.a) Addition carry when both operands are less than pov31

If I1 and I2 are the operands, SUM = I1 + I2, where 0 <= I1+I2 < pov32-1. So no carry can ever occur, and bit-32 of SUM is always clear.

The most significant bits of the operands are the same and carry is the same as bit-32 of SUM.

2.2.b) Addition carry when both operands are >= pov31

SUM = I1 + I2 + pov32 + pov32 = I1 + I2 + pov33
where:- pov32 <= I1+I2 < pov33-1.

So there is always a carry, and the bit-32 of SUM is always set.

The most significant bits of the operands are the same and carry is the same as bit-32 of SUM.

2.2.c) Addition carry when one operand < pov31 and the other >= pov31

SUM = I1 + I2 + pov32
where:- pov31 <= I1+I2 < pov32 + pov31 - 1

When pov31 <= I1+I2 < pov32, there is no carry, the I1+I2 term doesn't affect bit-32 of SUM, but the pov32 term means that this bit is set.

When pov32 <= I1+I2 < pov32 + pov31 -1, there is a carry.
SUM can be rewritten as:- SUM = pov32 + pov32 + (I1 + I2 - pov32) or
= pov33 + (I1 + I2 - pov32).
The range of (I1 + I2 - pov32) is 0 to pov31-1, so doesn't affect the bit-32 of SUM, which is therefore clear.

Hence when the most significant bits of the operands are different, bit-32 of SUM is the inverse of carry.

3) The subtraction operator SUB32

```
   NEG: word33 -> num
|- NEG  m = ((VAL33 m = 0) => 0 | (VAL33(NOT33 m) + 1)

   SUB32: word32£word32 -> word32£bool£bool
|- SUB32 (r,m) =
  (let dif      = WORD34((VAL33(SIGNEXT r)) + (NEG(SIGNEXT m))) in
   let opposite = (EL 31 (BITS32 r)) XOR (EL 31 (BITS32 m)) in
   TRIM34TO32 dif, (EL 32 (BITS34 dif)) XOR opposite,
                   (EL 32 (BITS34 dif)) XOR (EL 31 (BITS34 dif)))
```

As can be seen the subtraction operator is very similar to ADD32, but with NEG used to invert one of the operands. The effect of NEG is:-

```
For unsigned values:-
         I = 0,            NEG(SIGNEXT(W)) = 0
         0 <  I < pow31,   NEG(SIGNEXT(W)) = pow33 - I
     pow31 <= I < pow32,   NEG(SIGNEXT(W)) = pow33 - (I + pow32) = pow32 -I

For signed values:-
    -pow31 <= I <= -1,     NEG(SIGNEXT(W)) = pow33 - (I + pow32 + pow32) = -I
         I = 0,            NEG(SIGNEXT(W)) = 0
         0 <  I < pow31,   NEG(SIGNEXT(W)) = pow33 - I
```

Where W is I mapped onto a VIPER 32-bit word as discussed above.

3.1) Overflow during subtraction

If I1 and I2 are two 32-bit signed integer values to be subtracted, then the natural definition of overflow is any result of I1-I2 that cannot be represented as a 32-bit value. That is:-

overflow = ((I1-I2) < -pow31) OR ((I1-I2) >= pow31)

The definition of overflow given in the SUB32 function is such that an overflow is said to have occurred if bit-31 and bit-32 of the result of adding the sign extended and negated words together are different. This statement is to be justified in the next four sections.

The 32-bit value delivered from SUB32 is meant to be equal to the 2's complement representation of I1-I2 in the absence of overflow. If an overflow has occurred this value has no significance.

It should also be noted that in the COMPARE function, bit-32 of DIF is used as the LESS than condition (ie I1 < I2, or I1-I2 < 0). This will also be justified.

3.1.a) Subtraction overflow when I1 is positive and I2 negative or zero

DIF = I1 + (- I2)

Note that:- 0 <= I1 - I2 < pow32

There are two regions in the result space. If (I1-I2) < pow31, then no overflow has occurred, and DIF is also less than pow31, so bit-31 and bit-32 of the result are both clear. So bit-31 and bit-32 of DIF are the same and no overflow has occurred, and the 32-bit result delivered = DIF.

An overflow can only occur if (I1-I2) >= pow31. This corresponds to DIF also being greater than pow31. However the maximum value of (I1-I2) is pow31-1 - (-pow31) = pow32-1. So if an overflow has occurred:-

pow31 <= DIF < pow32

This means that bit-31 is set but bit-32 is clear. So bit-31 and bit-32 of DIF are different and an overflow has occurred.

Note that in both cases, LESS is always false and bit-32 of DIF is clear.

3.1.b) Subtraction overflow when I1 is negative and I2 is greater than zero

DIF = (I1 + pow32 + pow32) + (pow33 - I2)

Note that:- -(pow32-1) <= I1 - I2 <= -2

There are two regions in the result space. If -pow31 <= (I1-I2) <= -2, then no overflow has occurred, and DIF is:-

DIF = pow33 + pow32 + (pow32 + I1 - I2)

Where (pow32 + I1 - I2) is in the range pow31 to pow32-2, that is bit-31 of the result is set, and as pow32 occurs in the definition of DIF, bit-32 is also set. So bit-31 and bit-32 of DIF are the same and no overflow has occurred. Trimming DIF to 32-bits effectively subtracts pow33 and pow32 from DIF, so the 32-bit result delivered is (pow32 + I1 - I2), which is the 2's complement equivalent of the result.

An overflow can only occur if -(pow32-1) <= (I1-I2) < -pow31, but DIF is:-

DIF = pow33 + pow32 + (pow32 + I1 - I2)

where (pow32 + I1 - I2) is in the range 1 to pow31-1, that is bit-31 of the result is clear, and as pow32 occurs in the definition of DIF, bit-32 is set. So bit-31 and bit-32 of DIF are different and an overflow has occurred.

Note that in both cases LESS is true, and bit-32 of DIF is set.

3.1.c) Subtraction overflow when I1 is negative and I2 is negative or zero

Under these circumstances the result of the subtraction can never overflow, as the range of the result is:-

-pow31 <= I1-I2 < pow31

DIF = (I1 + pow32 + pow32) + (-I2) = pow33 + I1 - I2

If I1-I2 is positive, its maximum value is pow31-1, so bit-31 and bit-32 of DIF are both clear. So bit-31 and bit-32 of DIF are the same and no overflow has occurred. Trimming the result to 32-bits will deliver I1 - I2. LESS is false and bit-32 of DIF is clear.

If I1-I2 is negative, it is in the range -pow31 to -1, so DIF is:- DIF = pow32 + pow31 + (pow31 + I1 - I2), where (pow31 + I1 - I2) is in the range 0 to pow31-1. This doesn't affect bit-31 and bit-32 of DIF which are both set. So bit-31 and bit-32 of DIF are the same and no overflow has occurred. Trimming to 32-bits will subtract the pow32 term, so the result is pow31+pow31+I1-I2, or pow32+I1-I2, the 2's complement form of the result. LESS is true and bit-32 of DIF is set.

3.1.d) Subtraction overflov vhen I1 is positive and I2 is greater than zero

Under these circumstances the result of the subtraction can never overflov, as the range of the result is:-

-(pov31-1) <= I1-I2 < pov31-1

DIF = I1 + (pov33 - I2) = pov33 + I1 - I2

The arguments used in 3.1.c then follov.

So under all circumstances it has been (informally) shovn that if an overflov has occurred bit-31 and bit-32 of the sign extended difference differ, but if the result is legal they are the same. Also if no overflov has occurred the result of the subtraction is the 2's complement form of the sum I1-I2, and bit-32 of DIF corresponds to the value LESS.

3.2) Borrov during unsigned subtraction:-

The natural definition of borrov (the subtraction's analogy to addition's carry) is:- BORROV = (I1-I2) < 0

where:- 0 <= I1 < pov32, and 0 <= I2 < pov32

The definition of borrov in SUB32 is such that if the most significant bits of the operands are the same, then borrov is bit-32 of the 'sign extended' difference, othervise it is the inverse of this bit.

It should be noted that the 32-bit result of SUB32 for unsigned subtraction is alvays (I1-I2) modulo pov32.

3.2.a) Subtraction borrov vhen I1 < pov31 and I2 = 0

DIF = I1, vhere 0 <= I1-I2 < pov31
So no borrov can ever occur, and bit-32 of DIF is alvays clear.

The most significant bits of the operands are the same and borrov is the same as bit-32 of DIF.

3.2.b) Subtraction borrov vhen I1 >= pov31 and I2 = 0

DIF = I1 + pov32, vhere pov31 <= I1-I2 < pov32

or:- DIF = pov32 + pov31 + (I1 - I2 - pov31), vhere 0 <= I1-I2-pov31 < pov31

So no borrov can ever occur, and bit-32 of DIF is alvays set.

The most significant bits of the operands are different and borrov is the inverse of bit-32 of DIF.

3.2.c) Subtraction borrov vhen I1 < pov31 and I2 >= pov31

```
   DIF = I1 + (pov32 - I2)
   vhere:- -(pov32-1) <= I1-I2 <= -1
   and:-   1 <= DIF < pov32
   So there is alvays a borrov, but bit-32 of DIF is alvays clear.
```

The most significant bits of the operands are different and borrov is the inverse of bit-32 of DIF.

3.2.d) Subtraction borrov vhen I1 >= pov31 and 1 <= I2 < pov31

```
   DIF = (I1 + pov32) + (pov33 - I2) = pov33 + pov32 + (I1-I2)
   vhere:- 1 <= I1-I2 <= pov32-2
   So there is never a borrov, and bit-32 of DIF is alvays set.
```

The most significant bits of the operands are different and borrov is the inverse of bit-32 of DIF.

3.2.e) Subtraction borrov vhen I1 < pov31 and 1 <= I2 < pov31

```
   DIF = I1 + (pov33 - I2), vhere:- -(pov31-1) <= I1-I2 <= pov31-2

   vhen:- -(pov31-1) <= I1-I2 <= -1, there has been a borrov and
         DIF = pov32 + pov31 + (pov31 + I1 - I2)
         The range of (pov31 + I1 - I2) is 0 to pov31-1, so it cannot affect
         bit-32 of DIF, vhich can be seen to be set.

   vhen:- 0 <= I1-I2 <= pov31-2, there has not been a borrov and
         DIF = pov33 + (I1 - I2)
         The range of (I1 - I2) is 0 to pov31-2, so it cannot affect bit-32
         of DIF, vhich can be seen to be clear.
```

The most significant bits of the operands are the same and borrov is the same as bit-32 of DIF.

3.2.f) Subtraction borrov vhen I1 >= pov31 and I2 >= pov31

```
   DIF = (I1 + pov32) + (pov33 - (I2 + pov32)) = pov33 + I1 - I2
   vhere:- -(pov31-1) <= I1-I2 <= pov31-1
```

The arguments used in 3.2.e still apply (noting that pov31-2 is replaced by pov31-1, vhich doesn't change any of the subsequent reasoning)

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) ....

Overall security classification of sheet ..................................UNCLASSIFIED..

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No. REPORT 90009	Month JUNE	Year 1990

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
FORMAL SPECIFICATION OF THE VIPER MICROPROCESSOR IN HOL

Report Security Classification UNCLASSIFIED	Title Classification (U, R, C or S) U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period
Project Number	Other References
Authors PYGOTT, C H	Pagination and Ref vp

Abstract

This report provides a mathematically rigorous specification of the required behaviour of the VIPER microprocessor in the HOL notation (Higher Order Logic) of Cambridge University. This specification has been used as the starting point for a chain of proofs, in an attempt to show that a number of implementations of this specification are indeed correct.

This report replaces the early RSRE Report 85013, which describes VIPER in the language LCF-LSM (a precursor to HOL).

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED